University of California
Los Angeles

Coprocessor Acceleration for Domain-Specific Computing

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Yi Zou

2012

Abstract of the Dissertation

Coprocessor Acceleration for Domain-Specific Computing

by

Yi Zou

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2012

Professor Jason Cong, Chair

There is a growing trend to use coprocessors to offload and accelerate domain-specific applications in order to obtain significant performance improvement and energy/power reductions. Two important coprocessor components in the heterogeneous system are the GPU and FPGA. GPU (graphics processing unit) is increasingly used as a data-parallel coprocessor for general computations. The newest GPU has a much larger number of cores (compared to CPU) and very high peak FLOPS. FPGA (field programmable gate array), on the other hand, allows users to customize, at fine-grain level, the computational data path and memory hierarchy according to the exact need of the applications. FPGA excels in integer operations and bit-level operations.

The thesis starts with several coprocessor acceleration examples for our focus application domains: the first domain is on VLSICAD algorithms and the second is on computational medical imaging. We detail application acceleration examples in the domains including lithography simulation for IC manufacturing, medical image reconstruction using compressive sensing, and medical image registration using fluid models. Both GPU-accelerated versions and FPGA-accelerated versions have been implemented. Based on these implementations, we then analyze the performance and energy trade-offs, the interaction between the diverse application requirements and a spectrum of hardware systems, and how those domain-specific coprocessor acceleration case studies further bring us insights for domain-

specific architecture innovations. In the end, we showcase an example for collaborative execution on the heterogeneous platform. Different scheduling policies are needed to optimize performance or energy. The thesis concludes as we present reusable architecture templates and realizations for futuristic accelerator-rich CMPs.

The dissertation of Yi Zou is approved.

Alex Bui

Milos Ercegovac

Glenn Reinman

Jason Cong, Committee Chair

University of California, Los Angeles

2012

To my parents . . . who raised me

To all the people who love me and support me

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my adviser Prof Jason Cong for the insightful guidance and continuous help during the PhD study. It is not possible to have the thesis without our weekly group meetings and countless Email exchanges (with him) which inspire the research ideas and pinpoint the focus. The thesis is further strengthened through numerous iterations of paper/thesis revisions. I would also like to thank my committee members Prof Alex Bui, Prof Milos Ercegovac and Prof Glenn Reinman for their constructive and critical feedback for my thesis. I am very grateful for Janice Martin-Wheeler who helped the language editing of my papers and the thesis. The thesis also benefits from various discussions from current and former lab members: Jianwen Chen, Mohammadali Ghodrat, Karthik Gururaj, Hui Huang, Muhuan Huang, Wei Jiang, Bin Liu, Chunyue Liu, Bingjun Xiao, Ming Yan, Bo Yuan and Peng Zhang.

The research conducted in this thesis is generously supported by funds from UC MICRO program, UC Discovery program (with industry support from Altera Corp, Magma Design Automation, Nvidia Corp and Mentor Graphics Corp), and NSF Grant CCF-0926127 NSF Center for Domain-Specific Computing (CDSC). Some chapters are based on materials of my published works. Chapter 3 is based on the adaption of paper P4. Alfred Wong from Magma Design Automation gave us sample kernels data and engaged in valuable discussions. The XD1000 development system is obtained through Xtremedata university program. Chapter 4 is based on the adaption of paper P8. Igor Yanovsky and Prof Luminita Vese at UCLA department of mathematics provide us the reference source code for the fluid registration implementation. Chapter 5 is based on an extended version of paper P10. Ming Yan and Prof Luminita Vese at UCLA department of mathematics provide us the reference source code for the EMTV reconstruction implementation. Chapter 6 is based on the rewrite of materials in papers P6 and P11. The CnC-HC tool and the Habanero runtime are provided by Rice Habanero project as part of the CDSC effort. Xilinx Inc and AutoESL (now part of Xilinx Inc) donate us various FPGA devices/boards and licenses for the design tools (including high-level synthesis tools) through the Xilinx university program.

Vita

2000–2004	B.S., Department of Computer Science and Technology Tsinghua University, Beijing, P.R.China
2004–2006	M.S., Department of Computer Science and Technology Tsinghua University, Beijing, P.R.China

Publications

P1. J. Cong and Y. Zou, "Lithographic Aerial Image Simulation with FPGA-Based Hardware Acceleration", Proc. 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2008), pages 67-76, Monterey, CA, Feb 2008.

P2. J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou and Y. Zou, "Evaluation of Static Analysis Techniques for Fixed-Point Precision Optimization", Proc. 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2009), pages 231-234, Napa, CA, Apr 2009.

P3. J. Cong, W. Jiang, B. Liu, and Y. Zou, Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization, in Proc. International Conference on Computer-Aided Design (ICCAD 2009), pages. 697-704, San Jose, CA, Nov 2009.

P4. J. Cong and Y. Zou, "FPGA-Based Hardware Acceleration of Lithographic Aerial Image Simulation", ACM Transactions on Reconfigurable Technology and Systems, Vol.2, No.3, Article 17, Sep 2009.

P5. J. Cong and Y. Zou, "Parallel Multi-level Analytical Global Placement on Graphics Processing Units", Proc. International Conference on Computer-Aided Design (ICCAD 2009), pages 681-688, San Jose, CA, Nov 2009.

P6. J. Cong and Y. Zou, "A Comparative Study on the Architecture Templates for Dynamic Nested Loops", Proc. 18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010), pages 251-254, Charlotte, NC, May 2010.

P7. J. Cong, M. Huang and Y. Zou, "3D Recursive Gaussian IIR on GPU and FPGAs", Proc. IEEE Symposium on Application Specific Processors (ASAP 2011), San Diego, CA, Jun 2011.

P8. J.Cong, M. Huang and Y. Zou, "Accelerating Fluid Registration Algorithm on Multi-FPGA Platforms", Proc. International Conference on Field Programmable Logic and Applications (FPL 2011), Chania, Crete, Greece, Sep 2011.

P9. A. Bui, K.Cheng, J. Cong, L. Vese, Y. Wang, B. Yuan and Y. Zou, "Platform Characterization for Domain-Specific Computing (invited paper)", Proc. Conference on Asia South Pacific Design Automation (ASPDAC 2012), Sydney, Australia, Jan 2012.

P10. J. Chen, J. Cong, M.Yan and Y. Zou, "FPGA-Accelerated 3D Reconstruction using Compressive Sensing", Proc. International Symposium on Field-Programmable Gate Arrays (FPGA 2012), Monterey CA, Feb 2012.

P11. A. Sbirlea, Y. Zou, Z. Budimlic, J. Cong and V. Sarkar, Mapping a Data-Flow Programming Model onto Heterogeneous Platforms, Proc. ACM SIGPLAN/SIGBED Conference on. Languages, Compilers, Tools and Theory for Embedded Systems (LCTES 2012), Beijing, China, Jun 2012.

CHAPTER 1

Introduction

1.1 Motivation

Recently, the frequency scaling of microprocessors has slowed down, and the trend is that more and more cores are being put in a single chip. Programs need to actively leverage parallel computing to make the best use of available computing resources.

On the other hand, the conventional multi-core system may not be the best architecture realization for a particular application specification. These multi-core systems are general-purpose in nature and are typically less power-efficient. As the number of cores continue to increase, the power density of the system creates the utilization wall [1] where not all the components can be active all the time. The term "dark silicon" [2] is used to refer to the transistor underutilization on the future many-core chips. This motivates us to take the domain-specific approach to customize the hardware platform in order to improve the power/performance of the computing system. Off-the-shelf coprocessors such as FPGAs and GPUs can offload certain computation-intensive parts of the application, and can significantly accelerate overall system performance. GPU is a device that is traditionally used for rendering images for computer display and gaming purposes, but it is also used as a data-parallel coprocessor for general computations. The newest GPUs have a much larger number of cores (compared to CPUs) and very high peak FLOPS. The FPGA, on the other hand, allows users to customize, at fine-grain level, the computational data path and memory hierarchy according to the exact need of the applications. FPGA excels in integer operations and bit-level operations. While the specialization using the coprocessors does bring in further under-utilization, the superior energy-efficiency of the coprocessors creates

an effective means to battle the utilization wall.

The research is partially funded by the CDSC (Center for Domain-Specific Computing) [3] project. The vision of the CDSC project is to look beyond parallelization, adapt the hardware architecture to application domains, and focus on domain-specific customization as the next disruptive technology to bring orders-of-magnitude power-performance efficiency improvement to important application domains. Using this technology, we can potentially build a "supercomputer in a box" to illustrate the energy-efficient computing. As part of the effort of the CDSC project, we have built a heterogeneous testbed that contains both FPGA and GPU in a tightly integrated fashion. Still, the challenges for using the coprocessors are two-fold.

First, we need to obtain the component-specific implementation efficiently. Because the hardware architecture of the coprocessor may be completely different from the traditional multi-core, different implementation schemes for each coprocessor component are likely needed. For the GPU, we use the Nvidia CUDA toolkit, which is a C/C++ language augmented to support the single program multiple data (SPMD) paradigm. For the FPGA, we use the AutoESL high-level synthesis (HLS) tools to describe the bare-metal hardware accelerator in hardware-oriented C-language. The CUDA toolkit (or the more general cross-vendor API OpenCL) has been widely accepted in the GPGPU community. We have implemented several interesting applications using the GPU: mixed-size circuit global placement [4], and MRI image reconstruction using compressive sensing. But for the FPGA, the C-based high-level synthesis has not yet reached very broad adoption. The frustration can be seen in the FCCM'11 top 10 predictions: "No. 1 prediction: a high-level, object-oriented solution for FPGA design will be popular, it won't be C to gates; and the No. 5 prediction: VHDL and Verilog will remain ubiquitous for circuit expression." Yet we will allocate a major portion of this thesis to present several FPGA-based coprocessor accelerator cases to advocate the HLS development flow for the FPGA. I will detail the optimization scheme used (at C-level) to obtain a design implementation with the best performance, and the potential performance/energy benefit. The limitations and work-arounds of the current flow also motivate us to pursue further automations.

The second challenge is to manage those coprocessor accelerators efficiently, assuming the component-specific implementation is ready. For example, we map a medical imaging pipeline that includes image denoising, image registration and segmentation onto the heterogeneous platform. Efficient implementations on either FPGA or GPU can speed up each individual application by 10X to 20X respectively, and with a 2X to 4X improvement on energy-efficiency.

When we have a batch of images to process, the task-level parallelism in the macro dataflow permits concurrent execution on multiple processors as well as coprocessors (including GPU and FPGA). Moreover, some application kernels are more suitable for FPGA coprocessors, some are more suitable for GPUs, and some portion of the codes shall still be left to CPUs to execute. We need to consider these preferences of kernel binding as well as resource availability in the mapping.

By using the task-level data flow description called Concurrent Collections (CnC) [5], and the underlying Habanero run-time, we can achieve the load-balancing (through work-stealing) across different heterogeneous components. A dynamic task to coprocessor binding (with cross-device stealing) can boost the overall accelerated performance compared to a static binding. However, the Habanero runtime works in user-space and will not perform any coprocessor/accelerator management at the whole system-level. We then present a practical architecture template implementation that mimics the features of accelerator-rich CMPs. We realize a global accelerator manager (GAM) using an embedded processor. The embedded processor performs the actual accelerator invocation and is the gateway/abstraction-layer for the application code and device drivers.

1.2 FPGA/GPU Coprocessor Implementation: Lithography Simulation, Fluid Registration and CT Image Reconstruction

In this section we briefly highlight our achievements of the three applications that are described in this thesis.

Lithography simulation, as an essential step in design for manufacturability (DFM), is still far from computationally efficient. Most leading companies use large clusters of server computers to achieve acceptable turn-around time. Thus coprocessor acceleration is very attractive for obtaining increased computational performance with reduced power consumption. We designed a customized accelerator on an FPGA using a polygon-based simulation model. An application-specific memory partitioning scheme is designed to meet the bandwidth requirements for a large number of processing elements. Deep loop pipelining and ping-pong buffer based function block pipelining are also implemented in our design. A 15X speedup is obtained using the FPGA-device in the Xtremedata XD1000 system versus the software implementation running on a microprocessor of the same system. The performance is 2X that of a CUDA implementation GPU 8800GT, yet the FPGA only consumes about a fraction (1/20) of the power of the GPU. The implementation also leverages state-of-art C-to-RTL synthesis tools. At the same time, we also identified the need for manual architecture-level exploration for parallel implementations—with the most important one being memory partitioning.

In the clinical applications, medical image registrations on the images taken from different times and/or through different modalities are needed in order to have an objective clinical assessment of the patient. Viscous fluid registration is a powerful PDE-based method that can register large deformations in the imaging process. We present our implementation of the fluid registration algorithm on the multi-FPGA platform Convey HC-1. We obtain a 35X speedup versus single-threaded software on a CPU, and comparable performance with GPU (Telsa C1060). The implementation is achieved using a high-level synthesis (HLS) tool, with additional source-code level optimizations including fixed-point conversion, tiling, prefetching, data-reuse, and streaming across modules using a ghost zone (time-tiling) approach. These manual steps need to be further automated by existing HLS software.

The radiation dose associated with computerized tomography (CT) is significant. Compressive sensing methods provide mathematic approaches to reduce the radiation exposure, without sacrificing image quality. However, the computational requirement of the algorithm is prohibitive, and much higher than conventional image reconstruction algorithms like the

Feldkamp-Davis-Kress (FDK) algorithm. We present an FPGA implementation of one compressive sensing algorithm with applications on CT image reconstruction. The ray tracing forward and backward projection procedures have abundant random off-chip accesses, as well as load-balancing issues, and are good fit for the multi-FPGA platform that excels in interleaved memory access. Our FPGA EM kernel is 50% faster than the GPU implementation on Telsa C1060. Moreover, our FPGA kernel can process two independent images at the same time, and thus the kernel throughput is 3X that of Tesla C1060 and slightly better than the Fermi GTX480 GPU. Moreover, our EM kernel is deployed in a hybrid (CPU+GPU+FPGA) computer, and we show that the hybrid approach delivers better performance and energy than GPU-only solutions.

1.3 Managed Task-level Parallelism for heterogeneous Computing

We need a diverse and customizable architecture to meet the needs of diverse applications. To address the needs, we built a heterogeneous testbed that contains both the FPGA and GPU in a tightly integrated fashion. From the perspective of application-side, we need to consider two issues: first, the coprocessors introduce additional computational elements and thus we can further explore task-level parallelism where tasks on the CPU, GPU and FPGA can potentially run in parallel; second, some kernels are more suitable for FPGA coprocessors, some are more suitable for GPUs, and some portion of the codes shall still be left to CPU to execute.

We use the modeling language chosen by the Center for Domain Specific Computing (CDSC): Concurrent Collection [5], which develops a software flow that is capable of converting high-level task-level dataflow into a lower-level *async-finish* style parallel code. We demonstrate a working example that maps a medical image processing pipeline onto one heterogeneous platform with CPU, GPU and FPGAs. The limitation is that the management scheme works within a single process. We then discuss the architecture template and support that works at the system-level, which can move some scheduling logic from software into hardware.

CHAPTER 2

Coprocessor Acceleration

The goal of the Center for Domain-Specific Computing (CDSC) [3] is to develop domain-specific hardware architectures, and the software systems to greatly improve the performance and the energy efficiency of domain-specific applications. In this thesis we focus on two application domains. The first one is VLSICAD, because many of the algorithms in the domain are quite computational intensive and requires heavyweight clusters to compute. The second one is medical imaging, given its significant impact on the healthcare industry. To achieve our goal, it is essential to benchmark how well current commodity platforms perform, and identify opportunities for architectural innovations.

2.1 General-Purpose Computing on GPUs (GPGPU)

The first type of accelerator component is the graphics processing unit or GPU. The GPU is the processing unit specifically designed for handling the computation for computer graphics. It is increasingly used as massive-parallel many-core processor that can aid general purpose computation.

2.1.1 Graphics Processing Units (GPU)

Conventional GPU contains a number of processors for vertex processing, texture processing and fragment pipeline. Texture processing is normally vector-based as textures will contain RGB components for visualization.

The Nvidia GT200 series GPU uses a unified shader architecture, so that tasks for vertex processing, fragment pipelining and texture processing are all performed by the unified shad-

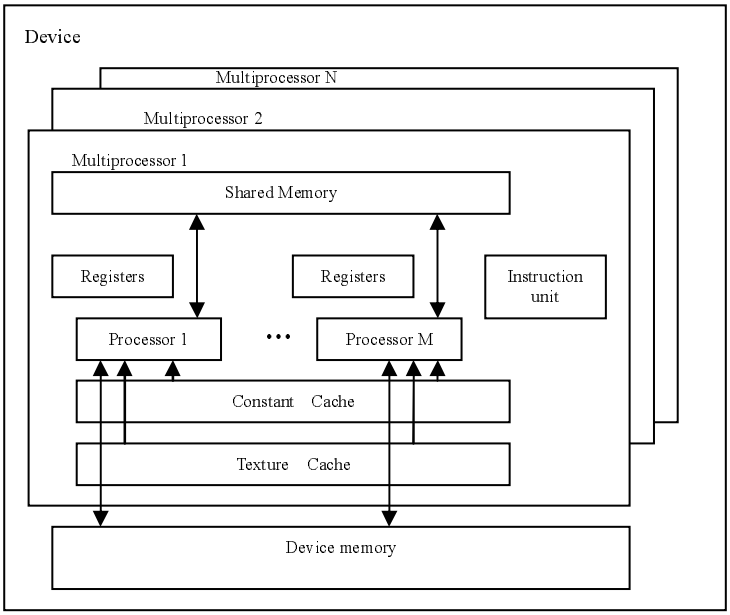


Figure 2.1: Hardware model of GT200 series GPU

er.[1] Also, it uses scalar processors rather than vector processors; thus it is much friendlier for general computing tasks. Figure 2.1 shows the hardware model for the GT200 series GPU. Each device is composed of N stream multiprocessors (SM), and each multiprocessor is composed of M stream processors (SP). Each processor has its dedicated registers. The M stream processors in each multiprocessor share a fast-access scratchpad memory called "Shared Memory" (as shown in Figure 2.1), and share the instruction unit. These M processors run exactly the same threaded program in the SPMD (single program multiple data) model. The constant cache and texture cache are also shared by the M processors in the same multiprocessor. Compared to the previous generation G80 series, GT200 also adds the double precision support. Each multiprocessor has one double precision unit.

Another major vendor of the graphics compute card is AMD. The overall architecture of the AMD GPU compute card is very similar to those from Nvidia. Yet they may have different ratio of arithmetic units, special function units and texture shading units. One major difference is that the AMD card uses 4-way or 5-way VLIW at the finest level, where the Nvidia card uses the scalar processor. AMD's recent Graphics Core Next (GCN) ar-

[1]GT200 is the codename for the latest version of Nvidia GPU. GTX280 and Tesla C1060 are specific names of the video cards that use the GT200 series of GPU.

chitecture also adopted the scalar architecture. A detailed comparison between the Nvidia GPU architecture and the AMD GPU architecture can be seen in [6].

Modern GPU compute cards sit on the PCI-express slots and talk to the host CPU using PCI-express protocols. The 2.0 version of the PCI-express will have 500MB/s peak bandwidth per lane and 8GB/s bandwidth for a 16-lane connection. A 16-lane connection is what we use in most modern GPU compute cards. Note the PCI-e bandwidth is still significant lower than the peak bandwidth of the on-board off-chip memory bandwidth of the GPU card, which can be around 100GB/s.

2.1.2 GPU Programming Environment

Writing general programs for GPUs requires an in-depth understanding of the graphics concepts and a conversion from the general computing task to a graphics processing pipeline. This, often, is not an intuitive and easy task.

CUDA is the C environment for programming the G80, G92 and GT200 series GPUs. It uses massive threading to overlap the computing with the memory access latency, and it also provides the scratchpad-like shared memory for fast access (note the size of the shared memory is very limited). Thread creation and switching are very lightweight on the GPU compared to a multi-core CPU. In the CUDA environment, users can write SPMD code to map onto the hardware architecture. Inside the code, users can define the parallelism among different multiprocessors by specifying the number of parallel blocks to use: N_block. Also the parallelism inside each multiprocessor is specified by the number of threads inside each block: N_thread. A total of $N\_block * N\_thread$ threads shall be created for that code. The block ID and thread ID are used to identify different threads, and they can access different data based on their IDs. Each thread can access per-thread registers, per-thread-block shared memory, per-thread-block cache and per-device global memory.

The hardware architecture of the Nvidia GPUs also has certain limitations. Because of the SIMD fashion used in the stream multiprocessor, there is significant overhead in handling divergent control flows. No synchronization between thread-blocks is provided

except through global memory on the board. Memory access needs to follow a special access pattern (called coalesced access) where a half-warp (16 in the GT200 architecture) of threads shall access continuous memory locations for optimal memory bandwidth. These architecture restrictions impose considerable challenges for implementation, and require significant tuning efforts for better performance.

Another general-purpose programming framework for GPU computing is OpenCL [7]. OpenCL is an open, industry-standard framework, which has gained support from almost all CPU/GPU manufacturers including AMD, Intel, and NVIDIA; CUDA is only supported on Nvidia GPUs. The GPU implementations in this thesis are done using CUDA rather than OpenCL, because at that time the CUDA toolkit is more mature than the Nvidia OpenCL toolkit. While the two share a similar programming paradigm, the OpenCL program is slightly lower-level (in order to achieve platform neutral) and is more lengthy to write.

2.2 Reconfigurable Computing

Reconfigurable computing leverages the specialized hardware platforms to adapt the computational need through hardware customization. In the 1960s, pioneering work by Prof Gerald Estrin at UCLA proposed the architecture that features the fixed plus reconfigure hardware [8, 9]. Similarly, the Stanford GARP project [10] combines a RISC processor with a reconfigurable hardware. Most modern reconfigurable systems use the field programmable gate array (FPGA) as the major component to realize the customizable hardware kernel. A typical FPGA architecture consists of an array of logic blocks (called configurable logic block, CLB), I/O pads, and routing channels. The logic block uses a lookup-table (LUT) to realize arbitrary bit-level logical expressions. The routing channel is also programmable at bit-level. Modern FPGAs also incorporated several hardware blocks such as block RAM, DSP units, and carry chains. High-end FPGAs now feature thousands of block RAMs, where multiple block RAMs can provide parallel data accesses simultaneously. Because of the nature of FPGA, it is well suited to accelerate bit-level, fixed-point parallel computation; though floating point arithmetic is also supported via efficient utilization of DSP units in

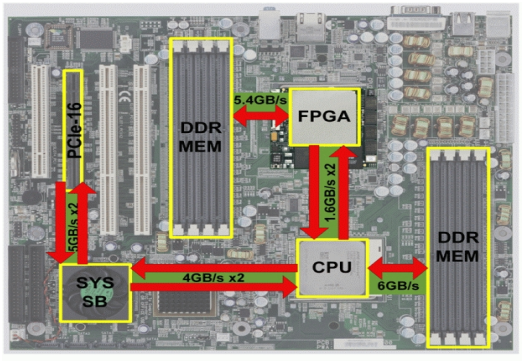


Figure 2.2: XD1000 system diagram

modern FPGAs.

The principle of customizable computing is not limited to FPGA-based computing; also applies to other reconfigurable systems that use coarse-grain reconfigurable logics. Several more recent reconfigurable architectures are further surveyed in [11]. The book by [12] presents comprehensive examples on reconfigurable computing.

This year, we celebrated the 20th anniversary of FCCM, which is a conference that is specifically dedicated to reconfigurable computing. We have seen many successful examples for reconfigurable computing, such as networking and communications [13], image processing [14], finance [15], data-warehousing [16], bioinformatics [17], radar and remote sensing [18] etc.

The case studies we present in the following chapters also showcase successful examples of reconfigurable computing where we target VLSICAD and medical imaging.

2.2.1 FPGA-Centric Accelerator Systems

FPGA accelerators still have no standard packaging or interconnect protocols, and thus a number of choices are available. Several cards (e.g., Xilinx ML605, Altera DE4, Nallatech PCIe-280) also provide PCI-express edges. These may be the easiest integration options, but typically only one or two SODIMMs are attached to these cards and the bandwidth from the SODIMMs is relatively poor compared to the bandwidth of GPUs. Axel [19] and QP [20] all use this type of FPGA accelerator.

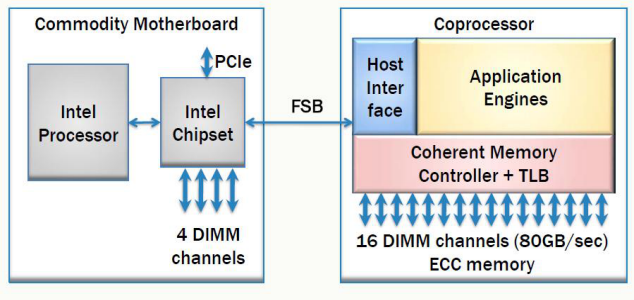


Figure 2.3: Overall system diagram of Convey HC-1 hybrid computer

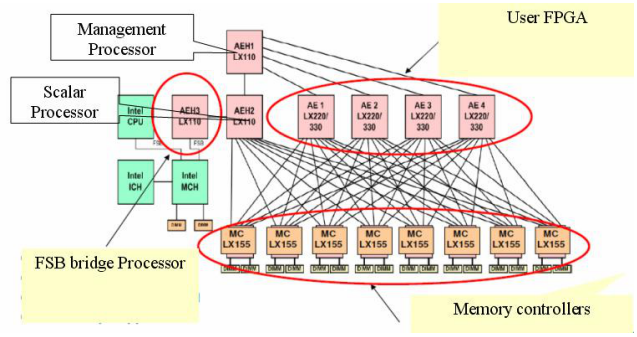


Figure 2.4: Coprocessor-side diagram of Convey HC-1 hybrid computer

In the lithographic simulation acceleration, we used the XD1000 system from XtremeData. Figure 2.2 is the system diagram of XtremeData's XD1000 development system, which is the hardware platform we use. This development system uses a dual Opteron motherboard and one Opteron is replaced by an XD1000 coprocessor module. The XD1000 coprocessor communicates with the host Opteron CPU via Hypertransport links, and it is built based on Altera's largest FPGA in the Stratix II family: EP2S180.

The applications in the medical imaging domain are very memory-intensive. Because the size of the 3D images is too big to fit in the on-chip RAM or on-board SRAM chip, we need to use off-chip DRAM extensively. Some cards (e.g., SRC Map station) use memory DIMM slots with proprietary protocol. The more compelling choices are the FPGA accelerators that use processor interconnects. AMD opened its HyperTransport bus through its Torrenza initiative, and Intel later opened its Front Side Bus to third party FPGA board vendors as well. Representative systems include Xtremedata's XD1000F and XD2000F (using HyperTransport), Nallatech's FSB development system, and Convey HC-1 (using FSB).

Among these systems, we further selected Convey HC-1(ex) as our baseline platform for implementing medical imaging applications. Figure 2.3 shows the system diagram for the Convey HC-1. The form-factor of the platform is a 2-U rack-mountable server box. Figure 2.4 shows the structures of the coprocessor hardware of the Convey HC-1. The HC-1 platform has four user FPGAs (Virtex5 LX330). The CPU and different FPGAs access the off-chip memory using a shared memory model. The system employs an on-board crossbar to realize the interconnection. Cache coherence is also handled through the FSB protocols. Convey realizes a cache-coherent NUMA (ccNUMA) system on top of a FSB protocol stack. Each FPGA is presented with 16 external memory access channels. (Eight physical memory ports are connected to eight memory controllers that run at 300MHZ. Core design runs at 150MHZ. Thus, effectively the design on each FPGA is presented with 16 "logical" memory access channels through time multiplexing.) Convey HC-1 provides a very large bandwidth (claims to have 80GB/s peak bandwidth) for coprocessor side memory. Note, a claimed 80GB/s coprocessor side memory bandwidth in Convey HC-1 is only slightly lower than the bandwidth of one Tesla C1060 card (around 100GB/s). In practice, we observe that 30% to 50% of the peak bandwidth can be achieved. As a comparison, the Xilinx ML605 board has only one SODIMM and the off-chip sustained memory bandwidth from SODIMM measured less than 1GB/s. In the Nallatech FSB development system, the FPGA coprocessor does not have a large coprocessor-side memory but accesses the system memory through the front-side bus and the bandwidth is limited to the bandwidth of FSB (around 8GB/s).

The multi-FPGA platform Convey HC-1(ex) uses an interleaved memory scheme. Different FPGAs access the off-chip memory using a shared memory model. The system employs an on-board crossbar to realize the interconnection. In HC-1ex, the user FPGAs are upgraded from Virtex5 LX330 to Virtex6 LX760. The system supports two modes of an interleave scheme. The prime number interleave is where the system uses a prime number of banks to better support power-of-two strides. The other interleave scheme is called the binary interleave.

The binary interleave scheme takes out the 12th to 3rd bits (counted from the LSB) of the virtual address to determine the bank the address falls in. The system features 1024

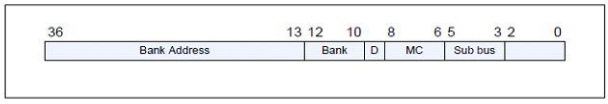


Figure 2.5: Binary interleave

banks. In particular, the 9th to 6th bits of the address determine which memory DIMM the address falls in. The system has 16 DIMMs in total. Figure 2.5 illustrates the binary interleave scheme. We can see from Figure 2.4 that each memory controller is connected to two DIMMs. The D bit in Figure 2.5 determines which DIMM to select from the two DIMMs. Each DIMM has 64 banks.

2.2.2 High-Level Synthesis Tools

One of our unique approaches on reconfigurable computing is through the use of high-level synthesis tools. As the complexity of FPGA designs goes up, it is more and more challenging to implement the design manually using RTL-level descriptions such as Verilog or VHDL. Electronic system-level (ESL) is a cutting-edge technology which allows users to describe their designs at high level, such as C/C++, and ESL tools further perform high-level synthesis (HLS) to generate platform-specific RTL automatically. The adoption of these tools leads to a significant savings in design effort and reduced turn-around time.

Currently there are many ESL products in the market, such as Impulse C [21] from Impulse Accelerated Technologies, Catapult C [22] from Calypto Design Systems, DK Suite and Handel-C [23] from Agility (now part of Mentor Graphics), Cynthesizer [24] from Forte Design Systems, AutoPilot [25] from AutoESL Design Technologies (now part of Xilinx), PICO Extreme FPGA [26] from Synfora (now part of Synopsys), C-to-Silicon Compiler [27] from Cadence, just to name a few.

AutoPilot [25] is the ESL tool that we use in this thesis, because it uses the latest technology and leverages a robust open compiler framework; it features a rich set of optimization passes and compiler transforms for hardware synthesis; it also has a very wide C/C++ lan-

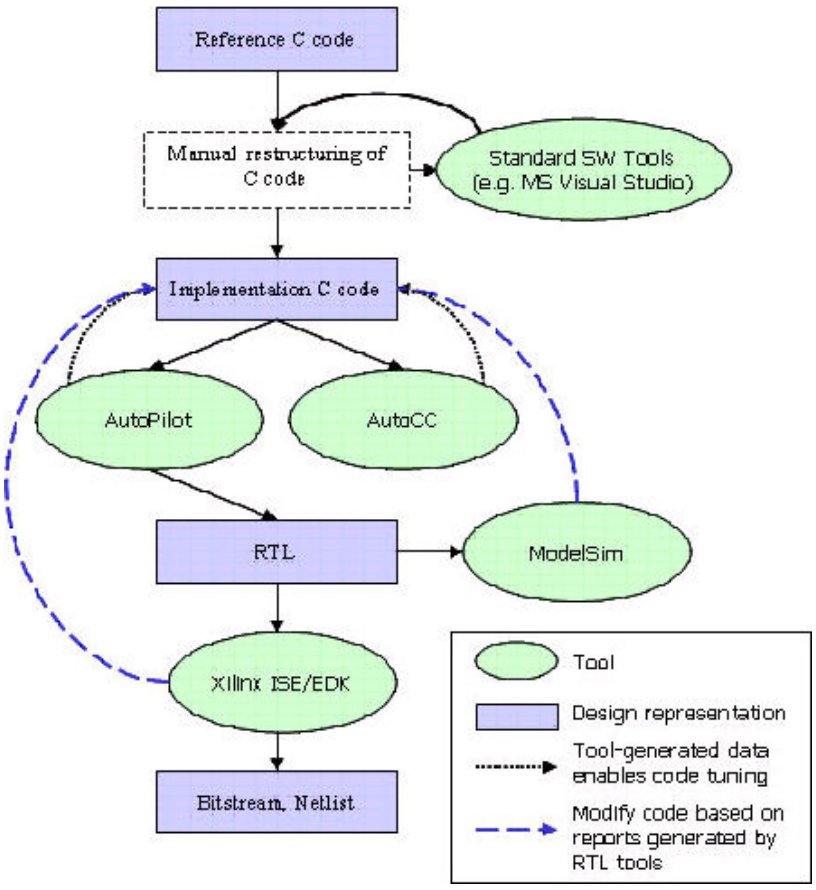


Figure 2.6: Design flow using AutoPilot HLS tools

guage support. (Also it is the tool that we have access to.) It accepts synthesizable C, C++, or SystemC as input and generates RTLs in Verilog or VHDL, while we primarily use C as input. AutoPilot is based on the xPilot system [28] licensed from UCLA, but has been commercialized by AutoESL Design Technologies for further development. Figure 2.6 (from BDTI certification report [29]) shows the design flow using AutoPilot tools. Starting from the C code, we perform hardware-oriented code refinement at C-level to obtain a bit-accurate synthesizable code. Simulation can be done by pre-synthesis *autocc* tool, or at RTL-level using third-party simulators. The RTL is further synthesized to obtain the bitstream through vendor tools (e.g., Xilinx ISE).

The AutoPilot tool will only generate the RTL for one IP. Designers need to connect the IP with other IPs or system interfaces. Platform-specific interface wrappers in RTL are needed to provide a fast integration path. For example, in the case of the Convey system,

the interface for one external memory access port is separated into one request FIFO and one response FIFO.

Note that from Figure 2.6 we can see that an important step in the flow is manual code refinement. In this thesis we describe many details on how those refinements are done. These also motivate further automations that need to be addressed by the high-level synthesis tools.

2.3 A Heterogeneous Node with CPU, GPU and FPGAs

As part of the CDSC project, we further combine the GPU accelerator and FPGA accelerator into a single server node.

The form-factor of the Convey HC-1 platform is a 2-U rack-mountable server box. The motherboard has two PCI-e X16 slots, but there is no physical space to host a Tesla compute card due to form-factor issues. Currently, we use a PCI-express expansion box, Magma ExpressBox 2, to host a Tesla compute card C1060.

In the HPC space, a distributed memory paradigm is typically used across the cluster nodes, and shared memory is used within each node. However, coprocessors bring in additional complexity, because coprocessors have access to coprocessor-side memory, and the coprocessor-side memory may not be directly addressable from the host. FPGAs also have a large amount of coprocessor-side memory, which can be placed in a shared virtual memory space along with the host-side memory space. This needs some OS support as well as hardware support. Convey Computer realizes a cache-coherent NUMA (ccNUMA) system on top of a FSB protocol stack. With the release of CUDA 4.0 that supports unified virtual addressing, our heterogeneous server node should be able to support a shared virtual memory space across CPU, GPU and FPGAs. Currently, the GPU card in our system still has its own address space.

Our system runs Convey Linux, which is essentially a Linux distribution with modifications to better support the shared virtual memory system. Convey also provides the compiler and debugger toolchains for the system as well. The compiler can recognize certain Convey-

specific pragmas. It also has the ability to generate vectorized code that runs on a vector ISA which can be implemented on FPGAs. NVIDIA CUDA drivers and CUDA toolkit are installed on the system as well.

2.4 Run-Time Support

Despite over four decades of research, few high-level parallel programming models and runtimes are available to domain experts who are not, at the same time, experts in parallelism. Fortunately, this situation is starting to change. Frameworks such as Map-Reduce [30] successfully exploit implicit parallelism on distributed systems and have also been extended to heterogeneous platforms such as GPU [31] and FPGA [32], but unfortunately have a restricted programming model. The StarSs model builds a task-level model using a pragma-based approach (similar to OpenMP) to ease the burden of task-level programming for different architectures; its instantiations include Cell Superscalar [33] for the Cell broadband engine and GPUSs [34] for a system with multiple accelerators. Dryad [35] is a research project at Microsoft Research for a general-purpose runtime data-parallel applications. An application written for Dryad is modeled as a directed acyclic graph (DAG). However, its focus is on the multi-core cluster platform and does not provide sufficient heterogeneity support.

We have installed the CnC-HC and Habanero-C runtime on the platform to provide the runtime and the high-level implementation flow for heterogeneous computing. The Habanero-C (HC) language [36] is a parallel programming model developed at Rice University in the Habanero Research project. Habanero-C has two basic primitives for the task parallel programming model borrowed from X10 [37]: async and finish.

The async statement (async ⟨ stmt ⟩ or async { stmt1 .. stmtN }) causes the current executing thread to fork a new child task that will execute one or more statements. The parent task continues executing the statements that follow the async statement and does not wait for the child task to finish its execution. The finish statement (finish ⟨ stmt ⟩ or finish { stmt1 .. stmtN }) performs a join operation. Consider a thread that executes the statements inside a finish scope. These may spawn one or more children, but no instructions which follow

the finish scope are allowed to start executing until all such children, grandchildren, etc., have finished executing. The language permits any level of nesting of the async and finish scopes, and supports terminally-strict computation and multiple scheduling policies (work-first or help-first) [38], which is more general than Cilk [39] that realizes spawn-sync computations which must be fully-strict and work-first .

For locality, Habanero-C uses Hierarchical Place Trees (HPTs) [40]. HPTs define hierarchical trees of execution location, which are an abstraction for underlying hardware. These places could be cores, groups of cores sharing different levels of cache, or devices such as GPUs or FPGAs. The HC language allows the programmer to spawn a task explicitly at such a place, and the work-stealing runtime is designed to take advantage of this information and preserve locality. The Habanero-C runtime uses a work-stealing scheduler that supports cross-device stealing. An FPGA can steal from a task queue for GPU and vise-versa.

Concurrent Collections (CnC) is a macro dataflow model developed by Intel [5] for execution of C++ programs on homogeneous multicore processors. CnC is a general programming model, while the CnC-HC is a specific instantiation of the model. CnC-HC was developed on top of the Habanero C (HC) programming language, and it uses the async and finish constructs available in HC. However, not all dependency graphs can be expressed using only async and finish, so we need a CnC runtime. CnC-HC took a new and different approach from those in previous instantiations of the CnC model—that of spawning a step only when all data is available in order to eliminate the overhead of repeated task creation when necessary. To better aid the user in specifying the preference of task binding, we added the affinity field in the CnC description. For each task in the CnC dataflow, users can specify a numerical value for each processor/coprocessor (a larger value denotes a greater preference). More details on CnC-HC are in [41].

CHAPTER 3

FPGA/GPU Accelerated Lithography Simulation

3.1 Introduction

Optical lithography is the technology used for printing circuit patterns onto wafers. As the technology scales down, and the feature size is even smaller than the wavelength of the light employed (e.g., 193nm lithography), significant light interference and diffraction may occur during the imaging process. Lithography simulation, which tries to simulate the imaging process or the whole lithography process— from illumination to mask to imaging to resist—is considered an essential technique for the emerging field of DFM.

Lithography simulation can be done through various methods with different accuracy. Model- or rule-based optical proximity correction (OPC) uses empirical rules and models from experimental data to perform the simulation and discover the defects caused by lithography [42]. It is fast but not accurate enough. On the other hand, using finite difference or finite element methods to solve the corresponding electromagnetic equations directly [43] is a very accurate approach, but is so expensive that it can only simulate small regions and designs.

The coherent decomposition method [44] can better balance the accuracy and running time, and is the main method used in computational lithography for large designs. It first decomposes the whole optical imaging system into many coherent systems with decreasing importance. The image corresponding to each coherent system can be obtained via numerical image convolution, and the final image is the weighted sum of the image of each coherent system. However, the method still needs a large amount of CPU time to perform the simulation because the number of layers and the size of images are large. As the technology

scales down and the accuracy requirement goes up, it will be more challenging to meet the tight requirement of design turn-round time.

Leading commercial computational lithography products have already started to use special coprocessor acceleration to further accelerate computation. Brion Technologies (now part of ASML) reports that each leaf node composed of two CPUs and four FPGAs in their Tachyon System can achieve 20X speedup over one single CPU node [45]. Mentor Graphics uses the Cell Broadband Engine to accelerate the computation in their nmOPC product [46]. Clear Shape Technologies has filed patents for using GPUs to accelerate the computation [47].

We present a new hardware implementation for accelerating lithography imaging simulation on FPGA platforms. Unlike the image-based approach Brion takes, which ultimately relies on the accelerated performance of 2D-FFT, we use the polygon-based approach [48, 49] instead. The polygon-based approach makes use of the fact that the actual layouts are solely composed of rectilinear shapes, and it has comparable or even better performance than an image-based approach in software implementation. Recent advances in OPC algorithms, e.g., IB-OPC [50], also employ a polygon-based approach for lithography intensity simulation. Moreover, the polygon-based approach pre-computes the convolution and stores that into a look-up table, and the subsequent computation mainly just involves some additions and subtractions on the look-up value. Thus the polygon-based approach could be better approximated via fixed-point computations without sacrificing much accuracy. The algorithm can be better parallelized and accelerated by utilizing the high bandwidth of on-chip memory in FPGA.

Another unique aspect of our work is that we leverage state-of-art C to HDL compilation tools and write all our design in C, whereas the FPGA accelerator design by Brion [45] was based on completely manual RTL coding. A design described in higher-level languages such as C/C++ is more portable to various platforms and easier to maintain. Also these tools could evaluate multiple design choices faster and perform various kinds of optimizations for improved performance against the RTL-based design, but those tools also have limitations on the supported language features. The challenge we experienced for this design is that of

manually developing an efficient memory partitioning scheme, based on the observation of the memory access pattern, to provide a large data bandwidth for a larger number of processing elements. Deep loop pipelining and the overlapping of the communication and computation via ping-pong buffers are also implemented to take advantage of both instruction-level parallelism and task-level parallelism. All the design techniques are represented at algorithmic level in the code refinement/rewriting of ANSI C, and the resulting C code is further synthesized into RTL though the automatic C-to-RTL synthesis tools.

3.2 Basics for Aerial Image Simulation

3.2.1 The Imaging Equation

The coherent decomposition method first decomposes the whole optical system, into a series of coherent optical systems (using eigenvalue decomposition). The series is truncated to a finite one based on the ranking of the eigenvalues. If we only keep K significant eigenvalues and eigenvectors, the image can be computed as:

$$I(x,y) \approxeq \sum_{k=1}^{K} \lambda_k |(O \otimes \phi_k)(x,y)|^2 \tag{3.1}$$

Here the $I(x,y)$ is the image intensity, λ_k is the k^{th} eigenvalue, $O(x,y)$ is the object function (field) and $\phi_k(x,y)$ is the k^{th} eigenvector. The symbol $\otimes$ denotes convolution (2D image convolution). For more details on the derivation of the coherent decomposition method, please refer to [51].

One way to perform the 2D image convolution is through 2D FFT. We can first transform the padded object pattern (see Section 3.2.3) and the eigenvector into the frequency domain via 2D FFT. (Note the FFT of the eigenvector ϕ_k only needs to be computed once and can be reused.) Then we multiply the eigenvector and object pattern in the frequency domain. Finally we can obtain the convolution result via an inverse FFT of the multiplied result. This is the image-based simulation that is also used by [45].

As the actual layout of VLSI circuits is only composed of polygons (or rectangles if we perform polygon decomposition on the layout), the convolution of different sizes of polygon-

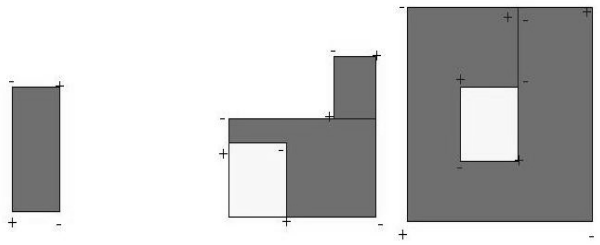


Figure 3.1: Rectilinear polygons can be processed similar to rectangles

s/rectangles can be pre-computed and stored. We first consider purely rectangle cases. The convolution for an object pattern solely composed of N rectangles with vertices at $(x_1^{(n)}, y_1^{(n)})$, $(x_2^{(n)}, y_1^{(n)})$, $(x_1^{(n)}, y_2^{(n)})$,$(x_2^{(n)}, y_2^{(n)})$ can be simplified via *quadrant functions*.

The object pattern in one padded area can be written as:

$$O(x,y) = \sum_{n=1}^{N}[Q(x - x_1^{(n)}, y - y_1^{(n)}) - Q(x - x_2^{(n)}, y - y_1^{(n)}) + Q(x - x_2^{(n)}, y - y_2^{(n)}) - Q(x - x_1^{(n)}, y - y_2^{(n)})] \tag{3.2}$$

where *quadrant function*

$$Q(x,y) = \begin{cases} 1 & if\ x \geq 0\ and\ y \geq 0 \\ 0 & otherwise \end{cases}$$

The convolution equation thus can be rewritten as:

$$\begin{aligned} & I(x,y) \approxeq \sum_{k=1}^{K} \lambda_k |(O \otimes \phi_k)(x,y)|^2 \\ = & \sum_{k=1}^{K} \lambda_k | \sum_{n=1}^{N} [\psi_k(x - x_1^{(n)}, y - y_1^{(n)}) - \psi_k(x - x_2^{(n)}, y - y_1^{(n)}) \\ + & \psi_k(x - x_2^{(n)}, y - y_2^{(n)}) - \psi_k(x - x_1^{(n)}, y - y_2^{(n)})]|^2 \end{aligned} \tag{3.3}$$

where

$$\psi_k(x,y) = Q(x,y) \otimes \phi_k(x,y)$$

is the convolution of the quadrant function with the k^{th} eigenvector. This is the polygon/rectangle-based algorithm we use, and it is described in more detail in [49].

For rectilinear polygons, an equation similar to Equation 3.2 can be written using quadrant functions on each vertex of the polygons. In Figure 3.1, we label the + and − on the

vertexes of rectilinear polygons (the rectangle is simply a special type of rectilinear polygon, and + and − are just the signs for the look-up value for the vertexes; see Equation 3.2 and Equation 3.3 for the rectangle case). We go from a vertex which is at bottom-left, and label that with + and go around the border of the polygon and label that with − and + respectively.

Using rectangles or polygons will not alter the overall algorithm and design presented in the following. For simplicity and benchmarking purposes, we assume we are given N rectangles with 4N vertexes for one region of the image in the subsequent illustration, while our litho simulation tool, which goes from GDSII to simulated image, is also capable of processing polygons from the GDSII directly without the need for rectangle decomposition.

3.2.2 Image-Based Simulation versus Polygon-Based Simulation

It is worthwhile to briefly discuss the trade-offs between image-based simulation and polygon-based simulation. Image-based simulation first converts the layout, which in most cases is stored in GDSII, into an object image, and then gets the image convolution via FFT and IFFT. Note that only image-based simulation needs to take this additional conversion step from the GDSII to the object image, which might also be expensive. The 2D-FFT algorithm, although well studied, still needs a large number of floating-point operations. On the other hand, the polygon-based algorithm can use fixed-point computation without losing much accuracy, and thus can be implemented without using any floating-point operations. Suppose the rounding error for the elements in ψ_k in Equation 3.3 is 2^{-p}, the absolute error for computing $\sum_{n=1}^{N}[\psi_k(x-x_1^{(n)}, y-y_1^{(n)}) - \psi_k(x-x_2^{(n)}, y-y_1^{(n)}) + \psi_k(x-x_2^{(n)}, y-y_2^{(n)}) - \psi_k(x-x_1^{(n)}, y-y_2^{(n)})]$ is bounded by $4N * 2^{-p}$. If we assume the error distribution for the elements in ψ_k is a uniform distribution between -2^{-p} and 2^{-p}, the error distribution of the sum $\sum_{n=1}^{N}[\psi_k(x-x_1^{(n)}, y-y_1^{(n)}) - \psi_k(x-x_2^{(n)}, y-y_1^{(n)}) + \psi_k(x-x_2^{(n)}, y-y_2^{(n)}) - \psi_k(x-x_1^{(n)}, y-y_2^{(n)})]$ follows uniform sum distribution. Its variance is proportional to N and the standard deviation is proportional to $\sqrt{N}$; thus the actual error is much smaller, statistically, than the conservative error bound which is linear with N.

Both algorithms scale linearly with the number of pixels to compute. The issue of the polygon-based approach is that the running time will also depend on the layout density, which determines the number of polygons or rectangles in a unit area within the interaction range (N in Equation 3.3), while the image-based approach only depends on the chip area. We implemented the 2D-FFT based 2D-convolution using the FFTW package [52], and tested that on kernels with size 400 by 400. We found that the running time is comparable to a polygon-based method with a moderate density (see Section 3.5.3). The polygon-based approach requires less computation and runs faster for layers that are not very dense, and the image-based approach runs faster for very dense layers. Note that the polygon-based approach also saves the step on conversion from polygons to images, as polygons are naturally stored in GDSII.

In terms of the FPGA-based acceleration, FFT is still tightly constrained by the available DSP units or logic slices, and the peak FLOPs of FPGA are at the same magnitude with the peak FLOPs of modern CPU; thus, typically only a 2 to 8X speedup is seen on accelerating FFT on FPGA platforms via parallel implementation [53]. Fixed-point FFT core for FPGA is also available and gives potentially larger speedup, but it will have worse accuracy because multiplication can enlarge the absolute error. For the polygon-based approach, the convolution on the quadrant function can be pre-computed using highly accurate floating-point computations (on CPU) and reused multiple times. The time to pre-compute the convolution can be ignored as it is a one-step process. The remaining computations only involve table look-up and simple addition/subtraction operations, and are much more suitable for a decent speedup. Therefore, in this work we use the polygon/rectangle-based approach rather than the image-based approach for accelerator design.

3.2.3 Detailed Settings for the Imaging Equation Using the Polygon-Based Approach

We assume that the convolutions of eigenvectors and the quadrant functions are already pre-computed, and sampled into a 2D array called *kernel*. The region/range of the kernel we use

is 2000nm by 2000nm; it is sampled on a 5nm grid, and thus contains 400 by 400 numbers. The image we need to compute is on a 25nm grid. Without loss of generality, we assume the layout corners (vertexes of the polygons) are also on the 5nm grid. (If the layout corner is on a much finer grid, interpolation will be used to get the kernel value.) These settings used in our algorithm and implementation were recommended by our industry collaborators from Magma Design Automation [54], but our architecture certainly is not confined to these settings and can be extended to other settings. Some setting changes require a recompile/re-synthesis while some do not. This depends on whether the change of underlying hardware is needed. For example, the design synthesized with a large N (layout density metric) can be used for a smaller N without the need of changing hardware. On the other hand, enlarging array sizes or changing the memory partitioning schemes will affect the underlying hardware and a recompile/re-synthesis will be needed to generate the new hardware bitstreams.

3.3 FPGA-based Accelerator for the Imaging Simulation

3.3.1 Image Padding for the Polygon-based Approach

Both the object pattern and the simulated image are large; however, when we compute one region of the image, only one padded region of the object needs to be considered due to the locality of the litho effects. For example, for a kernel ranges within a 2000nm by 2000nm area, if we want to compute a 1000nm by 1000nm image region, an object pattern within a range of 3000nm by 3000nm needs to be taken into computation. The reason for this is that some objects are far away from the current pixel and out of the interaction range, and therefore need not be considered.

The computation complexity is proportional to the number of rectangles N taken into computation, and the intensity of each pixel is determined by the rectangles within the interaction range (2000nm by 2000nm in our case) around this pixel.

```
for (x=0;x<pixel_max;x++)
 for (y=0;y<pixel_max;y++)
 {
//Initialize pixel intensity
  I[x][y]=0;
  for(k=0;k<K;k++)
  {
   //Initialize partial sum
   I_k[x][y]=0;
   //Core computation
   for(n=0;n<4*N;n++)
    {
     addr_x=5*x-rect_x[n]+c;
     addr_y=5*y-rect_y[n]+c;
     I_k[x][y]+=(-1)^n*kernel[k][addr_x][addr_y];
    }
    I[x][y]+=I_k[x][y]*I_k[x][y];
   }
  }
```

Figure 3.2: Pseudo-code for the nested loop

```
//Initialize pixel intensity
for(x=0;x<pixel_max;x++)
 for(y=0;y<pixel_max;y++)
  I[x][y]=0;
for(k=0;k<K;k++)
 { //Initialize partial sum
   for(x=0;x<pixel_max;x++)
    for(y=0;y<pixel_max;y++)
     I_k[x][y]=0;
    //The core computation
   for(n=0;n<4N;n++)
    for(x=0;x<pixel_max;x++)
     for(y=0;y<pixel_max;y++)
     {
      addr_x=5*x-rect_x[n]+c;
      addr_y=5*y-rect_y[n]+c;
      I_k[x][y]+=(-1)^n*kernel[k][addr_x][addr_y];
     }
    //Square operation
   for(x=0;x<pixel_max;x++)
    for(y=0;y<pixel_max;y++)
     I[x][y]+=I_k[x][y]*I_k[x][y];
 }
```

Figure 3.3: Pseudo-code for the rearranged nested loop

3.3.2 Rearranging the Nested Loop

Now we devote ourselves to implementing the nested loop corresponding to Equation 3.3, which is described in Figure 3.2, where c is a constant for addressing alignment, and $rect_x$ and $rect_y$ are arrays for the coordinates of rectangle corners. Note that this is the code for simulating one region of an image, and there is another outer loop over the pseudo code in Figure 3.2 for changing the current image region where the input N and $rect_x$ and $rect_y$ shall all be changed as we move to different image regions. Here the $pixel\_max$ is the number of pixels in either X or Y direction. We use $5 * x$ and $5 * y$ in the code as we use an image grid on 25nm and a kernel grid on 5nm. The outermost loop goes over different image pixels. Within the outer loop there is a loop that goes over different kernels. The innermost loop goes over different layout corners. This is a direct implementation of Equation 3.3, but it might not be suitable for generating synthesizable hardware. We apply loop interchange techniques to find a better rearrangement for the nested loop.

Typically, loop transforms can be applied to increase the parallelism or improve the data locality. Suppose two loop transforms have the same number of off-chip access count, we would prefer the implementation with a smaller size of on-chip reuse-buffer. In this case, we look at the choices for the outermost loop. Because the whole nested loop will need the data in the kernel array, which shall be reused for the image computation with different image regions or different pixels and layout corners, we would like to pre-fetch the kernel array and store the kernel array in the on-chip RAM of the FPGA. In our setting in Section 3.2.3, each kernel has $16 * 400 * 400$ bits of data, which is 2.44Mb, if we use 16bit precision for kernel data. As the total size of on-chip RAM of FPGA is limited, it is unlikely that all the kernels will fit, but in our case at least one kernel can be put in, (the device we use has around 9Mb on-chip memory in total). Thus, we would like to make the loop over different kernels the outermost loop. Otherwise, we may need an on-chip storage that is KX larger (or simply do not use any reuse).

Besides the consideration for data reuse, we want the data accesses (especially the innermost levels) are regular or affine. Affine data access can potentially enable the memory

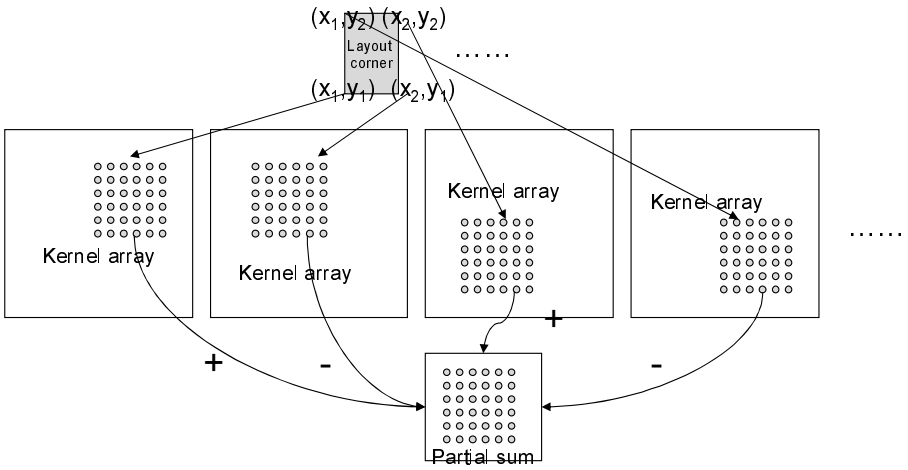


Figure 3.4: Computation of the inner nested loop

partition scheme which we will elaborate on in later sections. We can see that the loop over different layout corners is less *structured* than the loop over the image pixels. If we fix one layout corner and update the set of pixels, the memory access pattern is parametrical affine. (The parameter is the coordinates of the layout corners, which are loop-invariants for the loops over image pixels.) This property enables our subsequent memory partitioning scheme. But if we fix one pixel and change different layout corners, it will not be somewhat random (not regular) because we can not expect the layout corners to have some specific pattern. Thus, we would like to make the loop over different pixels the innermost loop. Figure 3.3 is the nested loop after the loop interchange.

The illustration of the computation is shown in Figure 3.4. The address of kernel data depends on the value of the rectangle corner. For one specific corner, the address for the kernel array is just an affine mapping over the pixel index x and y. As the data of $rect_x[n], rect_y[n]$ is in the layout corner array and accessed at runtime, the data access for the kernel array is still some type of indirect memory access. This imposes great challenges for the automation tools. After the rearranging of the nested loop, the key problem is to design and implement the inner loop—the inner loop over different rectangle corners and image pixels.

Later, automatic loop transform for data reuse is presented in [55]. The work still needs further extensions to handle our case because the data accesses in our nested loop are not strictly affine.

3.3.3 Communication Analysis and HW/SW Partitioning

In our design, the hardware component running on FPGA mainly initializes and computes the image partial sum I_k, while the result is sent back to a software component running on the processor; the software component parses and provides input data to the hardware component and also performs the square operation and stores the results.

As the computation is mainly performed on the FPGA coprocessor, input data required for computation needs to be transferred from the host CPU to the coprocessor, and the computed results need to be transferred back. Although the Hyper-Transport bus enables a low latency solution, overhead in the data transfer still exists.

The major part of data transfer from host (CPU) to coprocessor is the layout corner array ($rect_x$,$rect_y$ in the pseudo code). Note that kernel data also needs to be transferred from the host to coprocessor, but the same kernel can be used for a larger number of padded image region, thus the communication overhead in transferring kernel data can be neglected. The major data transfer from coprocessor to CPU is the array of image partial sum I_k. Assume we use 16-bit data for the elements in $rect_x$,$rect_y$ array and 32-bit data in array I_k. For settings shown in Section 3.2.3, the total bytes of data transfer for computing one padded region with size 1000nm by 1000nm on a 25nm grid is $(32/8) * 4N + (32/8) * 40 * 40$, where the first term corresponds to the data transfer for the layout corner array and the second term corresponds to the transfer of partial image. The data transfer is done in a DMA-like fashion. For a moderate density say $N = 100$, and a data bandwidth around 800MB/s (the peak bandwidth of the SRAM device we use as hardware/software shared memory), the data transfer needs around $10\mu s$. This is roughly 10% of the overall execution time of our accelerated design. In Section 3.3.8 we talk about the overlapping of the communication and computation, a technique that could completely resolve the overhead.

3.3.4 Exploring Parallelism

Typically, the accelerator on the FPGA platform is able to explore task parallelism, data parallelism, and instruction parallelism. We will use high-level synthesis tools, e.g.,

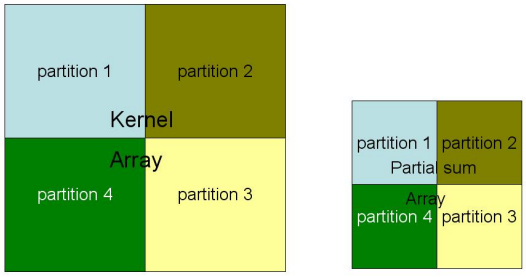


Figure 3.5: Naive partitioning based on geometric locations

AutoPilot™[25] to implement our design. AutoPilot first parses the input description, e.g., in C, and generates a control data flow graph (CDFG) of the code. Then it performs scheduling and binding on the CDFG to generate the final RTL. Instruction parallelism is directly realized by the scheduler of the tool, because scheduler might schedule multiple instances of function units (FU) at same cycle. Moreover, the loop pipelining pragma can inform the scheduler to schedule the loop in a pipelined fashion, creating more instruction parallelism. Task parallelism needs to explicitly write multiple processes or tasks, and needs to consider inter-process synchronization and arbitration of shared resources. Currently, AutoPilot can realize the task parallelism via functional block pipelining or through SystemC based description. Data parallelism, on the other hand, widely used in SIMD instructions or GPU accelerators, tries to use the same or similar program/code to cope with multiple data. AutoPilot uses loop unrolling pragma to invoke program transform for loops, and the scheduler will schedule the unrolled CDFG to create the data parallelism.

In our initial scheme, we developed a design based on task-level parallelism. We first partitioned the kernel array and the partial image array into several partitions based on the geometric locations. Figure 3.5 shows a 4-way naive partitioning.

We then allocate four PEs in the FPGA, and each PE is responsible for the computing of one partition of partial image array. As the computing of one partition of partial image might need all the four partitions of kernel data, access conflicts might occur. We schedule the operations such that different PEs will read or write different memory blocks at each computation stage (shown in Figure 3.6).

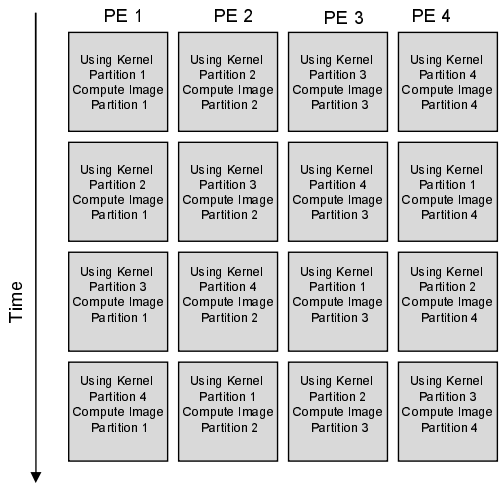


Figure 3.6: Block scheduling for naive partitioning

```
.....
  //Core computation, 4-way unrolling
  for(n=0;n<4N;n++)
    for(x=0;x<pixel_max/2;x++)
      for(y=0;y<pixel_max/2;y++)
      {
        addr_x=5*2*x-rect_x[n]+c;
        addr_y=5*2*y-rect_y[n]+c;
        I_k[2*x][2*y]
          +=(-1)^n*kernel[k][addr_x][addr_y];
        I_k[2*x+1][2*y]
          +=(-1)^n*kernel[k][addr_x+5][addr_y]
        I_k[2*x][2*y+1]
          +=(-1)^n*kernel[k][addr_x][addr_y+5]
        I_k[2*x+1][2*y+1]
          +=(-1)^n*kernel[k][addr_x+5][addr_y+5];
      }
.....
```

Figure 3.7: Pseudo-code for the partially unrolled nested loop

However, as the address of the required kernel data depends on the set of layout corners, it is likely that this approach might face load-balancing problems, if the layout corner is not uniformly distributed. For example, if the whole loop makes heavy use of one or several specific partitions, the benefit or speedup using partitioning might become degraded. Another drawback is that this type of task parallelism needs control flow in each PE and needs additional logic to do synchronization.

Later, we decided to mainly borrow the idea from data parallelism, where we first unroll the inner nested loop (the loop over different pixels) to some degree and try to execute the multiple operations in the inner loop at exactly the same cycle. The benefit is that the control flow is more simplified and the load-balancing problem no longer occurs. The 4-way unrolled code is shown in Figure 3.7, where we unroll once in x direction and once in y direction. (Note this unrolling doesn't need to be written explicitly, as in Figure 3.7, but can also be achieved by specifying unrolling pragmas in the original loop.) However, this rewriting technique does not help without further memory partitioning, as each on-chip memory block only has limited ports. When loop pipelining is further enabled, the unrolling might increase the initiation interval of pipelining and not contribute much for the overall latency. The goal of the memory partitioning is to make sure the correspondent simultaneous memory accesses in the unrolled loop are partitioned into different memory blocks.

Besides the parallelism similar to data parallelism, we also use a loop pipelining technique as an instructional parallelism technique, and the data prefetching or the overlapping of the SW/HW communication and computation using function block pipelining or task-level parallelism.

3.3.5 Memory Partitioning using Modulo Addressing

This subsection discusses the memory partitioning scheme to allow multiple memory access in the inner unrolled loop to be parallelized. For each block of on-chip memory in the FPGA, typically there are only two ports available. If the kernel array and partial image (a temporary buffer for each pixel to store the inner sum) are just single memory blocks without

partitioning, multiple memory accesses can not be scheduled to the same cycle due to port contention. Thus, clearly we need to partition the memory to allow for parallel processing and to achieve high bandwidth and high throughput.

The idea we used in memory partitioning is a variant of modulo addressing. The modulo addressing approach with a circular buffer which could get a row of data containing n elements from n banks is presented in [56]. The basic idea of modulo addressing is simple: if we want to fetch a row of data with address $x, x+1, x+2, \cdots, x+n-1$ simultaneously, we can use cyclic array partitioning where data with different modulo of address $x\%n$ are placed in different memory blocks. Cubic addressing [57] provides a partitioning scheme for 3-D array which can obtain a 2*2*2 voxel neighborhood simultaneously from eight banks, this is a special case for modulo addressing. Our case is a 2D case rather than 1D. Also, we want data with address $5x, 5(x+1), \cdots, 5(x+n-1)$ in different blocks; thus we need some natural extensions on the simple modulo addressing. Also the aforementioned work is in the domain of multi-port memory architecture design for medical image processing or video coding, while we describe the partitioning scheme for our design in behavior C for reconfigurable computing.

In our case, we mainly have three arrays: the kernel array, which serves as the look-up table for the computation; the partial image sum array, which is used to store the intermediate inner sum for different pixels; and the layout corners array. Without loss of generality, we assume that the overall size of the three arrays can be loaded into the on-chip RAM of the FPGA. This assumption holds for our test settings, but generally might not be true, and further partitioning of data and computations might be needed.

We would like to further partition the array to take advantage of the high peak bandwidth of the on-chip RAM. Multiple processing elements(PEs), can process multiple data concurrently if we partition the kernel array and the partial sum array effectively without access contention and serialization.

To obtain a better partitioning scheme for this specific nested loop, we need to take a look at the memory access pattern. For a specific layout corner, we need to update all the

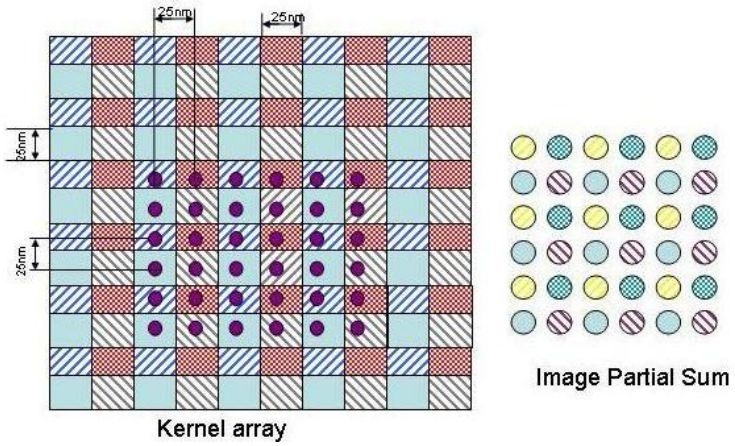


Figure 3.8: 4-way (2 by 2) memory partition scheme for load balancing

image pixels. The corresponding data access in the kernel array has a regular pattern (shown in Figure 3.4). A better partition scheme should be able to evenly distribute the memory access pattern shown in Figure 3.4 into multiple memory banks/blocks, regardless of the location value of specific layout corners. Therefore, we choose to use a modulo interleaving partition scheme.

We still take 4-way partitioning as an example, and the illustration is shown in Figure 3.8. *The same color/texture means the data are physically stored in the same memory bank/block.* Using this partition scheme, multiple data can be fetched concurrently without any conflicts.

The basic idea for partitioning is to follow the grid size of the access pattern in both X and Y directions so that memory accesses in the most inner unrolled loop in Figure 3.7 are always in different memory partitions. In our case, the kernel array is on a 5nm grid and the image array on a 25nm grid. The illustration of the partitioning is shown in Figure 3.8. In this figure, dots are memory accesses for kernel arrays (shown on the left) and the memory access for the image partial sum (shown on the right). We can see the concurrent accesses always lie *in different colors* in Figure 3.8, and thus can be scheduled to execute at the same cycle. Figure 3.8 could be obtained via first *coloring* (here we use *coloring* to represent the memory partitioning) the bottom-left 5 * 5 corner blocks in the kernel array, and *coloring* other blocks in an interleaved fashion, to ensure that the four memory accesses in the inner unrolled loop are in different memory blocks. Note that Figure 3.8 only shows a 4-way 2 *

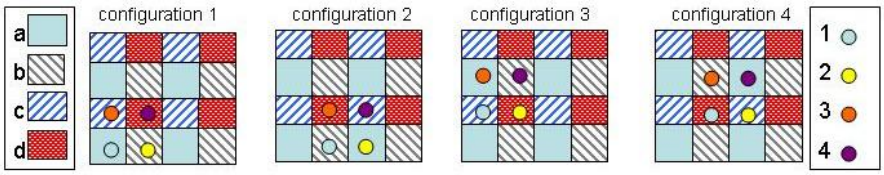


Figure 3.9: Address generation and output data multiplexing

2 partitioning corresponding to the partially unrolled loop in Figure 3.7, but a 5 * 5 or 8 * 8 partitioning scheme can also be developed similarly to allow for a larger data bandwidth.

The array of image partial sum is also partitioned in a fashion shown in Figure 3.8, so that we can write the output data into the array concurrently. As the addresses for the image partial sum are affine mapping of loop variables and do not depend on run-time data, the partitioning for the image partial sum array is somewhat simpler. The layout corner array does not need partitioning as the loop over layout corners is an outer loop.

3.3.6 Address Generation Logic for Partitioned Memory

As we explore the memory access pattern to partition the memory to allow for concurrent access, the addresses to fetch those data also need to transformed and mapped.

The address generation logic, and the memory partitioning and the multiplexing of output data, are all implemented via rewriting the code of the original algorithm written in ANSI C.

Again take the 4-way partitioning in Figure 3.7 and Figure 3.8 as an example. In Figure 3.9, a, b, c, d are four memory blocks after partitioning, and $1, 2, 3, 4$ are four concurrent memory accesses. There are four different configurations shown in the figure. With different address shifting determined by $rect_x$ and $rect_y$, the concurrent memory accesses have different combinations with the memory blocks they visit. In configuration 1, the four concurrent memory accesses $1, 2, 3, 4$ will need the data in memory block a, b, c, d, respectively. In configuration 2, the four accesses will need the data in memory block b, a, d, c, respectively. Different configurations will have slightly different address generation logics.

The address generation logic first looks at which configuration is among the four cases in the 2 by 2 partitioning design in Figure 3.9. Later, for each configuration, there is a mapping function to transform the original address into the mapped address. For each of the four configurations shown in Figure 3.9, the data we get from different memory partitions also needs to go through multiplexing to provide the required data for the accumulator. Let us go into more detail on the address generation. Again we take 2 by 2 partitioning cases as an example. Suppose we denote that the four addresses that the unrolled loop needs to access (before the address mapping) in Figure 3.7 are $[addr_x][addr_y]$, $[addr_x + 5][addr_y]$, $[addr_x][addr_y + 5]$ and $[addr_x + 5][addr_y + 5]$ respectively. We first determine which group the address lies in by looking at the quotient $addr_x/(2 * 5)$, $addr_y/(2 * 5)$; e.g., in Figure 3.9 $addr_x/(2*5) = 0$ and $addr_y/(2*5) = 0$. We can determine which configuration it is by looking at the modulo $addr_x\%(2*5)$ and $addr_y\%(2*5)$; e.g., $addr_x\%(2*5) < 5$ *and* $addr_y\%(2*5) < 5$ means that it is the first configuration. The divisor here is $2 * 5$: 2 relates to 2 by 2 partitioning, 5 relates to the image grid size which is 5X larger than kernel grid size (also in the pseudo code in Figure 3.2).

We denote the mapped address $[addr_x^a][addr_y^a]$, $[addr_x^b][addr_y^b]$, $[addr_x^c][addr_y^c]$, $[addr_x^d][addr_y^d]$ for the four different memory blocks shown in Figure 3.9. And a mapping function is

$$addrmapped_x = f(addr_x) = (addr_x/(2 * 5)) * 5 + addr_x\%5$$
$$addrmapped_y = f(addr_y) = (addr_y/(2 * 5)) * 5 + addr_y\%5$$

The first term determines the address that corresponds to different groups of concurrent access, and the second term determines the address shifting within a 5*5 block.

If it is the first configuration, then the addresses for the four memory block are the same.

$$addr_x^a = addr_x^b = addr_x^c = addr_x^d = addrmapped_x$$
$$addr_y^a = addr_y^b = addr_y^c = addr_y^d = addrmapped_y$$

If it is the second configuration, we have

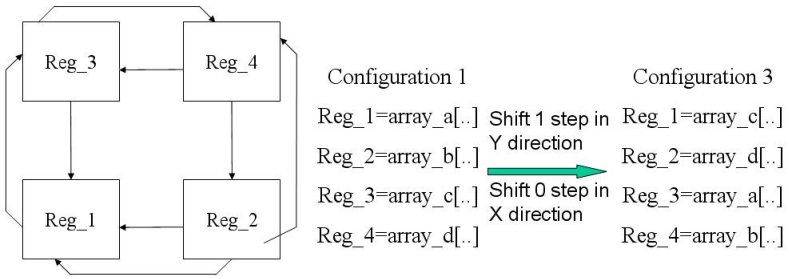


Figure 3.10: 2D ring for 2 by 2 partitioning

$$addr_x^b = addr_x^d = addrmapped_x$$

$$addr_x^a = addr_x^c = addrmapped_x + 5$$

$$addr_y^a = addr_y^b = addr_y^c = addr_y^d = addrmapped_y$$

We can get the address for the remaining two configurations similarly. After we get the addresses for each configuration, we can somewhat simplify the logic by extracting the common terms, and write the address generation logic as follows:

$$\begin{aligned} addr_x^a &= addrmapped_x \;\; if(addr_x\%(2*5) < 5) \\ &\quad\; addrmapped_x + 5 \;\; otherwise \\ addr_y^a &= addrmapped_y \;\; if(addr_y\%(2*5) < 5) \\ &\quad\; addrmapped_y + 5 \;\; otherwise \end{aligned}$$

When we use other partitioning, such as 5 by 5 partitioning, similar address generation logic can also be written using the same idea and format.

Note that the equations shown above use division and modulo operations. It is well known that these operations are relatively costly on FPGA in terms of both area and latency. However, we pre-compute these costly operations at the CPU side and store them so that the FPGA does not need to worry about this. If we recall the address generation shown in Figure 3.3, we only need to convert the array data in $rect_x$ and $rect_y$ to the form of $quotient * divisor + remainder$ so that the quotient and remainder used in address computation can be obtained directly.

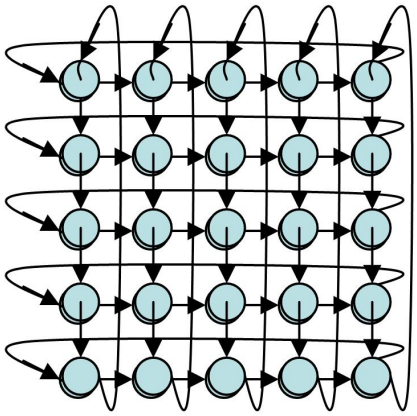


Figure 3.11: Ring based data multiplexing for 5 by 5 partitioning

3.3.7 Output Data Multiplexing

The data we fetched from the partitioned memory blocks needs to go through multiplexing before it is sent to the accumulator, because the computation of one partition of image partial sum might still require data in different partitions of kernel array. For the 2 by 2 partitioning shown in Figure 3.9, if the data from the four memory blocks are $array_a[..]$, $array_b[..]$, $array_c[..]$, $array_d[..]$ and the data we can directly send to the accumulator are $Reg_1, Reg_2, Reg_3, Reg_4$. We denote the multiplexing logic as

$$
\begin{aligned}
& Reg_1, Reg_2, Reg_3, Reg_4 \\
= \ & (array_a[..], array_b[..], array_c[..], array_d[..]) \ \ (configuration\ 1) \\
= \ & (array_b[..], array_a[..], array_d[..], array_c[..]) \ \ (configuration\ 2) \\
= \ & (array_c[..], array_d[..], array_a[..], array_b[..]) \ \ (configuration\ 3) \\
= \ & (array_d[..], array_c[..], array_b[..], array_a[..]) \ \ (configuration\ 4)
\end{aligned}
$$

But this naive multiplexing might have a large routing overhead when we have a larger partitioning. We use 2D ring-based shifting to implement the multiplexing. Figure 3.10 is the interconnect structure for the 2 by 2 partitioning using ring-based multiplexing. For configuration 1, no shifting is needed; for configuration 2, we can shift one step in X direction; for configuration 3, we can shift one step in Y direction; and configuration 4 requires shifting one step in both X and Y directions.

```
//shifting/multiplexing in X direction
 for(m=0; m<5-1; m++)
 {      if(sel_x>m){
              ring_shift_x;
              }
 }
 //shifting/multiplexing in Y direction
 for(m=0; m<5-1; m++)
 {     if(sel_y>m){
              ring_shift_y;
              }
 }
```

Figure 3.12: Pseudo code for 2D ring multiplexing for 5 by 5 partitioning

A larger partitioning scheme, e.g., 5 * 5 partitioning, can also use a similar multiplexing scheme. Figure 3.11 is the interconnect structure for the 5 * 5 partitioning using ring-based multiplexing. The pseudo code for the 2D ring-based multiplexing for 5 * 5 partitioning is shown in Figure 3.12. sel_x and sel_y are values to determine how many steps the whole ring needs to be shifted, which can be obtained by modulo of the layout corner; *ring_shift_x* means all data in the 2D-ring is assigned the value on its circular left side; *ring_shift_y* means all data in the 2D-ring is assigned the value on its circular upper side. Although the *if* statement inside the loop can be merged to the loop bound, we write it in the current form so that it has a constant loop bound. These loops are further unrolled/flattened to facilitate the loop pipelining of the outer loop. We wrote the pseudo code in such a way that the cycles needed to perform shifting are constant regardless of which configuration it is, otherwise a non-deterministic cycle count might bring difficulties for pipelining. The whole shifting is done in a multi-cycle fashion. In our 5 * 5 partitioning-based design, we will shift two steps in one clock cycle. Although the latency of the multiplexing through this 2D ring structure is long, it will not affect the overall performance due to loop pipelining, as the

```
for(x=0;x<pixel_max;x++)
    for(y=0;y<pixel_max;y++)
        {// loop pipelining pragma
            ......
        }
```

(a) without flattening

```
x=0;y=0;
for(idx=0;idx<pixel_max*pixel_max;idx++)
        {// loop pipelining pragma
            ......
            y++; if(y==pixel_max) {y=0;x++;}
        }
```

(b) with flattening

Figure 3.13: Loop flattening for deep loop pipelining

multiplexing block can be implemented as a unit that has a multi-cycle latency but with a one-cycle pipeline initiation interval (similar to many floating point IPs).

3.3.8 Loop Pipelining and Function Block Pipelining

The whole nested loop is pipelined to increase the throughput. Although many rewritings, including the address generation and data multiplexing, complicate the logic and increase the latency, they will not affect the performance much because the whole nested loop (over

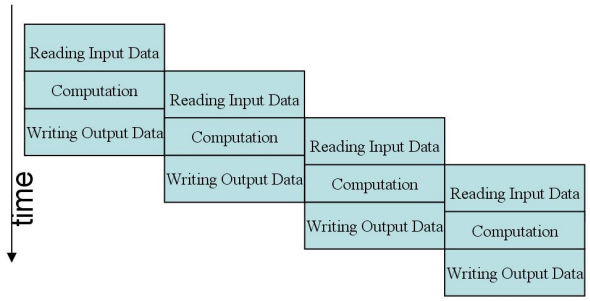


Figure 3.14: Block pipelining/overlaping communication and computation

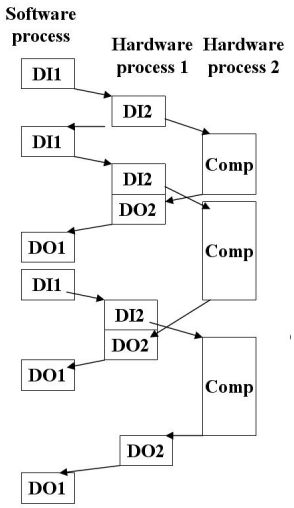


Figure 3.15: Explicit control flow on overlapping communication and computation

different rectangles and image pixels) can be pipelined and can achieve an initiation interval equal to one. Loop pipelining can be achieved by specifying loop pipelining pragmas within the nested loop. The core loop we implement is a nested loop. Specifying loop pipelining pragmas might only pipeline the inner loop. The pipelining possibility also lies between different instances of the middle loop and even the outermost loop. We manually flatten the loop to reduce the depth of the loop-nest. In this way, we can further reduce latency by reducing the startup latency of the pipeline. The flattening process is shown in Figure 3.13. This is a tool-specific rewriting. Some tools might be able to pipeline the whole nested loop without manual flattening.

Moreover, we would like to overlap all the communications and computations so that the hardware component is running the computations almost all the time. This can either be viewed as function block pipelining (Figure 3.14) or realized as an explicit control flow of multiple tasks (Figure 3.15). In Figure 3.15 $DI1$ means transferring data from the CPU side to the SW/HW shared SRAM, and $DI2$ means transferring data from the SRAM to the FPGA. $DO2$ means transferring data from the FPGA to the SRAM, and $DO1$ means transferring data from the SRAM to the CPU. *Comp* is the computation part in FPGA. This is an explicit control flow, and two hardware processes communicate with each other

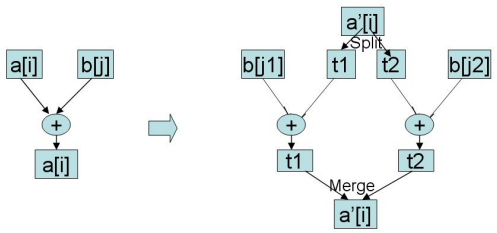


Figure 3.16: Using wider data to balance the use of the memory ports

via *signals*. Ping-pong buffers are used for both the layout corner array and the image partial sum array, which serve as input data array and output data array for the computation of one region respectively. 2X storage space is used for the ping-pong buffer while one is used in the current computations and the other is used for sending/receiving data. Using ping-pong buffers can ensure that the overlapping will not alter the data that is needed for current computation. In Section 3.3.3, we found that the communication time is not greater than the computation time, thus the overlapping can hide the overhead in the communications.

3.3.9 Using Wider Memory Access to Balance the Usage of Memory Ports

Besides the techniques shown in the previous subsections, we observe that the port accesses for the *kernel* memory and the image partial sum memory are not equal. Even in the pipelined loop, at each cycle we only get one unit of data from a partition of *kernel* memory, but we use two ports for the image partial sum memory due to the accumulator. As each memory block can have two ports, we try to use both of the two ports of the *kernel* memory. But at the output side, the image partial sum memory needs to store a wider bit of data to avoid port conflict. This would provide 2X more parallelism without further partitioning of the memory blocks. Figure 3.16 illustrates the computation elements using wider data.

3.4 Leveraging C to HDL Compiler for Hardware Generation

The entire algorithm is written in C so that we can leverage up-to-date C to HDL translation tools. We use AutoPilot [25], which is a commercial tool that can take ANSI C (within the

synthesizable subset) as input and generate synthesizable and optimized RTL.

3.4.1 C-Based Hardware Generation and Optimization without Code Refinement

The original core C code might be as short as shown in Figure 3.2 or Figure 3.3. However, simply taking these codes into the translation tools might generate a hardware design with even poorer performance than a software implementation, as the clock frequency of FPGA is much slower than conventional CPU, and we need a much larger degree of parallelism to get speedup. Automation tools can realize the parallelism via scheduling multiple operations in the control data flow graph (CDFG) at the same cycle. Currently, these tools are not able to extract system-level parallelism automatically, but they provide a set of pragmas that work as hints or directives for invoking parallel execution. These include loop optimization techniques such as unrolling and pipelining, which can increase the performance substantially compared to a generated hardware design without using these techniques.

Loop unrolling and pipelining techniques are provided by the tool to optimize the performance of the nested loop. Loop unrolling can increase the degree of parallelism if the computation is not constrained by memory access of input data or there is some input data reuse between iterations of inner loop bodies. Loop pipelining tries to start the execution of the loop body of the next iteration before completion of the prior iteration, and thus can greatly reduce overall latency of the nested loop. In our case, the unrolling will not help much compared to the pipelined loop as inner loop bodies will need the data from a single memory block with limited ports and there is not much data reuse for the loop. Pipelining did help as it could generate a pipelined loop with a small initiation interval (one clock cycle in our case). But as the execution of loop body without pipelining just uses around five to six clock cycles, the pipelined design still can not completely compensate for the low clock frequency of FPGA to get a decent speedup. AutoPilot recently added pragmas for memory partitioning, but till now they still can not handle the indirect data access in our case very well.

AutoPilot also recently added pragmas to specify the ping-pong buffer-based IO interface. Thus, the overlapping behavior can also be generated automatically.

3.4.2 Code Rewriting/Refinements for the Core Nested Loop

To break up the bottleneck at code generation for the data access, we conduct a set of rewriting shown in previous subsections to increase the bandwidth and throughput for the FPGA platform. The general idea is to partition the memory blocks to allow for concurrent data access, and the access pattern needs to be exploited to develop a good partition scheme. Also the addresses for the memory access after partitioning need to be mapped or generated, and data fetched from different partitions needs to be multiplexed. Besides the memory partitioning we presented in the previous section, another issue is interconnect. It is very difficult for high-level tools to estimate the impact of interconnect at the high level, and in most cases these tools ignore the interconnect issues and only use the delay of the functional unit during scheduling. The code for output data multiplexing shown in Section 3.3.7 only performs some small bit-width comparisons and assignment operations, which do not consume much function unit delay in the modeling of the tools. Thus, tools might chain a lot of comparison and assignment into one cycle. But the interconnect delay will significantly degrade the frequency in this case. We manually add clock boundary in that code to make sure that the number of shifting steps in one cycle is fixed.

Many of these rewritings are intrinsic for hardware design. We feel that the gap between the software C code and the C code suitable for hardware generation still exists. It is unlikely that a pure software designer can master these rewriting techniques, thus there is still much room for synthesis tools to extract the systematic parallelism and automate these refinements and rewritings, especially for users who are developing accelerators for high-performance computing who might not be very familiar with HDL and the memory hierarchy of FPGA.

However, even some rewriting and code tweaking can not be done automatically for the time being, C-based design greatly shortens the development cycle and helps maintenance

Table 3.1: Device information of EP2S180

ALUT	M512 blocks	M4K blocks	M-RAM blocks	Total Memory Bits	DSP/Multiplier	IO Pins
143,520	930	768	9	9,383,040	96 / 384	1170

Table 3.2: Device utilization of the design with 5 by 5 partitioning

ALUT	Memory Bits	DSP/Multiplier	IO pins	Fmax(MHZ)
22,457 (15.6%)	2,972,876(31.7%)	0 (0%)	485(41%)	112.30

of the design. Most of these refinements are specified at algorithmic level using C. We then use the C to HDL compilation tools to generate the RTL for the refined C code, and we also enable the loop pipelining for the nested loop. The performance is greatly increased with the help of large parallelism in the refined code. The core C algorithm is less than one hundred lines of code, while after explicitly memory partitioning, the C code becomes around a thousand lines of code, and the generated RTL contains several tens of thousands of lines. Thus, using the C-based design shall result in a significant saving in terms of design effort compared to a pure manual RTL design.

3.5 Experimental Results

We implement the algorithm on Xtremedata's XD1000 development system [58].

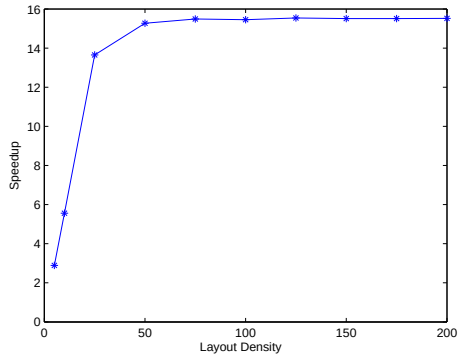


Figure 3.17: Speedup plot with accelerator, single kernel

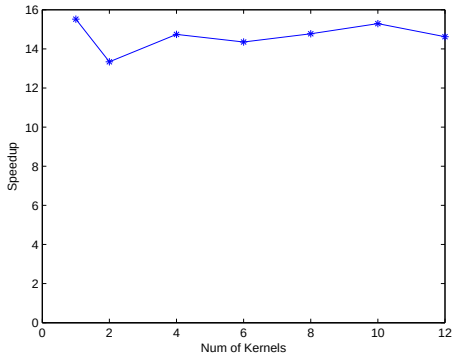


Figure 3.18: Speedup plot with accelerator, multiple kernels, N=200

Table 3.1 shows the device information of Altera Stratix II EP2S180 FPGA.

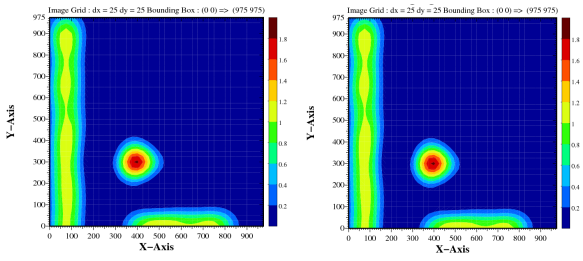
3.5.1 Speedup Measurement

We use a 5 by 5 partitioning scheme and it effectively drives $25 * 2 = 50$ processing elements. *Kernel* array spans a 2000nm by 2000nm area and is a 400*400 array containing 16-bit resolution fixed-point values. The window of image region we simulate has a size of 1000nm by 1000nm, thus image partial sum array is a 40*40 array containing 32-bit resolution fixed-point values. Layout corner array is an array containing up to 800 32-bit values and can store N up to 200 rectangles. All these arrays are stored in the on-chip RAM of the FPGA.

The design has only around a 20% device utilization in logic ALUT and 30% utilization of memory bits; it does not use any multiplier and DSP units. We run the design at 100MHZ. Note that around 8% ALUT is used by the HyperTransport core and SRAM interface cores in the framework, thus the design itself consumes less than 20K ALUT. Table 3.2 shows the device utilization of our design.

We first conduct our experiment on a layout design with size 200um*200um with the setting shown in Section 3.2.3 where we simulate each unit of image region of 1000nm by 1000nm, and the range of the kernel is 2000nm by 2000nm.

We generate the layout with different layout density N. Figures 3.17 and 3.18 depict the speedup curve of the FPGA accelerated version versus the pure software implementation



(a) Fixed point (b) Floating point

Figure 3.19: Contour graph using fixed-point or floating-point computation

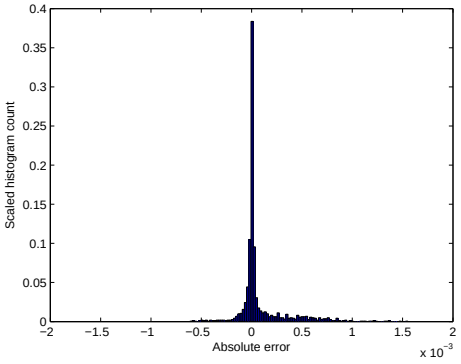


Figure 3.20: Error distribution for the contour graph

Table 3.3: Running time comparison with or without accelerator, single kernel

N	wo accelerator	w accelerator	speedup
5	4.16	1.44	2.89
10	8.01	1.44	5.56
25	19.94	1.46	13.65
50	39.88	2.61	15.27
75	59.80	3.86	15.49
100	79.74	5.16	15.45
125	99.64	6.41	15.54
150	119.63	7.71	15.51
175	139.45	8.99	15.51
200	159.44	10.27	15.52

Table 3.4: Running time comparison with or without accelerator, multiple kernels, N=200

NumofKernel	wo accelerator	w accelerator	speedup
1	159.44	10.27	15.52
2	274.27	20.55	13.34
4	606.37	41.12	14.74
6	885.71	61.70	14.35
8	1216	82.29	14.77
10	1572	102.80	15.29
12	1806	123.46	14.62

Table 3.5: Performance rate (Mpixel/s)comparison of various algorithm and platforms

N	polygon software	polygon FPGA	polygon GPU	2D-FFT software	2D-FFT FPGA [59]	2D-FFT GPU [60]
10	8.0	44.4	32.4	0.2	1.1	3.4
50	1.6	24.5	10.8	0.2	1.1	3.4
100	0.8	12.4	6.7	0.2	1.1	3.4

running on the Opteron CPU. The software implementation runs on the same development box of XD1000 with AMD Opteron 248 (2.2GHZ) 4G DDR memory and is compiled through gcc -O3. The measured speedup factor is around 15. Note that for a very small N, e.g., $N \leq 10$, the speedup we get is small due to the overhead in communications. For a moderate N, we can keep a speedup around 15, as the communication time is smaller than the computation time. We first just use one kernel for simulation. Table 3.3 shows the measured running time and speedup with different N. Then we keep the N fixed and change the number of kernels. That data is shown in Table 3.4.

One limitation of our current design is that we assume the three arrays used for computing using one kernel can be fitted into the on-chip RAM of the FPGA. However, we are not able to fit a larger partitioning with the current setting of input/output data size, although the overall on-chip memory bits have not exceeded the memory bits available in the device. The reason is that around 60% of the on-chip memory in the device is M-RAM, and each M-RAM can store only one memory partition. The more partitioning we use, the larger the percentage of partitioned arrays put into the remaining M-4K and M512 blocks, which will increase the difficulty of fitting the design.

It is possible to fit the design and achieve a higher bandwidth and speedup with a larger partitioning scheme, such as an 8 by 8 partitioning or even larger partitioning scheme, if we use a smaller kernel array or decreased resolution. Another way is to further partition the kernel array and computation into multiple parts and only load one part into the on-chip RAM and only do the computation that uses that part for one time.

3.5.2 Accuracy of the Fixed-point Computation

Using fixed-point computation will not have a big impact on accuracy. We measured the error of the approach versus the software implementation using all floating-point operations, and the absolute error is usually within 1% for the pixels with bright intensity. The relative error for pixels with very small/weak intensity, on the other hand, might be larger because of the truncation error. Figure 3.19 shows two plotted pictures of a 1000nm by 1000nm

region obtained via either software or hardware. We can see that almost no difference can be observed in the contour graph. We also plot the error distribution of the contour graph in Figure 3.20. From the figure, we see the maximum absolute error of all the pixels is around 10^{-3}. Most of the pixels tend to have an even smaller error, which is in line with our analysis in Section 3.2.2. Note that the maximum intensity of that contour graph is 1.83, and the relative error is indeed very small.

3.5.3 Comparison with the FFT-Based Approach and Other Acceleration Techniques

There is no prior published work on accelerating the polygon-based lithography image simulation. However, the 2D FFT-based image convolution has been extensively studied in various platforms. We list them here for reference. Table 3.5 shows the measured/expected performance rate on various platforms (and we assume only one kernel is used here). We use FFTW [52] in the software implementation of the 2D FFT-based convolution. Note that our setting uses different grid sizes for imaging and kernel, say a 5nm grid on the kernel/eigenvectors and layout corners and a 25nm grid for the simulated image, but FFT needs the same grid size for computation. If we use the finer grid size among the two (the kernel grid), it will report the effective performance rate shown in the column of the table. If we use the coarser grid, the effective performance rate will be 25X to 30X larger, but it might lose some accuracy in representing the objects and might cause some accuracy loss in the simulated image. It might be more fair to compare all implementations with a same grid size for kernel and image, yet we do not have data of the FPGA implementation for this setting.

From Table 3.5 we can see, for a moderate density N around 50 to 100, while the polygon-based approach is not as fast as the FFT-based approach using 25nm in the object and eigenvectors, it is much faster than the FFT-based approach using a 5nm grid. (Note that the extra overhead for the FFT-based approach—conversion from the GDSII to the object image, is not included.) In terms of accelerated simulation, our implementation can achieve up to 15X speedup over the software implementation, while the FPGA accelerated FFT-

```
.....
  //Core computation using CUDA
  for(n=blockIdx.x;n<4N;n+=gridDim.x)
   for(x=theadIdx.x;x<pixel_max;x+=blockDim.x)
    for(y=theadIdx.y;y<pixel_max;y+=blockDim.y)
    {
     addr_x=5*x-rect_x[n]+c;
     addr_y=5*y-rect_y[n]+c;
     I_k[x][y]
       +=(-1)^n*kernel[k][addr_x][addr_y];
    }
.....
```

Figure 3.21: Pseudo-code for the core computation using CUDA

based 2D convolution only reported around 5X in the single precision and around 10X in the 16-bit precision [59] using a Virtex-4 device. We also notice that recently FFT-based 2D convolution is shown to achieve very high FLOPS [60] on NVidia G80 with the help of the CUDA Toolkit and CUFFT library; if we use the coarser grid size, it can achieve 90Mpixels/s.

As the algorithm in Figure 3.3 is naturally data parallel, we also implement the algorithm using CUDA. CUDA mainly uses the SPMD (single program multiple data) model. Each thread has a thread ID, and each thread can use the ID to access different data and perform subsequent computation using the data. Figure 3.21 shows the part of the pseudo CUDA code. The parameter $blockDim.x$ and $blockDim.y$ define the number of the threads in one dimension and $gridDim.x$ defines the number of thread blocks. This code will launch $gridDim.x * blockDim.x * blockDim.y$ threads. Further optimizations need to determine the locations for the array data, which affects the effective internal bandwidth for the application. We place the image partial sum array in the shared RAM in the threading block, place the kernel array in texture memory, and place the layout corner array in global memory. The

overall size of the kernel array is too big to be put in the shared memory. Accessing kernel data via texture caching might not give as large a bandwidth as the carefully partitioned on-chip memory of FPGA, but the massive threading offsets the possible latency in memory access.

We test the performance on a 8800GTS video card. The current measured speedup we get for the design is around $8X$. Consider the usage of a higher-end card e.g., 8800GTX, GTX280 and more tuning possibilities, we expect the speedup for our litho design using NVIDIA GPUs should be somewhat the same against our accelerator design using FPGA. Note that GPUs also have their advantage on floating-point, while the FPGA design usually needs to use fixed-point for area efficiency.

External IO bandwidth is critical for certain applications. In this application, Section 3.3.3 gives the estimate on the communication for the FPGA design. The communication time is less than the computation time, and the overlapping scheme removes the communication from the critical path. For the GPU design using CUDA, the (external IO) communications are not overlapped with the computation, but the peak bandwidth of PCI-e x16 is 4GB/s and is sufficiently fast for this application. This makes the communication time only consist of less than 1/10 of the total execution time.

In terms of power consumption, the power for this FPGA design reported by the Quartus Power Analyzer tool is 6.2W and the peak power for this FPGA device is roughly 25W. The TDP (thermal design power) of the Opteron 248 CPU is 95W, and the TDP of the 8800GTS GPU is 147W. (It is difficult for us to measure the actual power for the CPU and GPU at runtime, thus we use the TDP here.) We can see that the power consumption of either GPU or CPU is much larger than the FPGA device. However, the FPGA coprocessor is plugged in a dual CPU motherboard. If we count the power consumption of other parts in the system, e.g., chipset, hard drive in the comparison, the gap on power consumption is not very large. If we consider the possibility of using multiple devices (more than one FPGAs or more than one GPUs), the gap will again become significant. In our design, the FPGA platform could deliver similar performance with much less power consumption.

In terms of ease of use, the CUDA toolkit, as a C development environment for NVIDIA GPU, is very friendly to use; this makes the GPGPU platform very attractive. Users need to rewrite their code to the SPMD form and tune the locations of array data for performance. This tuning needs the deep knowledge of GPU hardware. Traditionally, using the FPGA for computing has been much more difficult than GPU and CPU. C to RTL automation tools help to bridge the gap, and will also make the FPGA-based computing platform increasingly attractive. We share our experience to describe our litho design on FPGA using C to RTL automation tools. We also point out that currently it also needs many hardware-oriented refinements or tuning.

In terms of cost for reaching similar performance, high-end FPGAs for HPC markets are still relatively expensive, thus FPGA designs need to demonstrate larger speedups to justify a more competitive position. This is often the case for examples with high degrees of bit-level parallelism and data/task parallelism. (Note that the design we present does not have bit-level parallelism.)

3.6 Related Work

The use of the memory partitioning scheme is closely related to the conflict-free multi-port memory (or parallel memory) design in multi-media processors. The modulo addressing approach with a circular buffer which could get a row of data containing n elements from n banks is presented in [56]. Parker's skewed scheme [61] can support row, column and diagonal parallel access patterns. Kuzmanov [62] considers the parallel access for a square block of data. The access pattern in lithographic simulation (Chapter 3) is not square block or diagonal, and can not be directly supported by those prior works. We extended the modulo addressing approach [56, 62] to support our case. In these processor-based systems, researchers try to design a circuitry to allow concurrent access for a specific access pattern. Our design is implemented using high-level synthesis tools. Since the tool controls the scheduling of data accesses, it thus has the freedom to select the parallel access patterns [63]. Note that it is possible to perform data layout optimization to reduce our case to that

of [62].

3.7 Conclusions

We present a design for accelerating lithography aerial image simulation using a polygon-based simulation model. The adequate memory banking scheme for the on-chip memory can improve the load-balance and ensure a decent speedup. A 5 by 5 partitioning design can achieve around 15X speedup over software implementation. We also compare against other algorithms and implementations on a GPU.

We see several opportunities for making improvements over the current design. One is that the 2D-ring structure might have a large interconnect delay in the feedback path; thus buffers need to be explicitly inserted, especially when there is a larger partitioning. Another improvement concerns the assumption that at least one kernel can be loaded into the on-chip RAM. This might not be always true for different settings. Thus, further partitioning of the kernel and computation should also be implemented.

Current C to HDL code translation and synthesis tools have already enabled the designer to write and maintain the algorithm and logic in high level, purely in C, and help reduce the development cycle. However, our experience shows that a certain amount of effort is still needed to find a larger parallelism through manual refinement of the C code. More automation is needed for the extraction of systematic parallelism. Also for the mapping of memory models, the compilation tool should not simply convert one array in C into a memory block in HDL, but should provide more flexibility and optimizations on memory models that could possibly do a better job of handling the specific addressing patterns in our design.

CHAPTER 4

FPGA/GPU Accelerated Fluid Registration

4.1 Introduction

Image registration tries to find a transformation function of the coordinate system of one image into the coordinate system of another image. The ultimate goal is to align the two images. In the clinical setting, we may want to perform image registration of the image studies taken of the same patient to see the progressive development of the illness (e.g., tumors). Image registration is also broadly used for remote sensing and computer vision as well.

While there are various kinds of image registration algorithms, we can normally break the algorithm into four parts: *transform*, *interpolator*, *metric*, and the *optimizer* on the metric. Figure 4.1 shows the block diagram of a typical image registration algorithm.

Transform declares the type of transforms (or coordinate mappings) from the coordinates in the reference image to the coordinates in the target image space. An affine transform models translation, scaling, rotation, shear mapping, etc. A rigid transform model is a subset of the affine transform which only allows translation, rotation and reflection. The deformable transform allows a more general transform formulation which can map a coordinate in the reference image space into any coordinate in the target image space. *Interpolator* performs interpolation in order to obtain the pixel value at the non-integer coordinates. *Metric* is the objective we need to optimize. Typical similarity metrics for the two images include the sum of squared differences (SSD), mutual information and cross-correlation, etc. In order to restrict the transform or displacement functions, an additional regularization term on the transform or the displacement is also needed. Typical *optimization* schemes include steepest

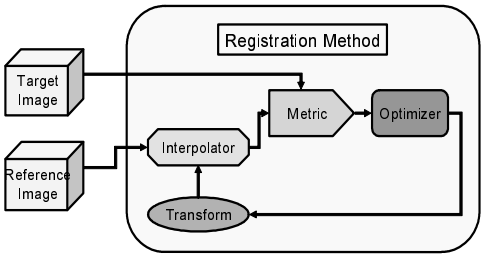


Figure 4.1: Block diagram of image registration algorithms

descent, conjugate gradient, and some global optimization algorithms.

Pioneering work on FPGA-based medical image registration includes FAIR [64], FAIR II [65] and the recent work by the same group that uses the sub-volume mutual information based method [66]. FAIR and FAIR II use mutual information as the optimization metric. Mutual information (MI) is particularly useful for multi-modality (e.g., CT vs. MRI) registration. Rigid transform is used in FAIR and FAIR II. Rigid transform can be used to find the global alignments, but most distortions (such as respiration effects) in the imaging process require a deformable transform. The sub-volume MI based method [66] hierarchically divides the image studies into sub-volumes, and allows rigid transform to be applied on each sub-volume. Clearly, this is a deformable transform because different sub-volumes may use different rigid transforms. The drawback is that the transform is not smooth—in particular for the voxels at the boundaries of a sub-volume.

We try to accelerate a PDE-based non-parametric registration called fluid registration. Fluid registration regularizes the deformation using a fluid PDE equation, and it allows registrations of large deformations. Fluid regularizers ensure that the transform function is smooth. Currently, the speed of the algorithm cannot meet the clinical need. It may take close to one hour to finish up a typical registration (image with size 256^3, 500 iterations) on a standard workstation. The application is both compute-intensive and data-intensive. This work tries to accelerate the fluid registration algorithm on a multi-FPGA platform. The highlights of our implementation are as follows. First, in the fluid registration algorithm that we implement, the transform is described through a point-wise displacement function u_1, u_2, u_3 where a point at (i, j, k) will get mapped to coordinate

$((i - u_1(i,j,k), j - u_2(i,j,k), k - u_3(i,j,k))$; thus each voxel (rather than a sub-volume) can move independently in a deformable fashion. Second, to accelerate this data-intensive application, we employ several source-code level optimizations that include fixed-point conversion, tiling, prefetching, data-reuse, and streaming across modules using a ghost zone (time-tiling) approach. We evaluate the impact of these optimizations. Third, we use a high-level synthesis tool to speed up the implementation process, and identify their pros and cons through this complex case study. We have reference code that uses either sum of square differences (SSD) or mutual information as the optimization metric. Currently, we apply the registration on the image studies with the same modality; thus, we use simpler SSD as the optimization metric.

4.2 Fluid Registration Algorithm Review

Fluid registration [67] performs smoothing on the velocity field v (or so-called incremental deformation field) rather than the total deformation field u. Solving large-scale PDE, especially for 3D image registration, (e.g., Navier-Stokes PDE for fluid registration), is a computationally expensive process. Bro-Nielsen et al. [68] proposed to use a scale-space convolution filter to accelerate the smoothing process. Our baseline implementation is a variant of the implementation in [68], except that we simply use a Gaussian filter to smooth the velocity field. Gaussian filter is used in the optical-flow based Demons algorithm [69], and also used in several more recent fluid registration works [70, 71].

The key mathematical derivations for fluid registration can be seen in [67, 68, 72]. Here we briefly review them for completeness. The two image studies are S and T. The deformation field is termed u. In each iteration, we first perform linear interpolation based on the deformation field:

$$\widetilde{T}(x,t) = T(x - u(x,t)) \tag{4.1}$$

We obtain the force field using the derivative of an L2 SSD metric:

$$f(x,u(x,t)) = -[\widetilde{T}(x,t) - S(x)]\nabla\widetilde{T}(x,t) \tag{4.2}$$

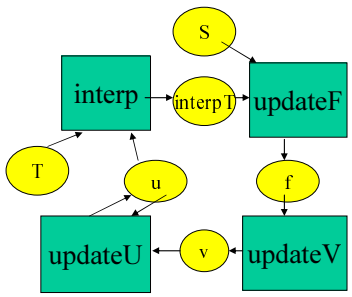


Figure 4.2: Dataflow between procedures

Instantaneous velocity $v(x,t)$ can be obtained by solving the fluid PDE:

$$\mu\Delta v(x,t) + (\mu+\lambda)\nabla div\ v(x,t) = f(x,u(x,t)) \tag{4.3}$$

In our implementation, we simply use a Gaussian convolution as in [70, 71].

$$v(x,t) \approx G_\sigma * f(x,u(x,t)) \tag{4.4}$$

We use the recursive Gaussian infinite impulse response (IIR) filter proposed by Alvarez and Mazorra [73]. This IIR only needs two MADD operations per dimension. This is the fastest 3D Gaussian filter we know of. It is one magnitude faster than the FFT-based convolution used in [71], and should also be faster than direct convolution and other recursive Gaussian IIR, e.g., those by Young [74]. Note that an additional normalization step can be fused into the subsequent computation procedures. After that, we obtain an updated deformation field by solving the PDE $du(x,t)/dt = v(x,t) - v(x,t)\nabla u(x,t)$, using an explicit Euler scheme:

$$R(x,t^i) = (v(x,t^i) - v(x,t^i)\nabla u(x,t^i)) \tag{4.5}$$

$$u(x,t^{i+1}) = u(x,t^i) + (t^{i+1}-t^i)R(x,t^i) \tag{4.6}$$

The advancement of timestep needs to be bounded so that $(t^{i+1}-t^i)\ max\|R(x,t^i)\|_2$ does not exceed the maximum displacement allowed in one iteration.

4.3 Module-Level Implementation

The whole fluid registration in our baseline contains the following computation kernels: *interp* which corresponds to Eq 4.1; *updateF* which corresponds to Eq 4.2; *updateV* which

corresponds to Eq 4.4 and *updateU* which corresponds to Eq 4.5 and 4.6. The registration algorithm is an iterative process that repeatedly calls these procedures. The dataflow between the procedures is shown in Figure 4.2. When non-core parts (file IO etc.) are excluded, it takes around 6s per iteration on an Intel Xeon 2.0GHZ CPU. Approximately 45% of the time is spent on Gaussian smoothing, 35% on displacement update, 15% on interpolation, and 5% on force calculation.

4.3.1 Gaussian Smoothing

Gaussian smoothing essentially performs

$$\widetilde{A} = G_\sigma \otimes A \tag{4.7}$$

where G_σ is a Gaussian function and $\otimes$ is the convolution operator. The 3D Gaussian function is

$$G_\sigma(x, y, z) = \frac{1}{(2\pi\sigma^2)^{\frac{3}{2}}} e^{-\frac{x^2+y^2+z^2}{2\sigma^2}} \tag{4.8}$$

There are many algorithms and implementations of Gaussian smoothing. Roughly these can be divided into three categories: FFT-based convolution, direct convolution, and recursive impulse infinite response (IIR) filtering.

The FFT-based method operates on the frequency domain. It is a very accurate implementation but typically involves more computations. The complexity of the FFT-based convolution is $O(NlogM)$, where N is the number of pixels/voxels and M is the radius of the convolution kernel. Direct convolution performs weighted sum operation for each output pixel/voxel. The kernel size is normally selected in proportion to the variance σ of the Gaussian filter. The complexity is $O(NM)$ because the number of operations per output pixel in the direct convolution method grows proportionally with the kernel size. For small convolution kernels, direct convolution is preferable because the sweeping for weighted sum computation exploits a good data locality. However, the extra computation involved quickly outweighs the benefit when the convolution radius is large. On the other hand, the computation involved in the recursive IIR does not depend on the size of Gaussian kernels. Famous recursive Gaussian IIR algorithms include Deriche-style IIR [75] (12 MADDs per dimension)

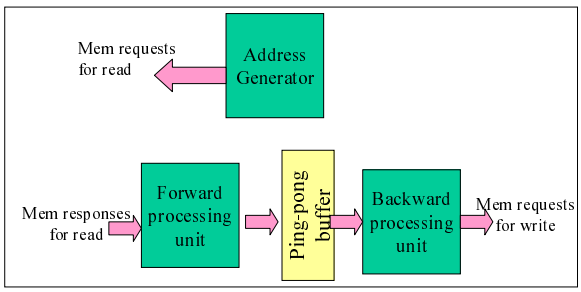


Figure 4.3: 1D IIR PE

```
//Input: array u_tmp
//Output: array u_tmp after in-place update
//Backward:
u[0] = u[0] * c1;
for(i = 1; i < N-1; i++)
 u[i] = u[i] + u[i-1] * c2;
//Forward:
u[N-1] = u[N-1] * c1;
for(i = N-2; i >= 0; i--)
 u[i] = u[i] + u[i+1] * c2;
```

Figure 4.4: 1D Gaussian IIR

and Young-van Vliet IIR [74] (6 MADDs per dimension). CUDA SDK provides an example that implements 2D Gaussian IIR based on Deriche's algorithm. In this chapter, the baseline implementation features a much smaller computation (2 MADDs per dimension). The algorithm is proposed by Alvarez and Mazorra in [73]. They employ the relationship between Gaussian smoothing and a heat equation PDE to provide a very fast smoothing algorithm. The complexity is $O(N)$. The implementation techniques presented here should be applicable to other IIR as well.

Recursive IIR smoothing works through the sweeping of the 3D image in the three dimensions (or six directions) sequentially. Suppose the image size is 256^3, we need to perform 256^2 1D IIRs in each dimension.

Our 1D IIR uses the recursive computation in each direction:

$$y(n) \approx a * x(n) + b * y(n-1) \tag{4.9}$$

where $y(n)$ denotes the new signal sequence that is generated by the IIR, and $x(n)$ means the input signal sequence. Figure 4.4 shows the code for 1D IIR.

We design the hardware for the Gaussian IIR in a way that there is a large number of parallel PEs that talk to different memory ports. Each FPGA is exposed to 16 virtual memory ports, and each processing element (PE) talks to 2, thus we realize 8 parallel PEs. Each PE (Figure 4.3) is a hardware unit that can compute a group of 1D IIR in an interleaved fashion. (We need to do such interleaving to hide the latency of the MADD unit.) Each PE is composed of three components. One component is the address generator hardware unit that keeps sending memory access requests into one memory access port. In parallel with the address generator, the main task-level pipeline includes the backward processing unit and forward processing unit. The backward processing unit reads memory responses, performs computation, and writes data into the on-chip BRAM (ping-pong buffer). The forward processing unit reads data from the ping-pong buffer and sends requests to another memory access port. The backward processing unit and the forward processing unit work in parallel. The computation within the forward processing unit and backward processing unit are deeply pipelined using standard loop pipelining techniques. We realize 8 PEs on each FPGA. Our multi-FPGA design incorporates 32 PEs and utilizes all the memory access ports available in the platform. Additional implementation issues of this module on multi-core CPU, many-core GPU, and FPGAs are shown in [76].

4.3.2 Displacement Update

This is a kernel that mainly performs stencil data access and performs some finite difference calculation. The input of this module is the deformation field u and velocity field v, the output of this module is the updated deformation field. This corresponds to Eq 4.5 and 4.6. To compute ∇u_1 at index (i, j, k), we need to compute:

$$\frac{\partial u_1}{\partial x}(i,j,k) = (u_1(i+1,j,k) - u_1(i-1,j,k))/2 \tag{4.10}$$

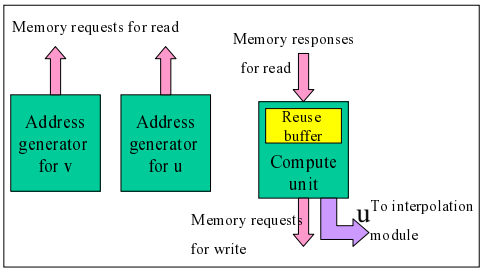


Figure 4.5: Displacement update

$$\frac{\partial u_1}{\partial y}(i,j,k) = (u_1(i,j+1,k) - u_1(i,j-1,k))/2 \tag{4.11}$$

$$\frac{\partial u_1}{\partial z}(i,j,k) = (u_1(i,j,k+1) - u_1(i,j,k-1))/2 \tag{4.12}$$

The equations for ∇u_2 and ∇u_3 are similar.

We assign two memory ports for reading u, two memory ports for reading v and two ports for writing u. The total memory ports available for one FPGA is 16. The computed u is also streamed to the interpolation module. This module is implemented by three independent components: the address generator for u, the address generator for v and the main computation unit. A small scratchpad is instantiated inside the computation unit to capture the data reuse opportunities. A diagram for this module is shown in Figure 4.5.

4.3.3 Interpolation

We need to access 2 by 2 by 2 voxels of the target image T to compute each voxel in the interpolated image. This module is very data-intensive. We use seven external memory access ports for this module. Six ports are used to access target image T, and another port is used to write interpolated image $interpT$. The computed $interpT$ is also streamed to the force calculation module. All the 3D objects used in our implementation are stored using 32 bits. For each voxel, we send four requests to the request FIFO. At the response FIFO side, four units of 64-bit data are fed into the interpolation unit. An obvious possible optimization is to use 16-bit or 8-bit to store the target image, so that we can fetch the required data using fewer requests.

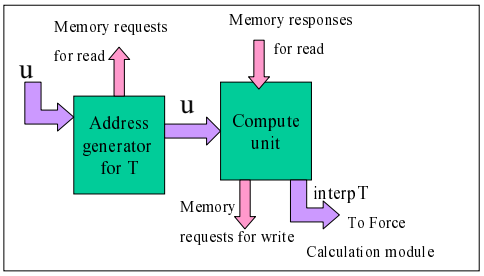


Figure 4.6: Interpolation module

This module is implemented using two components. One component reads in the data that is streamed by the displacement update module and computes the addresses for voxel access. Another component performs the weighted sum computation for interpolation. Both the address generation and weight calculation are computed based on the displacement value. Because of this, the displacement is also streamed from the address generator into the weighted sum computation unit. The diagram of this module is shown in Figure 4.6.

4.3.4 Force Calculation

This is another kernel that mainly performs stencil data access and finite difference calculation. One port is used to read S and two ports are used to write f. The overall architecture is very similar to that of displacement update (although the actual computation performed is different).

4.3.5 Initial Implementation

Our initial implementation uses the high-level synthesis tool AutoPilot [25] to obtain a baseline design. Modifications of the code are restricted to those that are related to platform-specific external memory interfaces. External data accesses are converted to corresponding memory requests/responses directly. Functional modules are generated one by one and are invoked sequentially. Floating point is used throughout the design. Although it works correctly on the board, that design takes around 1s per iteration. AutoPilot does not perform sharing across function (module) hierarchies, so the floating point units will be instantiated

for each module. Because this causes some fitting issue, we need to downsize the computation modules. In the next section we will talk about several optimizations that we did to improve the performance Note that the diagrams for the modules we described in this section already reflect some of our optimization schemes.

4.4 Optimizations

The algorithm is both compute-intensive and data-intensive. When we are using floating-point as the data representations, the implementation is more bounded by area. After we convert the algorithm into fixed-point, it is then bounded by off-chip memory bandwidth. We use several techniques to reduce the total data traffic to save the bandwidth.

4.4.1 Algorithm Adaption

4.4.1.1 Conversion to Fixed-Point Computation

The whole registration process is an iterative procedure, and the accuracy can be a trade-off with slightly more iterations. Note that a similar study [77] was conducted for other application domains.

We manually convert the code from floating-point to fixed-point. We perform the range analysis on each procedure to obtain the integer bits of the fixed-point data. Because of the iterative nature of the algorithm, conventional static precision analysis will not apply as the errors may accumulate rapidly. In our implementation, we set the fractional bits to be 10. The convergence curves with different fractional bits are shown in the experimental results section.

AutoPilot provides arbitrary precision integer (APInt) and arbitrary precision fixed-point (APFixed) to describe integers and fixed-point data format. We use APInt to describe our design.

4.4.1.2 Removal of Reduction Steps

The original code needs to compute $max\|R(x,t^i)\|_2$ to determine the timestep. Max operation is a reduction process which breaks time-tiling (discussed in subsection 4.4.4 and 4.4.5). We instead use a constant timestep in our implementation. The timestep is parameterized through user-input.

4.4.2 Prefetching

Off-chip memory access has a long latency. The memory system used in Convey HC-1 optimizes for the scatter-gather type of random access rather than burst access. We need to employ prefetching to obtain good memory-system performance.

In our implementation, we model each memory access port with a request FIFO and a response FIFO. We decoupled the "helper threads" that are responsible for sending memory requests for reads, and the "compute threads" that obtain data from response FIFOs. This way, the helper threads can keep sending as many requests as possible (until the FIFO is full). Effectively, the helper thread is performing the prefetching of the required data, and the response FIFO serves as the prefetch buffer. Note that our helper thread only performs prefetching for reads. Off-chip memory writes are still performed by the compute threads. In the Convey system, memory write requests do not generate responses in the response FIFO.

The address generator (or helper thread) and the computation unit (compute thread) are realized by functions with no dependence between each other. However, the scheduler of HLS may or may not schedule them precisely at a same state. If they are not scheduled to execute at a same state, deadlock may happen. In our implementation, we use some tool-specific tricks, where we create one additional function hierarchy that includes the address generator and the computing functions, to ensure that the parallel functions are scheduled at a same state.

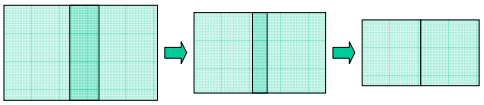


Figure 4.7: Ghost-zone in 2D

4.4.3 Data Reuse

The computation of $\nabla\widetilde{T}(x,t)$ and $\nabla u(x,t^i)$ involves finite difference stencil access. A reuse buffer is used to decrease the bandwidth pressure. The stencil used in our application is a typical 7-point stencil where we need to access voxels with indexes (i,j,k), $(i+1,j,k)$, $(i-1,j,k)$ $(i,j+1,k)$,$(i,j-1,k)$,$(i,j,k+1)$ and $(i,j,k-1)$ to perform the finite difference computation. The size of the reuse buffer can be kept as 3*256*256 in the simplest case, or three 2D slices. (The maximum image we deal with is 256^3.) When we move on to process a new 2D slice, we discard the oldest slice in the reuse buffer and write the newly fetched data from the response FIFO into that slice. Similar techniques are used in many applications in literature that use stencil access such as [78, 79]. However, the reuse buffer that holds three 2D slices is too large to fit or get a good timing closure. In our final implementation, the reuse buffer is allocated as 3*20*256, where we do spatial tiling on one iteration dimension as well. We observed a 2X performance improvement for the 3D IIR, and 4 to 5X performance improvement for the stencil computation modules (displacement update and force calculation), because the amount of external data accesses is reduced significantly.

The reuse buffer is referenced using addresses involving *mod* operations. (Note that we use a circular increment scheme to realize the *mod* operation.) The memory partitioning optimization pass inside AutoPilot fails to analyze it properly but treats the data accesses as generic indirect accesses. We manually partition the reuse buffer to a number of banks to allow concurrent accesses. This motivates research to improve the memory partitioning passes for *mod* operations [80].

4.4.4 Streaming Across Modules

If we invoke the four procedures in a sequential fashion, each hardware unit only gets a 25% utilization on average. Instead, we stream the newly computed data from one module to another module, so that several modules can execute in parallel. Because of the nature of stencil access, the consumer module can only start computation when a few slices in the reuse buffer have been filled up.

The Gaussian IIR module performs sweeping in three dimensions. Although that processing step is not a reduction step, the data access pattern is neither sequential access nor stencil access. Currently, the streaming only occurs in the remaining three modules. This also translates to an obvious performance enhancement. The initial implementation invoked the three modules in a sequential way, and we now allow them to work in parallel. This is important for FPGA-based designs, because it needs to combine data parallelism with deep pipelining or streaming to get good performance.

4.4.5 Time Tiling Using a Ghost-Zone Approach

We have two procedures that involve 7-point stencil access. For one procedure that works on a input data tile with size 256*20*256, an output data with size 254*18*254 can be generated. When this amount of data passes through another stencil access procedure, it will generate an output data with size 252*16*252. This effect is called ghost-zone, where the output data size gradually decreases because of the stencil access. Note that the ghost-zone approach involves the recomputation of some intermediate data to avoid communications and synchronization. We adapt this approach in our implementation. The ghost-zone approach is initially used for optimizing CUDA programs in [81]. A diagram that illustrates a 2D ghost-zone is shown in Figure 4.7. Note that our case is in 3D. With the ghost-zone tiling and streaming, the effective performance of the three modules except 3D IIR (displacement update, interpolation and force calculation), is improved by around 2X.

Table 4.1: Performance comparison of 3D Gaussian IIR

Image	CPU (single-thread)	CPU (4-thread)	GPU	Speedup	FPGA	Speedup
$256 * 256 * 64$	0.179	0.046	0.0070	25.6X	0.0055	32.5X
$256 * 256 * 128$	0.359	0.092	0.0139	25.8X	0.0109	32.9X
$256 * 256 * 256$	0.719	0.186	0.027	26.6X	0.0216	33.3X

Table 4.2: Performance comparison of the remaining modules

Image	CPU (single-thread)	CPU (4-thread)	GPU	Speedup	FPGA	Speedup
$256 * 256 * 64$	0.82	0.21	0.020	41.0X	0.023	35.6X
$256 * 256 * 128$	1.59	0.40	0.039	40.8X	0.044	36.1X
$256 * 256 * 256$	3.07	0.79	0.075	40.9X	0.087	35.3X

Table 4.3: Overall performance comparison

Image	CPU (single-thread)	CPU (4-thread)	GPU	Speedup	FPGA	Speedup
$256 * 256 * 64$	1.37	0.38	0.041	33.4X	0.039	35.1X
$256 * 256 * 128$	2.67	0.73	0.081	33.0X	0.077	34.7X
$256 * 256 * 256$	5.24	1.41	0.156	33.6X	0.152	34.5X

4.4.6 Towards Automated Code Generation

We use the HLS tool AutoPilot to obtain the implementation. The good thing is that this enables us to design using C rather than RTL, and it produces correct RTL. The feature we use extensively in this implementation is loop pipelining. Note that AutoPilot only generates computing IPs. We also work with Convey to develop platform-specific interfaces (RTL wrappers and C interface headers) for AutoPilot. These can be viewed as a one-time process and can be reused for other designs. However, as the optimization steps described by the previous subsections are performed manually, this is still a very tedious step. This case study motivates us to do further automated code generation at task level to obtain an efficient implementation.

In the registration implementation, we manually perform many optimizations such as tiling, prefetch, data reuse and streaming, etc. There are several academic efforts that may aid our process, but they are perhaps not directly applicable. KPNGen [82] derives process networks from sequential C code. It does not perform tiling, and thus it may generate very large FIFO buffers (may not fit) for our registration design. We use tiling with the ghost-zone approach to resolve the issue. The ghost-zone approach is initially used for optimizing CUDA programs in [81], and the corresponding compiler approach is called overlapped tiling [83]. The work in [84] combines the optimization for loop-level data parallelism and data reuse. However, it primarily works for a single loop nest. In our case, we need to work with several modules that are streaming with each other. The sliding window optimization in [78] works in the 2D case and also does not consider inter-module streaming. Potentially, one may also leverage the automated prefetching and data reuse analysis in [55, 85, 86].

4.5 Experimental Results

We implement the whole algorithm on a multi-FPGA platform Convey HC-1. The design is described using synthesiziable C code and then converted to Verilog RTL using AutoPilot version 2010.a.3. The RTL is connected to memory interfaces and control interfaces provided

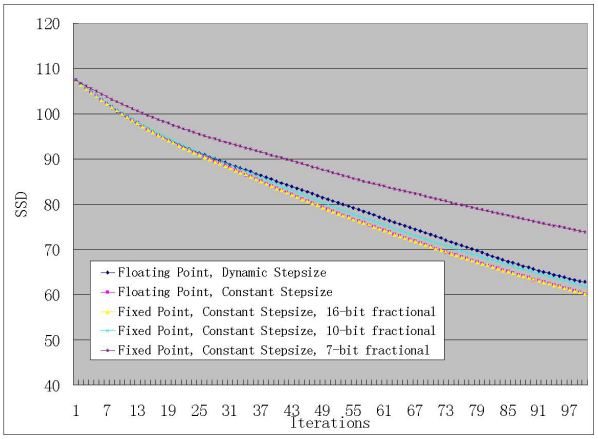


Figure 4.8: Accuracy comparison

by Convey. Xilinx ISE 11.5 is used to obtain the final bitstream. A single bitstream is used to configure all four user FPGAs. Different FPGAs work on different data tiles in a way similar to the SPMD (single program multiple data) scheme.

4.5.1 Accuracy Comparison

Using a constant time-step may change the curve of convergence. When the time-step is chosen properly, the algorithm converges in similar (or even smaller) number of iterations. In general, an implementation that uses the dynamic time-steps is more stable and should also converge faster. Our fixed-point conversion results in only marginal accuracy degradation. We plot the curve of SSD in Figure 4.8. We can see that a fixed-point implementation using 10 bits in the fractional part has an SSD curve that is very close to the curve of the floating-point implementation. An implementation that uses 7 bits for the fractional part has noticeable error and slower convergence.

4.5.2 Performance of Our Implementation

Typically, we need to run hundreds of iterations to get a good registered image. We run 200 iterations in our experiment. One of our test image sets of size 256^3 is shown in Figure 4.9. Note that because the images are 3D, we can only show a few 2D slices. S and T are

Table 4.4: Area results of our design

	Utilized	Total Available	Utilization Ratio
LUT	84,246	207,360	40%
FF	79,456	207,360	38%
Slice	32,082	51,840	61%
BRAM	176	288	61%
DSP	72	192	37%

the two images we register, *interpT* is the output image. As one can see, the registered output image of FPGA-based implementation (column 4) is very close to the reference one (column 3). Note that *interpT* is getting closer to S with the registration. In the beginning of the registration, the output image is the same as the second column. Overall, we obtain around a 35X speedup compared to single-threaded CPU implementation, and 9X speedup compared to 4-thread CPU implementation. We also implement the algorithm on a many-core GPU (Tesla C1060). The speedup is comparable to our multi-FPGA implementation. Data are shown in Table 4.3. The results shown in the table are the averaged value for a single iteration.

Note our Gaussian IIR implementation is faster than the GPU implementation. Each 1D IIR needs a small working set (256 elements in our case) to capture the data reuse. This limits the available parallelism that we can put in each GPU stream multiprocessor. If we do not exploit this reuse on the GPU, we can get a faster implementation. However, that results in 2X bandwidth pressure and is still slower than our multi-FPGA implementation that exploits the reuse. The results are shown in Table 4.1. Note, in each iteration of fluid registration, we need to invoke the 3D Gaussian IIR three times as we have force fields $f1$, $f2$ and $f3$. The data shown in Table 4.1 are for a single 3D IIR.

For the remaining modules, the on-chip scratchpad in the GPU serves as the reuse buffer. It is hard to realize the streaming approach in the GPU that we've realized in the FPGAs. However, the ghost-zone approach within a stream multiprocessor like [81] still applies.

Moreover, the GPU has dedicated texture caches and texture units to perform 3D interpolation. Thus the GPU implementation of the remaining modules is a little faster than our multi-FPGA implementation. The results are shown in Table 4.2.

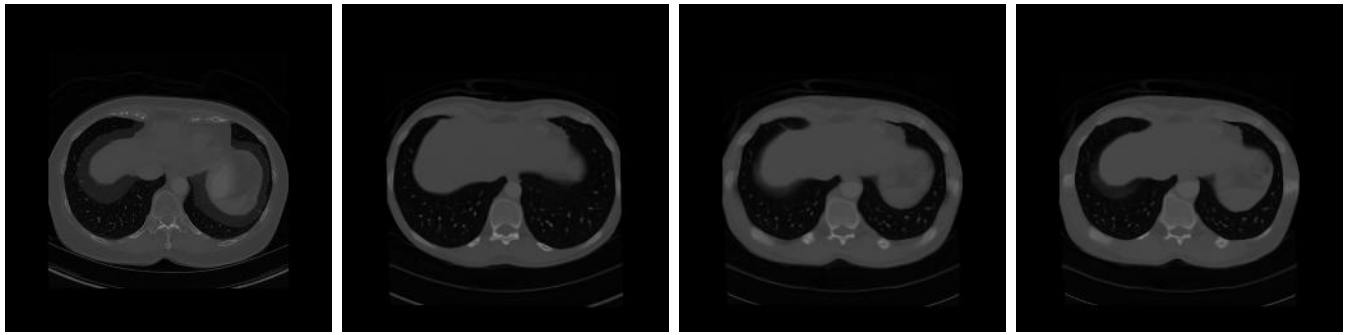
The area results of our FPGA design are listed in Table 4.4. Note these numbers are for a single FPGA bitstream. Our design runs at 150MHz while the memory controller runs at 300MHz.

4.5.3 Power Consumption

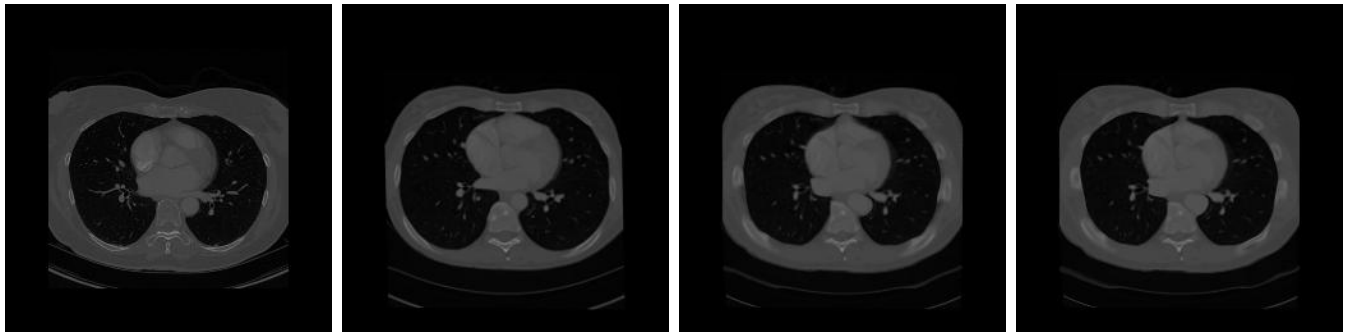
The Xilinx power analyzer reports that each FPGA design consumes 22W. So the 4-FPGA design consumes 88W. The TDP of the Tesla GPU card is around 200W. Our FPGA implementation delivers a slightly better performance while consuming less than half of the power of the Tesla GPU card.

4.6 Conclusions

We present the implementation of a deformable medical image registration algorithm called " fluid registration" on the multi-FPGA platform. We detail our application-specific optimization strategy to make the design competitive. We also show that commercial HLS tools still need to embrace more automation to further enhance design productivity.



(a) S, slice 81 (b) T, slice 81 (c) interpT, slice 81 (by C-PUs) (d) interpT, slice 81 (by F-PGAs)



(e) S, slice 126 (f) T, slice 126 (g) interpT, slice 126 (by C-PUs) (h) interpT, slice 126 (by F-PGAs)

Figure 4.9: One test image set with size 256^3

CHAPTER 5

FPGA/GPU Accelerated CT Reconstruction using Compressive Sensing

5.1 Introduction

Computerized tomography (CT) plays a critical role in modern medicine. However, the radiation associated with CT is significant, and scientists are exploring various approaches to reduce that. Compressive sensing methods are among the mathematical approaches that can potentially enable CT imaging with less radiation exposure but without sacrificing image quality.

The conventional image reconstruction method used in CT is the Feldkamp-Davis-Kress (FDK) algorithm, and currently it is still widely deployed by the manufacturers in clinical settings. The computation kernel of the FDK algorithm is called filtered back projection (FBP). [2] Many researchers have proposed accelerated engines for these algorithms using GPUs or FPGAs [87, 88, 89, 90, 91], and reported remarkable speedups. The FDK/FBP reconstruction directly calculates the image in a single backward reconstruction step using analytical expressions. However, when we reduce the radiation dose, that presents a case of incomplete data. the FDK method is no longer suitable, or generates a very poor-quality image. Iterative reconstruction or compressive sensing techniques are much more insensitive to noise and provide greater flexibility. The data can be collected over any set of lines, the projections do not have to be distributed uniformly in angle, and can be incomplete.

Various algorithms and implementations of iterative reconstruction or compressive sens-

[2] FBP generally only works for 2D images. For parallel beam CT [87], FBP can be applied since images can then be easily decomposed into 2D slices.

ing are proposed, with different objectives or regularization terms. However, many of these algorithms, such as Expectation Maximization (EM)[92], Simultaneous Algebraic Reconstruction Technique (SART)[93], etc., share a common computation procedure (with minor differences in scaling etc.), that includes a forward ray tracing step (forward projection) and a backward ray tracing step (backward projection). Prior work used GPU [94, 95] to accelerate iterative reconstruction like SART, and obtained good speedup over the CPU. However, prior FPGA implementation of iterative reconstruction is scarce with an exception in [96]. But that work also just implemented backward projection in FPGA and left forward projection to GPU.

This chapter presents an FPGA implementation of an iterative reconstruction algorithm called EM+TV [97] that is a variant of the classic EM algorithm [92]. The computation involved in the ray tracing steps is simple MADD; however, a lot of off-chip random access occurs. Moreover, the computation and the required data access for one ray is proportional to the intersection length between the ray and the object. This also poses a load balancing issue. Because of these facts, we think this application is a natural fit for FPGA-based custom computing, compared to the massive parallel GPU device which prefers coalesced data access and balanced load. We implemented the ray-tracing forward projection and backward projection onto the Convey HC-1ex multi-FPGA platform. The remaining stencil computation kernel for total variation regularization (TV) is left to GPUs which will simplify the design of our FPGA hardware. The highlight of our implementation includes:

- Shared hardware module that can support both forward projection and backward projection.
- Separation of the machine configuration and the tracing engine.
- Better performance in terms of latency or throughput than GPU implementation on Tesla or Fermi.
- A working implementation done at C-level by using AutoESL high-level-synthesis tool from Xilinx. Caveats about using the tool to target the high-performance reconfig-

urable hardware are documented in detail.

5.2 EM+TV Algorithm

5.2.1 Algorithm Overview

At high-level, the reconstruction tries to recover signal (vector or images) x from measurements b where $Ax = b$. A is a $M \times N$ matrix describing the transform from the original image to measurements; M is the number of measurements, and N is the dimension of the image. The system is underdetermined as we want to reconstruct the whole image using fewer samples (thus reducing the radiation). Least-square methods optimize for the $||Ax - b||_2$, and are easier to compute through the pseudo-inverse. However, the resulting solution is usually not sparse. Compressive sensing is the approach that tries to find a sparse solution via minimization of 1-norm $||Ax - b||_1$ or 0-norm $||Ax - b||_0$ or other objectives that can maintain the sparsity.

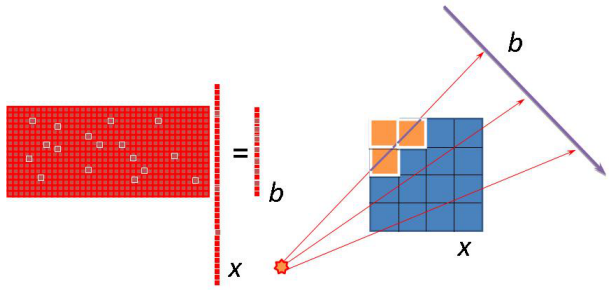


Figure 5.1: Ray tracing in forward projection

Our baseline compressive sensing implementation is called EM+TV, a recent development proposed in [97] which is based on the combination of: 1) expectation maximization (EM) [92], an iterative method that maximizes the likelihood function (in our case, we use that to perform CT image reconstruction under a Poisson noise assumption); and 2) total variation (TV) regularization, which has been used to preserve edges, given the assumption that most images are piecewise constant.

EM+TV reconstruction [97] tries to solve the non-linear optimization problem:

$$\min_x \int_\Omega |\nabla x| + \alpha \sum_{i=1}^{M}((Ax)_i - b_i \log(Ax)_i)$$
$$x_j \geq 0, j = 1, \cdots, N \tag{5.1}$$

The first term is the TV term and the second is the EM term. We omit the mathematical derivation details which can be found in [97]. The constraint optimization problem is solved using the following semi-implicit iterative scheme.

```
Input: x^0 = 1;
  for Out = 1 : N do
    x̃^0 = x^{Out-1};
    for k = 1 : 1 : K do
      x̃^k = EMupdate(x̃^{k-1});
    end
  x^{Out} = TVupdate(x̃^K);
end
```

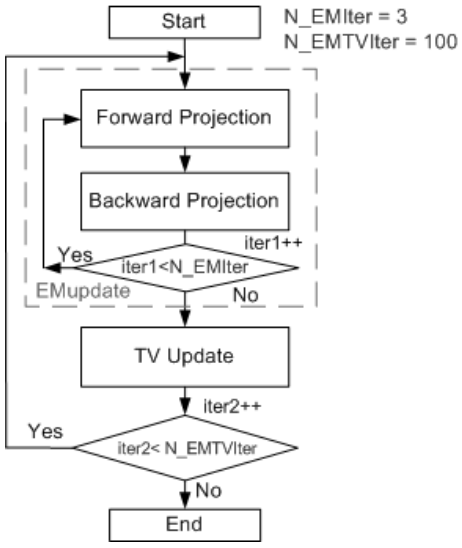


Figure 5.2: EM+TV block diagram

Figure 5.2 shows the overall flow chart for the EM+TV algorithm. It has two updating modules, EMupdate and TVupdate. *EMupdate* is more critical because it occurs in the

inner-most loop. *EMupdate* is trying to perform EM iterations

$$\widetilde{x}_j^{k+1} = \frac{\sum_{i=1}^{M}(a_{ij}y_i)}{\sum_{i=1}^{M} a_{ij}}\widetilde{x}_j^k \tag{5.2}$$

where

$$y_i = (\frac{b_i}{(A\widetilde{x}^k)_i}) \tag{5.3}$$

Inside the *EMupdate* kernel, we need to do forward projection to obtain $A\widetilde{x}^k$, perform element-wise division to obtain y, do backward projection to obtain $A^T y$ (or $\sum_{i=1}^{M}(a_{ij}y_i)$), and then obtain the updated value $\widetilde{x}_j^{k+1}$ using element-wise scaling. Note that because matrix A is very large and sparse, A is never constructed explicitly. A ray-tracing based technique is used to compute the forward projection and backward projection. Figure 5.1 illustrates the ray-tracing technique in a forward projection.

Details for *TVupdate* are omitted as *EMupdate* occupies the majority of the computation time. The tracer engine presented below is responsible for computing the forward projection Ax and backward projection $A^T y$. This algorithm is much more computationally intensive, because it needs to invoke forward and backward projection repeatedly (100*3 times in the setting of Figure 5.2). The conventional FDK algorithm only has a single backward projection.[3]

5.2.2 Tracer Engine

In the EM+TV 3D application, forward and backward projections have the same iterative hierarchical structure. The first level of iteration comprises the number of views (sources of the ray), and the other two layers consist of the array of 2D detectors/sensors. The ray tracer engine works on one source and detector pair and is the computation kernel of both forward and backward projections. As illustrated in Figure 5.3, the ray tracer is composed of two parts: tracer_precal and tracer_loop. For forward projection and backward projection, the tracer has a similar computation structure. The tracer_precal part is exactly same. The only

[3]Scaling required by our application is also fused into the projection in our implementation. However, different algorithms, e.g., SART vs EM use different scaling. We try to keep our discussion on ray tracing as general as possible and ignore the scaling in the following sections of the chapter.

difference is in tracer_loop. In forward projection, the tracer_loop will read pixels along with the ray and output one sino value for each ray; while in backward projection, the tracer_loop will read and update pixels along each ray.

The code of the forward and backward projection is shown in Figure 5.4 and 5.4. The difference between backward and forward projection is quite small. The code first finds out the direction for the next voxel in the ray, then it performs a MADD operation to accumulate the sinogram or update the image. Note $\lambda - \lambda_0$ is essentially the coefficients for matrix A and this coefficient is computed on-the-fly. Then it proceeds to the next point in the ray. The forward projection tries to compute a line integral, while the backward projection tries to distribute a line integral into the points on the ray. The tracing stops if the voxel hits the boundary of the object.

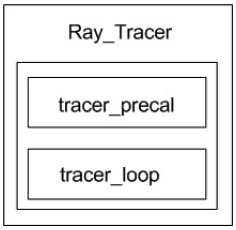


Figure 5.3: Ray tracer block diagram

5.2.3 Intersection Computation

The *tracer_precal*() function is responsible for computing the intersection point of the ray with the object and finding out the parameter required for the tracing. Given a source coordinate (s_x, s_y, s_z) and destination (d_x, d_y, d_z), the procedure finds the intersection point with the object which is a cube $0 \leq x < N_x, 0 \leq y < N_y, 0 \leq z < N_z$.

The procedure first needs to find the intersection ratio in each dimension.

$$\lambda_{xmin} = min(\frac{0 - s_x}{d_x - s_x}, \frac{N_x - 1 - s_x}{d_x - s_x}) \tag{5.4}$$

The equation above computes the x-dimension intersection ratio that is closer to the source. And similarly

$$\lambda_{xmax} = max(\frac{0 - s_x}{d_x - s_x}, \frac{N_x - 1 - s_x}{d_x - s_x}) \tag{5.5}$$

```
for all the views
for all the detectors
{
  tracer_precal(); // find initial ray parameters
  //  λx ,λy ,λz ,λ0 , vx ,vy ,vz , Lenx ,Leny ,Lenz , signx , signy , signz

  if(mode==0) tempsino=0; //forward projection
  else   value= sinogram(..); //backward projection
  for (i=0;i<Nx+Ny+Nz;i++)//(tracer_loop)
      {
      if   (λx <= λy && λx <= λz) λ = λx;
      else if (λy <= λz) λ = λy;
      else λ = λz;
       //MADD computation
      if(mode==0) // forward projection
            tempsino+ = imageData(vx, vy, vz) ∗ (λ − λ0);
      else // backward projection
            imageData(vx, vy, vz)+ = value ∗ (λ − λ0);
      λ0 = λ;

       //Find the next point on the ray
      if   (λx <= λy && λx <= λz) {λx+ = Lenx; vx+ = signx;}
      else if (λy <= λz) {λy+ = Leny; vy+ = signy;}
      else   {λz+ = Lenz; vz+ = signz;}

     //Exit conditions
      if(vx < 0||vx > Nx − 1) break;
      if(vy < 0||vy > Ny − 1) break;
      if(vz < 0||vz > Nz − 1) break;
      }
     if(mode==0) sinogram(..)=tempsino;
}
```

Figure 5.4: Ray Tracing Core Engine

The routine then finds out the min and the max of the ratios.

$$\lambda_{min} = max(\lambda_{xmin}, \lambda_{ymin}, \lambda_{zmin}) \quad (5.6)$$

$$\lambda_{max} = min(\lambda_{xmax}, \lambda_{ymax}, \lambda_{zmax}) \quad (5.7)$$

The ray intersects with the object if and only if $\lambda_{min} < \lambda_{max}$. After we are sure that the ray intersects with the object, we can then compute the near-end integer intersection coordinate (v_x, v_y, v_z) using λ_{min}. Other parameters λ_x,λ_y,λ_z,λ_0, used in the tracing loop can be derived based on the coordinate (v_x, v_y, v_z). Len_x,Len_y and Len_z are determined by the reciprocal of distance vector $(d_x - s_x, d_y - s_y, d_z - s_z)$. $sign_x$, $sign_y$, $sign_z$ are the signs of the distance vector. The actual code is lengthier because we also need to consider a set of special cases where division-by-zero occurs.

5.3 Overview of the Design

5.3.1 Ray-by-Ray Parallelism vs. Voxel-by-Voxel Parallelism

Recall that in the forward projection, we need to read the voxel values along the ray, and update (accumulate) the corresponding sinogram value based on voxel value. In the backward projection, we need to update (accumulate) the voxel values on the ray based on the sinogram value associated with the ray. The code shown in Figure 5.4 is already suggesting a ray-by-ray tracing approach.

However, we are aware of that there are two approaches to parallelizing the ray-tracing forward/backward projection. One is called the ray-by-ray approach like Figure 5.4, the other one is called the voxel-by-voxel approach. For the forward projection, a ray-by-ray approach is preferred, because the accumulation of different signogram data for each ray is independent, and we can avoid the concurrent update on the (shared) sinogram data. For the backward projection, the voxel-by-voxel approach can avoid the access conflict. However, since the forward and backward projection share a lot of similarity, we use the ray-by-ray approach to enable the sharing of the hardware.

Using the ray-by-ray approach also enables the isolation of the machine configuration and the tracing engine. There are various kinds of source/detector configurations in CT, such as fan-beam, cone-beam, parallel-beam, etc. If we use the voxel-based approach, the computation required to obtain the list of sinograms that contribute to voxel heavily depends on the machine setups. We use a ray-tracing approach that realizes ray-by-ray based parallelism. Once the set of rays is known, the hardware for tracing can be reused. Figure 5.4 depicts the code for a single tracer. Using this architecture, it is much easier to migrate from one machine setup (e.g., cone-beam) to another (e.g., fan-beam). In Section 5.3.3 we describe how we cope with access conflicts for backward projection.

5.3.2 No Cache Interleaved Access

The ray tracing procedure has a lot of random data access. Those accesses present a certain degree of reuse; however, these reuses are hard to capture if we do not use a cache-based system. Note that it is also possible to use the BRAM scratchpad to capture reuse within the application design. But that requires deep knowledge of the geometry of rays and how they intersect. On the contrary, our architecture does not make that assumption. It is tedious to implement the cache in FPGA, and it is also an active research area. Our target multi-FPGA platform, Convey HC-1/HC-1ex, features a high external memory bandwidth through interleaved data access. Each system has four user FPGAs. Each FPGA is presented with up to 16 (virtual) memory channels. Most existing FPGA computing boards prefer burst access. In the Convey system, parallel data access is not done though a burst access, but rather through interleaving. Requests in different channels can be processed in parallel if they fall into different banks. The system has 16 DIMMs and 1024 banks in total in the memory system. So the possibility of the bank conflicts is low if the parallel accesses are quite random. Such an interleaved memory design is also seen in the on-chip scratchpad memory of Nvidia GPUs. A fair amount of BRAM is already used to build up the memory interface crossbar and FIFOs. Because the bandwidth of the external memory is already quite high, we do not implement cache, but talk to memory channels directly.

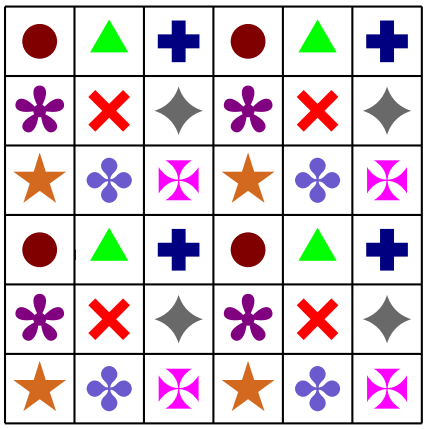


Figure 5.5: Ray-based parallel mapping

The particular interesting point of the interleaved memory system is the differentiation. For example, cache is an effective design technique, however, it is widely used in CPU, and also present in the recent Fermi GPUs. Many FPGA boards prefer continuous off-chip DIMM access (using burst transfers), but GPU also features coalesced (contiguous) access with a much higher bandwidth. The interleaved memory system shows the power of customization, and this memory system matches the need of our application quite well. Our experiment results also show that our performance can match (or beat) GPU, although the peak bandwidth of the FPGA system is still lower than the GPU.

As a side note, prior work [96] does use caching to improve the bandwidth for backward projection. And the paper [89] discusses how to obtain good memory bandwidth on an FPGA-based system that uses burst transfers.

5.3.3 Resolving Access Conflicts in Parallel Backward Tracing

The forward projection can be parallelized easily. A large number of parallel units can operate on the forward ray tracers simultaneously for different source and detector pairs. For backward projection, there are dependencies among views. Moreover, even within one view, there are conflicts when two parallel units update one pixel. To resolve the data conflicts within one view, atomic functions that guarantee the mutual exclusion of an address in

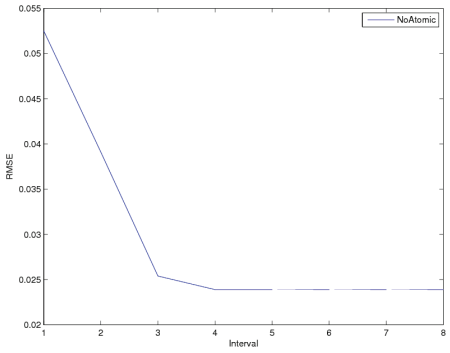


Figure 5.6: RMSE vs. Intervals

memory can be used to handle such potential data conflicts. However our target FPGA platform does not provide atomic operations on the memory system.[4] The only way to obtain a correct design is to enforce that memory requests complete in order. This has substantial overhead because the memory system is designed to be weakly ordered and supports parallel data access. We instead exploit algorithm-level changes to avoid the use of atomic operations. First, we ensure that the computation for different views (sources) are done in a sequential fashion. For a same view, the detectors that are far enough are set to one group. Mathematically there will be no conflicts within the group, and all tracers in one group can be processed in parallel. As illustrated in Figure 5.5, we can choose the tracer lines in the same picture pattern. The selection of distance between two adjacent detectors is a tradeoff of parallel granularity and algorithm performance. In our implementation, we choose the distance to be 5. The performance of different distances and final reconstructed image quality is provided in Figure 5.6. The figure shows the root mean square error (RMSE) for the results with different intervals. When the interval is 5$\sim$8, the algorithm without atomic operation can obtain the same result as that with atomic operation.

[4]It is possible to realize the atomic operation within the BRAM. The off-chip memory does not support atomic updates.

5.4 Implementation & Optimization

Our whole design is realized using the C-to-FPGA tool AutoESL from Xilinx. While we are confident that we have done substantial optimization and should match what manual RTL can do, it took us several hardware-oriented steps to achieve that.

5.4.1 Streaming Architecture

The code shown in Figure 5.4 does not look very complex.[5] If we synthesis the code directly, a few problems arise: First, the *tracer_precal* and the main tracing loop are done in a sequential way. The synthesis tool supports loop pipelining. However, that pipelining only works for the innermost loop. The support of task-level pipelining is still quite limited, and only works for certain synchronous data flow examples. In principle, this feature can be embedded in the tool further, where we can specify the loop pipeline for the inner-most loop, and task-level pipeline for the second innermost loop.

We synthesize the *tracer_precal* and the tracer loop individually to obtain their corresponding latency reports. Because the loop bound of the *tracer_loop* is not known, we use an average loop bound from the simulation of the test data to compute the average-case latency of the tracer loop. The throughput of the memory interfaces is also considered. Roughly, the latency of the *tracer_precal* is around 1/4 of the latency of the tracer loop for a 128^3 test data. Because of this, we realize two *tracer_precal* modules and eight *tracer_loop* modules in a single FPGA. Each FPGA has 16 virtual memory channels, and each tracer loop module talks to two of them (one for read and one for write). The multi-FPGA system has four user FPGAs (application engine or AE); we distribute the workload using SIMD fashion.

The diagram of our implementation in one FPGA is shown in Figure 5.7. To realize such a diagram in C level, we invoke the function *tracer_precal* twice and invoke the function of the tracer loop eight times. These different invocations take different FIFO channels and memory interfaces as parameters. The compiler can recognize that these function calls are

[5]Note we cleaned up and rewrote the code several times to be able to get to that. The original code is 10X longer with a lot of messy control and data flows.

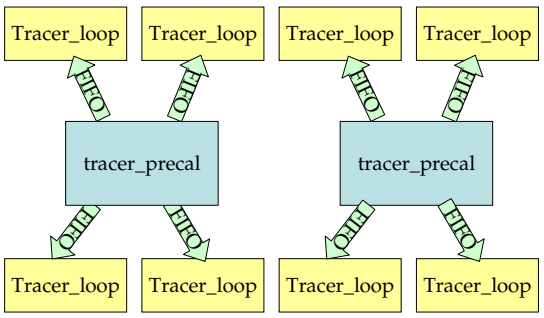


Figure 5.7: Overall streaming architecture inside one FPGA AE

independent and shall generate a parallel hardware.

The streaming FIFOs are instantiated in a top-level RTL manually, although these FIFOs are actually "internal FIFOs." Sources tell us that AutoESL tool is adding the support for internal FIFOs for C-based flow. Note that it is easy to realize these internal FIFOs using SystemC flow.

The transform that converts the code in Figure 5.4 to a C code that calls for two *tracer_precal* and eight *tracer_loop* seems counter-intuitive for software engineers. At a higher level, our manual step in this subsection can be viewed as a combination of loop unroll transform and loop splitting transform, where the split loops then take different unrolling factors. In practice, these decisions still need to be coded at a lower level.

The round-robin distribution logic is also coded in the *tracer_precal* function. At the receiver side *tracer_loop*, the control is just a simple counter to maintain the number of rays processed. Each *tracer_loop* will process a pre-determined number of rays. Note that it is possible that some rays do not intersect with the object. In this case, the *tracer_precal* will send a special flag to denote that no processing is needed, but the counter should still be updated to obtain a correct exit condition.

The intersection computation we implemented is fairly generic. Currently, the controls that set the list of sources and detectors are also coded in the function, along with the lookup tables ROM for *sin cos* functions. Note that it is very easy to change these controls to reflect another scanner machine setup. Currently, we are working to further break this module into two submodules, where one submodule simply provides the (s_x, s_y, s_z) and (d_x, d_y, d_z) in a

streaming fashion, and another submodule performs the intersection computation. We can move the first submodule to a soft processor like a microblaze (or an ARM core in Virtex-7).

5.4.2 Prefetching

The second problem is that the generic HLS tool is not strongly coupled with a particular FPGA-board or high-performance reconfigurable system. In this case, we need to code the C code in a way that can talk to the off-chip memory interfaces used in the RTL design templates. Off-chip memory access has a long latency. We are told by Convey engineers that the latency is 125 cycles at 150MHZ (target clock frequency of user FPGA application engine). And the latency can be longer if congestion (bank conflicts) occur. The memory system used in Convey HC-1 optimizes for the scatter-gather type of random access rather than burst access. AutoESL only supports burst transfers using a built-in "memcpy" function. Our application requires a large amount of random access that is not in burst mode. Without proper prefetching, latency for the individual access will kill the system performance.

In our implementation, we model each memory access port with a request FIFO and a response FIFO. We need to invoke two parallel functions inside the hierarchy of *tracer_loop*. One function is the "helper thread" *tracer_loop_addrGen* which is responsible for sending memory requests for reads; the other function is the "compute thread" *tracer_loop_compute* which obtains data from response FIFO and writes out the computed result into another request FIFO. This way, the helper threads can keep sending as many requests as possible (until the FIFO is full). Effectively, the helper thread is performing the prefetching of the required data, and the response FIFO serves as the prefetch buffer. Note that our helper thread only performs prefetching for reads. Off-chip memory writes are still performed by the compute threads. In the Convey system, memory write requests do not generate responses in the response FIFO. Figure 5.8 depicts the architecture inside the *Tracer_loop* function. Note that there is an additional FIFO to pass the data from *tracer_loop_addrGen* to *tracer_loop_compute*, because we need those parameters that are generated by the *tracer_precal* to perform the tracing.

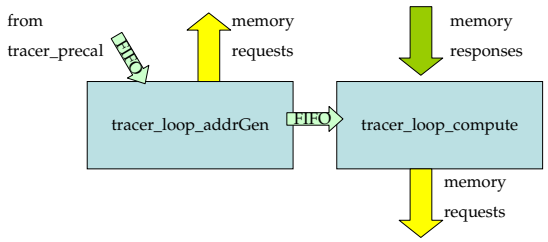


Figure 5.8: Streaming architecture inside one *Tracer_loop* kernel

The address generator (or helper thread) and the computation unit (compute thread) are realized by functions with no dependence between each other. However, the scheduler of HLS may or may not schedule them precisely at a same state. If they are not scheduled to execute at a same state, deadlock may happen. In our implementation, we use some tool-specific tricks, where we create one function hierarchy that includes the address generator and the computing functions, to ensure that the parallel functions are scheduled at a same state.

There are additional risks that may cause deadlock. When we use multiple channels in the loop pipeline, the coupling between the different channels is then introduced. If one channel gets blocked, the FSM will not switch to process the other channel, causing the FIFO in the other channel to fill up. This also can potentially introduce deadlock for a large dataset. We enlarge the FIFO size for requests or responses in the top-level RTL template to resolve that. Using a non-blocking FIFO interface can help as well.

5.4.3 Fixed-Point Conversion

The EM+TV algorithm is one iterative reconstruction algorithm that can recover object information from incomplete acquisition data. To reduce the area of our design, we convert floating-point computation into fixed-point. We use the standard range analysis technique to obtain the range of all the values in our datapath. Because the algorithm is iterative, static precision analysis would generate quite pessimistic results. We use dynamic analysis instead to determine the number of fractional bits.

We try different numbers of fractional bits and compare these with the floating-point reference code. As illustrated in Figure 5.9, the bitwidth of the fractional part will influence

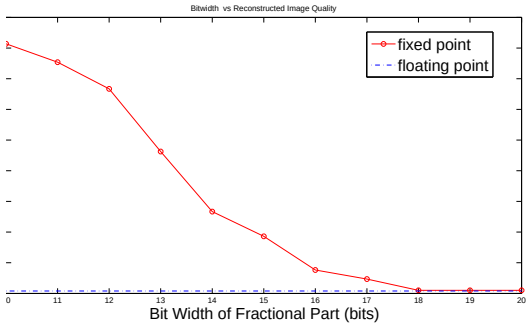


Figure 5.9: Fractional bit width and reconstruction quality

the reconstruction quality greatly. When 18 bits (10^{-5}) are used, the fixed-point version can achieve the same reconstruction quality of the floating-point version. We enlarge the bidwidth by additional 2 bits to bring in more safe margins, and use 20 bits for the fractional part. Note that it is still possible to store all those array data using 32-bit data when we use 20-bit fractional.

There are multiple multiplications and divisions in the operations; to preserve the precision of 10^{-5}, 64-bit is used for the core intermediate operations. For example, when we do the division in fixed-point, we need to first left-shift the dividend so that the quotient can still have enough fractional bits.

5.4.4 Arithmetic Specialization & Area Optimization

There are a number of division operations used in the *tracer_precal*. The divider generated by the tool consumes a large area, because the divider IP seems to support a low initiation interval (II). In our case, we do not need a high-throughput divider as the latency for the *tracer_precal* is not that critical. We instead write our own divider using a simple shift-and-subtract method. This reduces the area substantially.

Additionally, because the expression $\frac{0-s_x}{d_x-s_x}$ and $\frac{N_x-1-s_x}{d_x-s_x}$ share a common divisor, we first compute the reciprocal $\frac{2^N}{d_x-s_x}$, and then use multiplication to replace the division. We can reduce the number of divisions used by a half. We perform additional optimizations to tune the functional hierarchy to facilitate hardware sharing. Table 5.1 shows the area results

Table 5.1: Area optimization for the *tracer_precal*

	DSP48E	FF	LUT
Original	39	373146	115947
Optimized	28	9287	10831

```
if(mode==0) // forward projection
        tempsino+ = imageData(v_x, v_y, v_z) * (λ − λ_0);
else // backward projection
    if(image_denote(v_x, v_y, v_z)==1)
        imageData(v_x, v_y, v_z)+ = value * (λ − λ_0);
```

Figure 5.10: Masking for backward projection

before and after these optimizations. Note that both versions are in fixed-point already.

5.4.5 Reducing the Data Accesses via Sparsity

The final output image of the compressive sensing algorithm is sparse. Also we know that the image voxel value is non-negative. Based on these two facts, we develop a simple heuristic to reduce the amount of data access. In the beginning of the iteration, we perform a single forward projection. If any accumulated sinogram value falls below a threshold, we conclude that any image value on that ray shall be close to zero. Based on this, we build a mask of the image called *image_denote*. When we do the backward projection, we only update the voxels that are not masked. Note that this mask only needs 1-bit data, so we merge this 1-bit data into the *imageData* array. This way, we reduce the number of data accesses in the backward projection. In our test dataset, this reduces the number of external memory writes by 70%.

5.4.6 Simultaneous Reconstruction of Two Images

After fixed-point conversion, the external data accesses are all in 32-bit. The memory interface of the Convey multi-FPGA platform supports 64-bit memory interface. Because the

data access in the tracing is somewhat random, it is hard to use the 64-bit interface to enlarge the application bandwidth. However, it is straightforward to use that to reconstruct two images simultaneously—by properly pack two 32-bit data from two images into a 64-bit data. These two images need to have an exact machine setup where the *tracer_precal* part does not need to be changed.

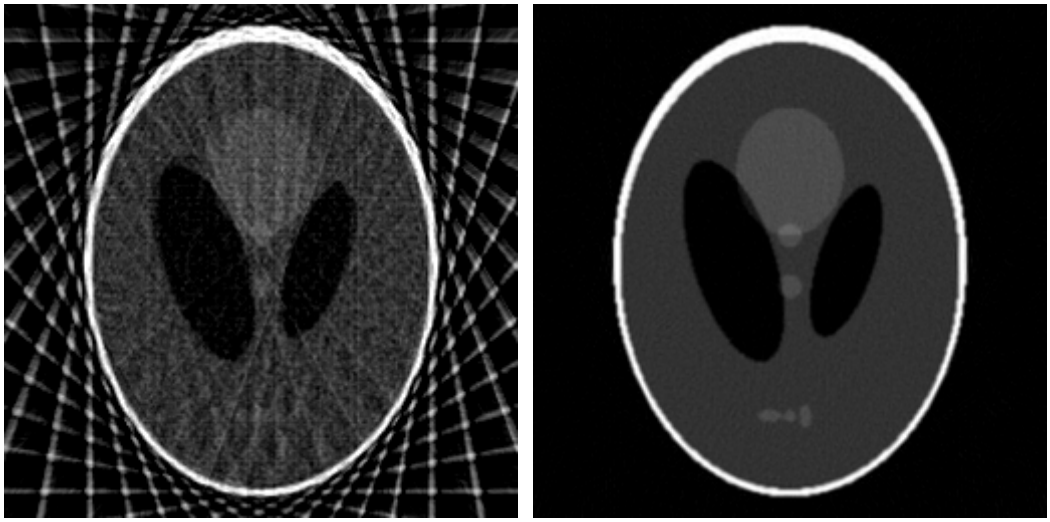
We do not increase the number of MADDs to support the 64-bit data. We measured that the external memory FIFO interfaces would return one data element in about three cycles in the average case. [6] We simply enlarge the initiation interval (II) of the tracer loop from 1 to 2 to facilitate the sharing of MADD units.

5.5 Experimental Results

Our whole design is described in C and synthesized into Vverilog RTL using the AutoESL HLS tool, version 2011.1. The target hardware platform is the Convey HC-1ex with four Virtex6 LX760 user FPGAs. We designed the RTL interfaces for the AutoESL tool to hook up with Convey's Personality Development Kit (PDK). Those interfaces are reused by a number of designs that we implemented. PDK is the RTL-based synthesis and simulation environment for the HC-1ex platform. We synthesize the RTL generated by the AutoESL HLS tool along with the PDK infrastructure RTLs using Xilinx ISE 12.4.

Our test setup assumes a Cone-Beam CT system. Currently, we tested a phantom dataset of size 128^3 which is supplied by the authors of [97]. We have 36 views (sources) and the size of detector (destinations) is 301*257. According to [97], the EM+TV algorithm using 36 samples can get a similar image quality that is obtained by using FDK/FBP algorithm that requires 360 samples. Potentially, that can reduce the radiation dose by 10X. The 2D slices of the phantom test images are shown in Figure 5.11.

[6] The peak rate is one data element in every cycle. We did not reach this high rate because our application logic is connected to a crossbar logic which performs arbitration and packet routing.



(a) FDK/FBP with 36 views

(b) FDK/FBP with 360 views

(c) EM+TV with 36 views, floating-point software reference

(d) EM+TV with 36 views, our implementation (in fixed-point)

Figure 5.11: Slices of test phantom images

5.5.1 Kernel Performance and Energy Consumption

Table 5.2 presents the performance and the energy consumption of the forward projection kernel and the backward projection kernel. The number is collected by averaging 1000 invocations. The performance on a dual-core CPU and many-core GPU is also reported. The CPU used is the Intel Xeon 5138 with 2.13GHZ clock frequency and 35W TDP. The GPU1 column denotes the Nvidia C1060 with 240 cores and 200W TDP. The GPU2 column denotes the Nvidia GTX480 with 480 cores and 250W TDP. We parallelize the CPU code using OpenMP and implement the GPU kernel using the Nvidia CUDA Toolkit 3.2. The throughput of the FPGA design is better than latency because we can reconstruct two images simultaneously. The power of the FPGA application engine is measured by the Xilinx xPower tool. We have four user FPGAs in the system. The actual system power of the Convey system is larger as the coprocessor memory, coprocessor PCB etc., also consume a lot of power.

From Table 5.2 we can see that, when the latency of forward and backward is added together, our multi-FPGA engine is about 50% faster than the CUDA implementation on the Tesla C1060, but about 2X slower than the Fermi GTX480. When we consider the fact that we can do two reconstructions simultaneously, it means our FPGA-engine is 3X faster than Telsa C1060 and on a par with the Fermi GTX480. The energy number is listed in the table as well. We can see that the FPGA platform delivers a good performance with much lower energy.

Note that it turns out that the execution time for backward projection is noticeably slower on other platforms. This is because the amount of data access is up to 2X larger (we need to first read the voxel value and then write it back). Also we need to use more invocations (and synchronization) to avoid conflicts and ensure correctness. That also reduces the available parallelism. For the FPGA design, we use the same architecture for both forward and backward. Each PE is connected to two memory channels, one for read and one for write. Thus their execution times are similar. However, in the forward projection, the memory channel is somewhat underutilized, because the number of writes is much smaller than reads.

Table 5.2: Performance and energy numbers for computing kernels for 128^3 data. Latency of FPGA (former number) is roughly 2X of the throughput (latter number) when we construct two images simultaneously.

	Power	Forward Projection		Backward Projection		Forward+Backward	
		Latency/Throughput(s)	Energy(J)	Latency/Throughput(s)	Energy(J)	Latency/Throughput(s)	Energy(J)
CPU	35W	1.94	67.9	3.23	113.0	5.17	180.9
FPGA	94W	0.305/0.153	28.7/14.4	0.308/0.154	29.0/14.5	0.613/0.307	57.7/28.9
GPU_1	200W	0.342	68.4	0.668	133.6	1.01	202
GPU_2	250W	0.085	21.3	0.276	69	0.361	90.3

Table 5.3: Area results

	BRAM	DSP	LUT	FF	Slice
Consumed	79	68	113,355	104,099	36511
Total Available	720	864	474,240	948,480	118,560
Utilization	11%	7%	23%	10%	30%

Table 5.4: Application performance and energy consumption

	Throughput(s)	Energy(J)
CPU	1189	41.6E3
Hybrid	92.0	12.7E3
GPU_1	361	72.2E3
GPU_2	114	28.5E3

Potentially, the forward projection can be made 2X faster if we separate the design for forward and backward.

Another interesting observation is that the Fermi GPU GTX480 is between 3 to 4X faster than the Telsa C1060. The number of cores is 2X that of C1060, and the peak off-chip bandwidth is about 1.6X (from 100GB/s to 160GB/s). So it is safe to say that there is an additional 2X performance benefit attributed to its cache systems. Our current FPGA design does not have a cache, but it is indeed worthwhile to investigate that possibility given the performance benefit we see from the GPU.

The area results for the complete design are listed in Table 5.3. Note that our core computing RTL consumes fewer logic slices, because the PDK infrastructure also consumes about 10% to 15% area. Most of the BRAM utilization is due to the PDK infrastructure.

5.5.2 Application Performance and Energy Consumption

We then test the application performance of the EM+TV algorithm on a hybrid configuration where the EM part is done by the FPGA-subsystem and the TV part is done by the GPU. The flowchart of the application is shown in Figure 5.2, where the outer EM+TV iterates

100 times, and the inner EM step iterates 3 times. The Convey system provides PCI-e X16 interfaces in their HC-1ex platform. However, there is not enough physical space for the GPU device. We instead use one external PCI-e enclosure to hook up the GPU.

Our hybrid configuration connects the Fermi GTX480 onto the Convey HC1-ex platform. After one EM iteration completes, the image data is copied into the GPU memory space and the TV CUDA kernel starts. The data transfer would not add substantial overhead in this case. We measured that a pipelined data transfer (FPGA coprocessor-side memory to PCI-e) can reach close to 1GB/s. Each EM iteration only needs to copy 128^3 or 8MB of image data to the GPU. And similarly, we need to do the transfer backwards when one TV invocation finishes. That only adds about 0.016s for each EM+TV iteration, or about 2s for the whole EM+TV application. Because the TV kernel is a highly regular stencil computation, we believe the GPU is a good device for that application kernel. The execution time of the TV is much shorter than EM. In the energy calculation for the hybrid configuration, we assume that the GPU can be powered off when it is not actively running CUDA applications. In practice, a 10% to 15% idle power would exist. The numbers for application performance/throughput and estimated energy consumption are shown in Table 5.4.

5.6 Conclusions and Future Work

In this chapter we share our experience of using the AutoESL HLS tool to map one compressive sensing iterative reconstruction algorithm on an FPGA-based reconfigurable computer. Our hybrid approach provides good performance and potential energy savings.

Currently, we are working to separate the intersection computation into two small submodules so that it can be easier to switch to a different setup. We are also investigating different algorithmic or architectural approaches that can improve the data locality/reuse for our applications. We are also in the process of testing the algorithm and our implementation on clinical patient data.

CHAPTER 6

Architecture Templates for Coprocessor Acceleration

In this section, we study several architecture templates which assume that the component-specific implementations are ready. The first one uses a pure software-based approach and is better suited for coarse-grain tasks. The second one focuses on fine-grain tasks and performs management entirely in hardware. The third one focuses on system-level sharing which uses device drivers and a dedicated embedded controller to do sharing across multiple processes.

6.1 Collaborative Execution on the Heterogeneous Platform

One of the goals of the CDSC project [3] is to raise the level of abstraction to develop high-level parallel programming models and runtimes that are available to domain experts who are not at the same time experts in parallelism. Frameworks such as Map-Reduce [30] successfully exploit implicit parallelism on distributed systems and have also been extended to heterogeneous platforms such as the GPU [31] and FPGA [32]; unfortunately these have a restricted programming model. Dryad [35] is a research project at Microsoft Research for a general-purpose runtime data-parallel applications. However, their focus is on the multi-core cluster platform and does not provide sufficient heterogeneity support. In this section, we provide an example in the medical imaging domain that uses the CnC-HC toolflow to enable collaborative execution on the heterogeneous platform. We provide a short overview of the CnC-HC toolflow in Chapter 2 while more details can be found in [41].

Table 6.1: Performance of the applications on CPUs, GPUs and FPGAs

	Denoise	Registration	Segmentation
Num. of Iterations	3	100	50
CPU	3.3s	457.8s	36.76s
GPU	0.085s(38.3X)	20.26s(22.6X)	1.263s (29.1X)
FPGAs	0.190s(17.2X)	17.52s(26.1X)	4.173s(8.8X)

6.1.1 Benefit of Heterogeneous Computing

As we demonstrated in previous chapters, using accelerators can significantly speed up these computational-intensive applications like medical imaging [98]. Table 6.1 shows the performance of the different application steps on CPUs, GPUs and FPGAs. Note that the time values measured are for computation kernels and exclude file IOs (around 2s overhead for each invocation).

From the table, we can see that the GPU and FPGA deliver decent speedup compared to the single-threaded CPU implementation. We also notice that different applications prefer different accelerators. For *registration* application, the FPGA delivers better speedup than the GPU, and for *segmentation*, the GPU delivers better speedup.

Note that while it is possible to run an FPGA kernel on multiple applications, in practice that involves a large reconfiguration overhead. We configure the FPGA in the system to accelerate *registration* application only.

6.1.2 Image Pipeline Example

We further show how the CnC-HC toolflow described in Chapter 2 can be used to aid heterogeneous computing. Suppose we have a medical imaging pipeline, which consists of image denoising, image registration and image segmentation, We want to make good use of the heterogeneity to achieve good performance. We first construct the task-level description CnC model of the application. CnC is a task-level dataflow model, and a complete description of the model can be seen in [5].

```
< int [1] denoise_tag > ;
< int [1] reg_tag > ;
< int [1] seg_tag > ;

[ float* denoise_output ] ;
[ float* registration_output ] ;
[ float* final_output ] ;

<denoise_tag >:: (denoise@CPU = 2,GPU=1);
<reg_tag> :: (registration@GPU = 1, FPGA = 2);
<seg_tag> :: (segmentation@GPU = 1);

( denoise : k ) -> [ denoise_output : k ];
[denoise_output : k]-> ( registration: k )
                    -> [ registration_output : k ];
[registration_output : k] -> (segmentation : k)
                          ->[ final_output : k ];
```

In the description shown above, the affinity like $CPU = 2, GPU = 1$ describes the preference of the task bindings, where a larger number denotes that the task is preferable to run on that component.

Optionally, we may specify the control or data that is generated by the environment (the main thread). For example, we may add

```
env -> <denoise_tag : {0 .. 9} >;
env -> <reg_tag : {0 .. 9} >;
env -> <seg_tag : {0 .. 9} >;
```

onto the CnC file to describe that we want to create an application that performs batch processing of the image pipeline which processes 10 images. 0..9 are the *ranges* of the control tags to prescribe the computation steps.

In this CnC file, we describe the list of computation tasks *denoise*, *registration* and *segmentation*. We also describe the input and output dependencies of each task. After that,

we call the CnC translator to convert that description into a collection of Habanero C (HC) files. Users can further edit those files to create a working implementation. For example, for the above-mentioned CnC file, the auto-generated skeleton for *registration.hc* is

```
#include "Common.h"
void registration( int k, float* denoise_output0, \
Context* context){
   /*
   float* registration_output1;
   // allocate memory if necessary and fill
   //in values to put here

   char* tagregistration_output1 = createTag(1, k);
   Put(registration_output1, \
           tagregistration_output1, \
    context->registration_output);
  */
}
```

Basically, the auto-generated code provides hints for the actual implementation.

The edited code is also presented here:

```
#include "Common.h"
void registration( int k, float* denoise_output0, \
Context* context){

    float* registration_output1;
    if(current_place() == MEM_PLACE)
    {
        registration_output1=REG_cpu(k,denoise_output0);
    }
    else if(current_place() == NVGPU_PLACE)
    {
        registration_output1=REG_gpu(k,denoise_output0);
    }
    else if(current_place() == FPGA_PLACE)
```

```
    {
        registration_output1=REG_fpga(k,denoise_output0);
    }
    char* tagregistration_output1 = createTag(1, k);
    Put(registration_output1, tagregistration_output1,\
            context->registration_output);
}
```

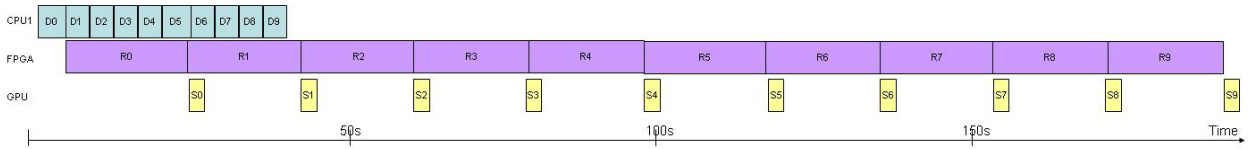
From the code above, we can see that a function *current_place()* is used to obtain the current place of the task. Then based on the type of coprocessor, we can call different routines. *MEM_PLACE* denotes a CPU worker/place, *NVGPU_PLACE* denotes a GPU worker/place and *FPGA_PLACE* denotes an FPGA worker/place. The scheduling and dependency checking is auto-generated where users do not need to edit.

In the *main* function of the program, we simply initialize the required data structure (CnC graph), and prescribe (invoke) the computation *steps*. In this example, our application wants to perform a batch processing on ten images (with a same size), where we prescribe ten instances of *denoise*, *registration* and *segmentation*.

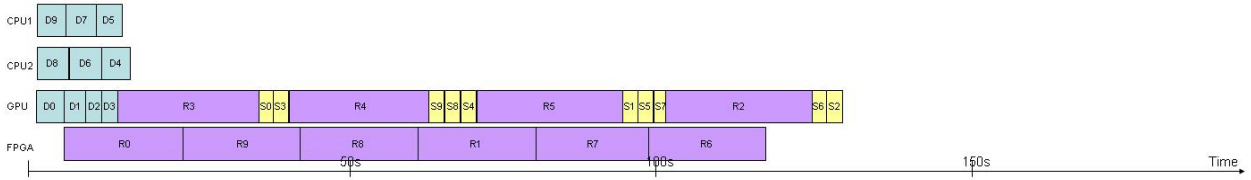
6.1.3 Benefit of Dynamic Work Stealing Across Heterogeneous Components

Without using the CnC-HC framework, we may simply construct a static mapping, like using CPU for *denoise*, FPGA for *registration* and GPU for *segmentation*. Such a mapping is a greedy approach which uses a component that is best suitable for the application. However, it does not keep track of resource availability. In the simple image pipeline, the execution time of *segmentation* is smaller than *registration*, and if we use that static mapping, the GPU would be left idle for a large proportion of time. Figure 6.1(a) shows the scheduling graph of the static scheme. Using a dynamic stealing can achieve a better load-balancing. (Note Figure 6.1(b) has 2 CPU rows but Figure 6.1(a) uses only 1 CPU to emulate a static binding case.)

Table 6.2 shows the performance results for different hardware configurations: CPU only, GPU only, CPU+GPU, and CPU+GPU+FPGA (static or dynamic bindings).



(a) Static Binding (with 1 CPU worker)



(b) Dynamic Binding (with 2 CPU workers)

Figure 6.1: CPU+GPU+FPGA schedule graph

Table 6.2: Dynamic work stealing

	Exec time	Active Energy
CPU only	3493s	69.8KJ
GPU only	276s	54.8KJ
CPU+GPU	251s	49.4KJ
CPU+GPU+FPGA (dynamic biding)	129s	36.1KJ
CPU+GPU+FPGA (static biding)	193s	23.0KJ

We can see with the cross-device stealing, a setup of CPU+GPU+FPGA with dynamic binding (Figure 6.1(b)) can get better performance than the static scheme. The energy column is computed by summing up the energy spent by each device that contributes to the computation. (We assume CPU is at 10W per worker(core), GPU at 200W and FPGA at 94W, idle power is ignored so far.) The static binding does have a lower energy number in the table. However, when the idle power and the power of other system components are considered, we may prefer the dynamic binding which obtains the results in a shorter period of time. Still, the data in the table shows that different scheduling policies may be needed to optimize the energy consumption or the overall performance.

Note that in Figure 6.1, *Dk* means *denoise* instance k, *Rk* means *registraion* instance k and *Sk* means *segmentation* instance k. We can see that the stealing happens by analyzing the schedule graph Figure 6.1(b). Initially 10 tasks of *denoise*($D0$ to $D9$) are pushed into the queue of the GPU, while the $D4$ to $D9$ are actually stolen by the CPU. Similarly, the *registration* tasks are pushed into the queue of the FPGA initially, but several task instances are stolen by the GPU as well.

We want to point out that, all results shown in Table 6.2 can be achieved by simply modifying the affinities of the CnC description. For example, a static binding can be realized by

```
<denoise_tag>:: (denoise@CPU = 1);
<reg_tag> :: (registration@FPGA = 1);
<seg_tag> :: (segmentation@GPU = 1);
```

A hardware configuration with only 1 CPU worker is used to achieve the effect shown in Figure 6.1(a), otherwise the *denoise* tasks shall still be spread across 2 CPU workers.

We just need to rerun the CnC translator and then recompile the program, and the one

```
<denoise_tag>:: (denoise@CPU = 2,GPU=1);
<reg_tag> :: (registration@GPU = 1, FPGA = 2);
<seg_tag> :: (segmentation@GPU = 1);
```

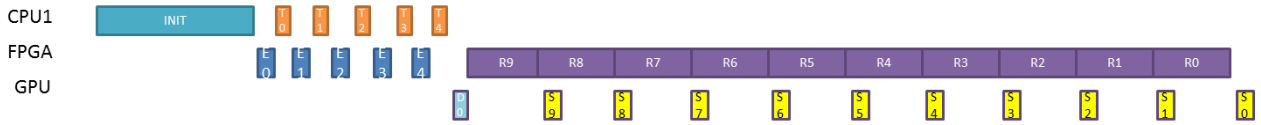
is used to realize the dynamic scheme. CPU-only, GPU-only and CPU+GPU can be constructed in a similar fashion.

While the work-stealing runtime is quite powerful, the decision it makes are simply based on the status of the queues. In the above CnC fragments for the dynamic binding, if we use

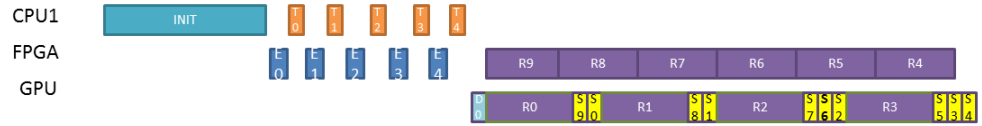
```
<reg_tag> :: (registration@CPU=1,GPU = 2, FPGA = 3);
```

and then the CPU shall also help to perform registration application, and will drag down the total performance of the batch processing. Nonetheless, the affinity provides a great tuning capability to boost the overall application performance.

We also constructed another pipeline by adding a few iterations of compressive sensing reconstruction (EMTV) before the pipeline shown above. More specifically, we first perform *reconstruction* and *denoise*, then we register the denoised image with multiple images in the database and then perform *segmentation*. Figure 6.2 shows the execution states for this pipeline. Ek means the EM instance k, Tk means TV instance k, Dk means *denoise* instance k, Rk means *registraion* instance k and Sk means *segmentation* instance k. While the graph is quite similar to the one for the pipeline Figure 6.1, we see that the task dependencies in the beginning of the pipeline (EM+TV iterations and *denoise*) limit the freedom to explore task-level parallelism. When the *denoise* finishes, 10 instances of *registration* and *segmentation* allow work-stealing across multiple devices.



(a) Static Binding



(b) Dynamic Binding

Figure 6.2: CPU+GPU+FPGA schedule graph for a revised pipeline

6.2 Scheduling Dynamic Loops

The CnC-HC flow uses a pure software-based runtime system to perform load-balancing and scheduling. For fine-grained tasks (for example, a task is computing a sparse dot product in a sparse matrix vector multiplication), the overhead of software-managed scheme may become large. This motivates several studies on architecture support for flexible scheduling such as Carbon [99] and ADM [100]. We study the hardware-based scheduler in the context of the FPGA-based accelerator for *Dynamic Loops*. We discover that, in the settings of dynamic scheduling, the effective bandwidth for data access is lower due to potential access conflicts.

The FPGA is frequently used as a loop accelerator. The primary reason is that these loops typically have a large degree of parallelism where multiple instances of the loop body can execute simultaneously.

A loop typically has an iteration space. Some loops have fixed iteration spaces where the loop bounds are known in advance. Many loop optimization or transform techniques such as iteration space tiling/slicing [101] and affine transforms [102] can be easily applied. When the iteration space is not a fixed one, many of these techniques do not directly apply. We denote these loops as *dynamic nested loops*.

Actually, these dynamic loops are frequently encountered in practice. For example, in sparse matrix vector multiplication, the dynamic behavior occurs because of the fact that the number of non-zeros in each row/column is not the same. In hypergraph traversal, the number of pins for each node or each hyperedge is different. Recently, there is a growing interest in porting map-reduce framework onto FPGAs [103, 104]. This can also be viewed as a case for dynamic loops. Multiple map (or reduce) instances can run in parallel, and there is no guarantee that these instances shall finish in a constant cycle count.

Let's look at a typical dynamic loop: sparse matrix vector multiplication (code shown in Figure. 6.3). The sparse matrix here is represented in a compressed row format. Clearly, the inner loop bound is not a constant value. Multiple inner loop instances can execute in parallel.

Each execution of the inner loop is roughly proportional to the loop bound of the inner

```
for(i=0;i<n;i++){
  temp=0;
  for (k=rows[i];k<rows[i+1];k++)
        temp+=val[k]*x[col[k]];
  y[i]=temp;
}
```

Figure 6.3: Sparse matrix vector multiplication

loop (and they differ row by row). Thus, parallelizing the outer loop in a naive way may cause load balancing issues. We can also leverage fine-grain parallelism to parallelize the inner loop using a tree-adder type of architecture to do the reduction.

For the sake of simplicity, we assume we are dealing with a two-level nested loop similar to Figure. 6.3. The loop bound of the inner loop varies between instances. There are no write-conflicts or any dependencies between multiple inner loop instances, and their executions can be arbitrary ordered. These conditions can be checked easily using standard compiler techniques.

There are mainly three ways to implement the hardware architecture for these dynamic nested loops: approach A: parallelizing the inner loops (the loops with dynamic bounds); approach B: parallelizing the outer loops with static allocation/scheduling; and approach C: parallelizing outer loop with dynamic allocation/scheduling. Here we mainly want to evaluate three architecture templates and discuss their trade-offs. We find that static allocation, with various kinds of static allocation strategy, is often the best architecture among the three. It is simple to implement and is very effective. The dynamic allocation approach, although attractive, faces many practical difficulties or overheads. These overheads include data access conflicts, serialization in the centralized scheduler and queue structures etc. Efforts to reduce the overhead are also presented.

We use SPMxV as a representative example to make this comparative study, but the discussion presented is broadly applicable to many other dynamic nested loops. SPMxV is the key computation kernel in a wide range of applications (e.g., quadratic placement

in EDA domain, level-set solver for medical image segmentation etc.) We also need to point out that FPGA-based implementations for SPMxV have been heavily studied by the FCCM community in the past. A tree-adder based approach is implemented in [105] and [106]. The work in [107] parallelizes multiple sparse dot products and also discusses load-balancing improvement through graph partitioning. The work in [108] partitions the matrix into multiple strips and implements a streaming approach.

6.3 Architecture Templates and Implementations for Dynamic Loops

This section presents the three architecture templates in the general sense, and illustrates their implementations on the SPMxV example. We assume the computation is done in fixed point, and all the data required is already stored on-chip.

6.3.1 Parallelizing Inner Loops

Because the loop bound of the inner loop is not fixed, complete loop unroll is not possible. However, partial unroll of the inner loop is possible. For SPMxV, the result of the partial unroll can be viewed as an architecture which uses a tree-adder as the atomic unit. The tree adder has a fixed number of multipliers in the leaf nodes, and a fixed number of adders in the non-leaf nodes.

In most cases, the latency of this atomic unit is larger than one, and the unit should be pipelined if possible.

The loop bound of the inner loop may not be a multiple of the partial unroll factor. To cope with this, dummy data (zero) should be added into the input of the atomic unit. Clearly, this brings in the architecture overhead, because the relative resource utilization rate is lower if dummies are added.

Each architecture template requires a specific memory arrangement scheme to achieve efficiency. Because the inner loop is partially unrolled, this unrolling increases the number

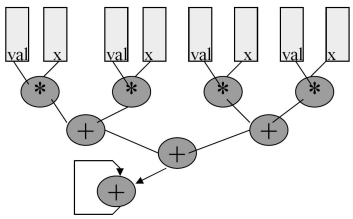


Figure 6.4: Tree adder with banked storage

of memory accesses of the inner-loop body. We can use a cyclic memory partitioning scheme so that multiple data accesses (leaf nodes of the tree-adder) always access data in different banks. Figure 6.4 shows the tree adder with one final accumulator. *val* (and *col*, but not shown in the figure) are banked using cyclic partitioning, while vector x is duplicated because it uses indirect data access.

The inner loop of SPMxV can be parallelized. However, the inner loops of some applications have certain loop-carried dependencies that enforce strictly sequential executions. For example, in random walk simulation, different walks of one trajectory seem to be strictly sequential. The architecture templates that leverage coarse-grain parallelism should be used instead.

6.3.2 Multi-PE Realization with Static Allocation

As an alternative, we can unroll the outer loop and parallelize multiple inner-loop instances (multiple dot products for SPMxV).

In this architecture, the atomic processing unit is the hardware realization of one inner-loop instance. For SPMxV, this is one MAC (Multiply-And-aCcumulate) unit with some control logic for boundary checking. The parallelism comes from instantiating multiple processing units, and each unit is handling different inner loop instances.

The allocation of the inner-loop instances into the processing elements affects the overall load-balancing. This fact is also discussed in the SPMxV design in [107], but their goal is to mainly reduce communication messages rather than help load-balancing.

The optimal allocation that minimizes the makespan of the parallel execution is NP-

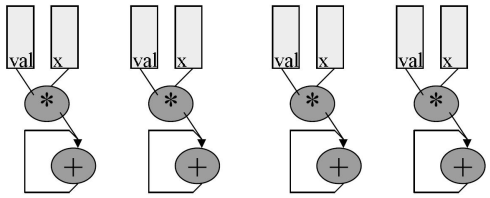


Figure 6.5: Multiple PEs with local storage

complete. Four static allocation technique are compared: cyclic (deterministic), random permutation, random and quasi-random. Note that these four techniques are general, and they do not make use of the loop bound of inner loops explicitly. More efficient static allocation can be done based on the analysis on those loop bounds.[7] Cyclic assignment determines the assignment using equation $P(i) = i\%N_{PE}$, where N_{PE} is the number of PEs, i is the subtask ID and $P(i)$ is the assigned PE ID (from 0 to $N_{PE} - 1$) for subtask i. Random permutation generates one random permutation for each group of contiguous N_{PE} subtasks and uses the permutation to do the assignment. Random assignment or quasi-random assignment determine the assignment using equation $P(i) = Rand()\%N_{PE}$ or $P(i) = Qrand()\%N_{PE}$, respectively.

Each PE has its own local memory, and it works on these local memories independently. The data required for the computation is distributed using the allocation strategy. Figure 6.5 shows the diagram of this architecture template for the SPMxV case.

6.3.3 Multi-PE Realization with Dynamic Allocation

The subtasks that are mapped into the PEs can also be allocated dynamically at runtime. To support the dynamic allocation, we conceptually need a queue of idling PEs and a queue of pending subtasks. When a PE completes the execution of one subtask, its ID shall be added into the idling PE queue. A dynamic scheduler checks the status of these two queues and maps one pending subtask into one idling PE. This is the architecture for the dynamic allocation/scheduling implemented in [104].

[7] By explicitly looking at the values of the inner-loop bounds, better load-balancing results can be obtained. An extreme case is to solve the NP-complete optimal allocation off-line and then use the results to do mapping. Our static strategies in this section do not perform any preprocessing of the inner-loop bounds.

A queue is typically implemented through FIFO. The elements in the idle queue are the IDs of PEs. FIFO can accept at most one input each cycle. When multiple PEs complete their subtasks in the same cycle, some arbitration logic (e.g., priority encoder) is needed and the IDs shall be added into the FIFO one by one. And a few more PEs may complete their subtasks in subsequent cycles, which further complicates the process.

Our implementation does not use any queue explicitly. Instead, the scheduler checks a group of (say K) idle signals from K contiguous PEs every clock cycle, and assigns some new subtasks for the idle PEs among the K PEs. When K is smaller than number of PEs N_{PE}, the scheduler checks the next K contiguous PEs in the next cycle in a circular fashion. The circuity for assigning the subtasks for K contiguous PEs is a simple bit-counting logic. Let us denote b_0 to b_{K-1} as the idle signals for K contiguous PEs. The assignment can be known after $p_i = \sum_{j=0}^{i} b_j$ is computed. If $b_i = 1$, PE i shall get a subtask with ID $M + p_i$, where M is the last subtask ID that is assigned before this cycle.

The data required for the computation needs to be accessible by all the processing elements. If these data are in one big global buffer or in off-chip memory, the bandwidth may be a severe problem since it needs to serve all the PEs. We also statically partition the required data into N_{PE} banks to ensure that the available bandwidth in the dynamic allocation case is at least the same as the static allocation case. The banking is based on a cyclic partitioning scheme.

Currently, the interconnect structure between the banked on-chip storage and the PEs is implemented as a full crossbar, where any PE can access any bank immediately if that bank is not accessed by others. Other interconnect structures such as ring can be implemented to further reduce interconnect complexity.

Dynamic allocation determines which subtask each PE shall process at runtime. When one PE is assigned one subtask, it requests the required data from the banked global buffer (or even off-chip memory) and starts computation. This data access or data copy is on the critical path and can not be overlapped, because we can not predict what data is accessed by specific PEs until run-time, thus we can not use prefetching to hide data access latency. Because

Table 6.3: Cycle counts for test matrices

	Lower bound	Approach A	Approach B				Approach C			
	a)	b)	c)	d)	e)	f)	g)	h)	i)	j)
1138_bus	254	1139	283	294	312	317	255	1140	392	283
494_bus	105	494	117	120	135	117	106	496	165	114
662_bus	155	662	178	172	202	175	158	668	228	172
685_bus	204	685	223	224	257	249	206	686	298	218
abb313	98	313	103	103	121	139	100	318	134	101
arc130	65	149	99	93	120	120	67	159	100	99
ash219	28	219	28	28	40	36	28	220	28	28
ash292	138	292	158	150	211	180	141	296	197	148
ash331	42	331	42	42	66	56	42	332	42	42
ash608	76	608	76	76	96	86	76	609	76	76

of this, paper [104] only did the double buffering for the common data path (broadcasted data).

There are two ways to resolve the issue. First, we can force the architecture to use double buffers at local PEs to do the latency hiding. Each PE shall have two (or more) slots of local storage. The dynamic allocation and scheduling determines the assignment and copies data into the slot that is recently used, but each PE still has another slot to process when the data copy occurs. Note that the solution of this dynamic assignment is different then the original one. This approach tries to overlap data access and computation in a coarse-grain fashion. This idea is inspired by the virtual function unit in the architecture of a super-scalar processor. Second, the computation can be overlapped with data access in a fine-grain fashion. For example, the computation of the sparse dot product is done in parallel when we fetch/access the required data (in a streaming fashion). We use the second approach because our data is already on-chip. However, the first approach is still useful for other dynamic loops if the fine-grain approach fails to overlap the data access and computation completely (for example, the data access needs to be done in a burst mode for off-chip access).

It is possible that one PE will try to access one data bank that is occupied by another PE in the dynamic allocation case. To reduce the overhead due to bank conflicts, we also try to use static allocation at the beginning of the computation, and then switch to a dynamic approach (this is essentially a type of work stealing) when the whole execution is about to finish.

6.4 Results with Different Templates

We implemented these architectures using VHDL and simulated them using ModelSim. Because we use fixed-point computation, we do not need to consider the extra complication due to the long latency of floating-point units. Floating-point SPMxV needs to use one MAC unit to compute multiple dot products in an time-multiplexed interleaved fashion to compensate for the long latency of floating-point units [107]. We assume a same clock frequency

although our preliminary synthesis results suggest that the design with dynamic allocation gets a worse timing. We use 16 multipliers and 16 adders in all the designs ($N_{PE} = 16$). For Approach A, this is translated into an adder-tree with 16 leaf nodes. For Aapproach B and C, this is translated into a design with 16 MAC units.

We tested the first ten test matrices from the UFL sparse matrix collection [109]. The lower-bound of the cycle count using N_{PE} multipliers and N_{PE} adders to compute a SPMxV with a sparse matrix with nnz entries is $\lceil nnz/N_{PE} \rceil$. The values are shown in column a) of Table 6.3.

The cycle count data for Approach A is shown in column b). Because the number of non-zeros in one row is typically small (average about 3 to 4), the overhead due to the dummies added is very large. However, this approach can be an effective solution if the inner-loop bound is much larger than N_{PE}.

The total cycle count by Approach B depends on the static allocation strategy. The results of cyclic allocation, random permutation allocation, random allocation and quasi-random allocation are shown in columns c) to f). All four approaches are not far away from the lower bound (likely due to the central limit theorem). The cyclic or random permutation approaches are almost always better than random or quasi-random approaches, because the latter two can not guarantee that the number of instances allocated in different PEs is more or less the same.

The cycle count for Approach C with zero-cycle overhead is shown in column g). Zero-cycle overhead means that the cycle-overheads in data transfer and arbitration are all ignored, and each PE can always get a new subtask (a new row) in zero-cycle latency. The values in this column are very close to the lower bound $\lceil nnz/N_{PE} \rceil$, which suggests that dynamic allocation indeed improves the load-balancing considerably. But we can not get to zero-cycle overhead in the practical settings. The actual cycle counts for $K = 1$ and $K = 16$ are shown in columns h) and i). We can see that the cycle count for $K = 1$ is very bad. The average cycle count for one subtask (one dot product) is 3 to 4 cycles, but we may need to wait up to 15 cycles to get a new subtask assigned in the $K = 1$ case. The $K = 16$ case is much

better because it can assign a new subtask immediately if any PE signals the scheduler that it is idle. However, due to banking conflicts, the cycle count is still worse than the static allocation approach for the majority of examples.

The last method makes the static assignment initially (cyclic allocation), but leaves the last $NumRows\%N_{PE}$ rows unassigned and makes decisions at runtime based on the progress of the computation. This is a simple heuristic that combines the benefits of dynamic and static allocation. The data is shown in column j). Further it reduces the cycle count as compared to the cyclic allocation. However, it still is some distance away from the lower bound. Other ways of combination, such as the guided scheduling approach which gradually decreases the chunk size of allocation in runtime, can also be implemented.

Note that we assume the data are already stored on-chip. This seems a little bit unfair and makes dynamic allocation unnecessary. But this is a valid assumption if we use SPMxV for iterative methods such as conjugate gradient methods. Multiple SPMxV shall be invoked in a sequential fashion. Most of our discussions are still valid if the data we need to fetch resides off-chip. For example, off-chip memory may have multiple memory modules and banks. The bandwidth shall be higher if we could avoid banking conflicts through static allocation. If we only have one bank of off-chip memory, the differences between approaches may become marginal, and the off-chip bandwidth becomes the bottle-neck for this application.

To summarize, the overhead of Approach A is the resource underutilization due to dummies. It should work better if the inner-loop bound is significantly larger than N_{PE}. The overhead of Approach B is load imbalance, especially if we can not predict the workloads of individual inner-loop instances in advance. The overhead of Approach C mainly comes from bank conflicts and serialization of the centralized scheduler. Although we assume the data is on-chip, similar argument holds if the data is placed off-chip with multi-banked memory controllers. Because the job allocation is performed dynamically at runtime, the data distribution can not be done statically and shall also be performed at runtime. It is important that we are aware of these different architecture templates and select a best one that matches a particular application.

6.5 Architecture Template for Accelerator Management

In the previous a few sections, we presented the on-chip scheduling for a type of fine-grained work-load called *dynamic loops*. Yet the discussion focused on an FPGA-centric system, and no CPU or any software is involved in the architecture template. On the other extreme side, the CnC-HC flow presented in the beginning of this chapter uses a pure software scheme to perform the accelerator management.

In this section, we want to combine the two extreme points to provide the system-level scheme for accelerator management. This also serves as the proof-of-concept prototyping of accelerator-rich CMPs called ARC [110]. The overall architecture of ARC is composed of cores, accelerators, the Global Accelerator Manager (GAM), shared L2 cache banks and shared NoC routers between multiple accelerators. These components are further connected by the NoC. Accelerator nodes include a dedicated DMA-controller (DMA-C) and scratch-pad memory (SPM) for local storage and a small translation look-aside buffer (TLB) for virtual to physical address translation. GAM is introduced to handle accelerator sharing and arbitration. The results of ARC are obtained through architecture simulators. We want to gradually realize those architecture supports through FPGA prototyping. Note that the template implemented in this section presents a preliminary one that only realizes a minimal feature of ARC. In our implementation, components are connected through crossbar or bus rather than NoC. Off-chip memory are shared, but cache banks are not shared.

6.5.1 System-level Diagram

Figure 6.6 shows the system-level diagram for our proposed architecture template. One microblaze processor is configured with cache and MMU, and we run a Linux distribution on the processor. The second microblaze processor is configured with no cache or MMU and we run bare-metal control code on that. Different components communicate through AXI buses. Two buses are instantiated on the system: one data bus is configured in the crossbar mode, while the other one is a shared bus. We also added dedicated point-to-point FIFO channels to communicate between two microblaze processors. DDR memory is connected

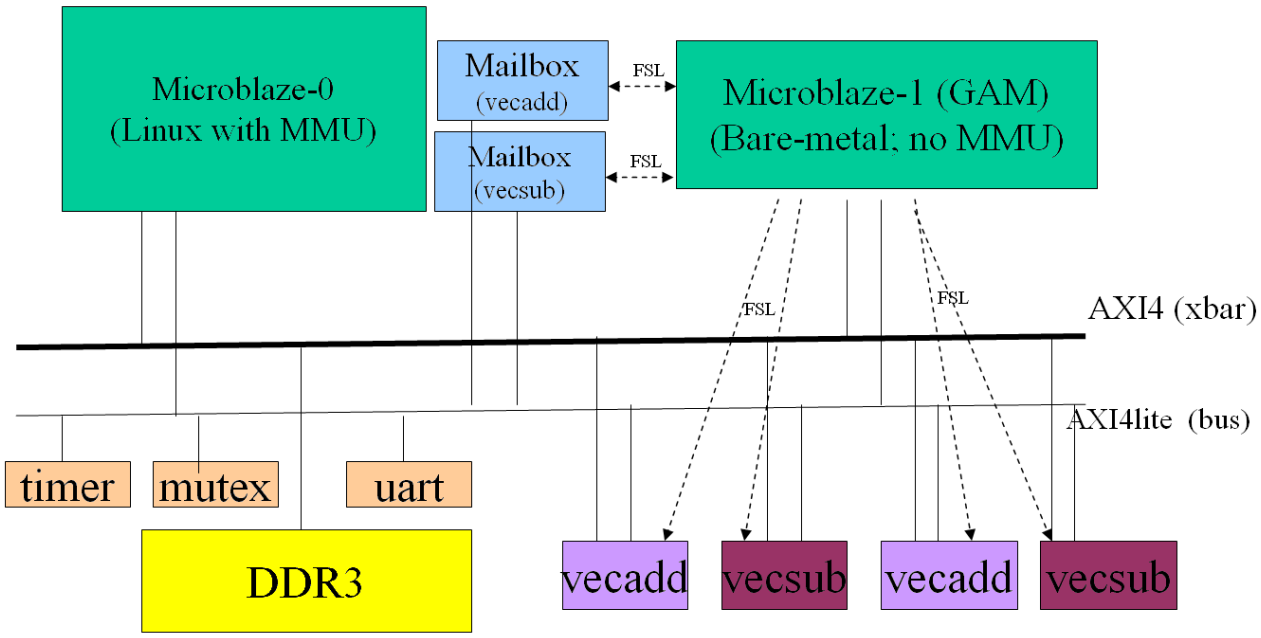


Figure 6.6: System-level prototyping diagram on Xilinx ML605 board

to AXI bus and is physically shared across different components. The MMU is present on the first microblaze, and the applications on that microblaze operate using virtual memory addressing. Other system components operate using physical memory addressing.

The system has four major components to be able to function. The first one is the application code (at user-space). The second one is the kernel-space device driver which can perform virtual-to- physical conversion and native access of hardware resources. The third one is the hardware-based on-chip accelerator manager called Global Accelerator Manager(GAM). The fourth one is the actual accelerator components.

6.5.2 Application Code

In the application code, the application passes the type of accelerator it wants to use, the virtual address of the array, the length of the array and other task-specific parameters into the device driver. A sample code can look like the following listing:

```
int* ptr=malloc(ARR_SIZE*sizeof(int)) ;
int fid=open("/dev/GAM",O_RDWR);
```

```
int i;
for(i=0;i<ARR_SIZE;i++)
        ptr[i]=INIT_DATA;
ioctl(fid,VEC_ADD_TYPE);
int task_id=write(fid,ptr,ARR_SIZE*sizeof(int));
while (read(fid,ptr,task_id)==0)
usleep(SLEEP_INTERVAL);
close(fid);
free(ptr);
```

In this code, we use the system call *ioctl* to select the type of the accelerator. We use *write* to pass array information into kernel-space device drivers, and use system call *read* to check the completion of the task. When we have more parameters to pass, we can further encode the information through the *ioctl* system call. The device driver is paired with other system components to understand the parameters passed by the system call. Note that the user-space application can work on something else in-between before the checking. In this code, we simply let the process sleep for some constant interval and check again. Note that we use the return value of the *write* system call to denote the task ID, and this ID is further passed as one parameter of the *read* system call to query the status of this particular task.

6.5.3 Device Driver

The device driver is a kernel-space module that realizes the body of those *read*, *write*, *ioctl* system calls. The device driver has two main features. First, it performs the virtual-to-physical translation for all the pages the accelerator needs to access. The device driver then sends the translated physical addresses along with other task-specific parameters into GAM FIFO (essentially it is an object FIFO). The driver will block if FIFO is full, as internal storage will have a finite size. Note that we use different FIFO channels for different task types. For example, in Figure 6.6, we have two FIFO channels between two microblaze processor; one FIFO (through Xilinx mailbox IP) for *vecadd*, and another FIFO for *vecsub*. Inside the device driver, we also use the kernel mutex/semaphore to prevent concurrent

access for the driver.

Second, in order to pass the done signal of the accelerator back to the application, we implemented a linked-list/hashtable inside the device driver. When the application reserves the accelerator, the device driver generates a unique task ID; this task ID is passed back to both the application and the GAM. The application code uses this unique ID to query the completion of the task. The GAM sends out the completed task ID from a dedicated FIFO channel. Whenever a new task completes, a new completion ID is sent to the FIFO, which triggers an interrupt handler in the device driver which can update the entries in the linked-list accordingly.

6.5.4 Global Accelerator Manager

The Global Accelerator Manager is the component that actually monitors the accelerator status and performs the bare-metal accelerator invocation. Currently, the GAM is implemented through an embedded processor (the second microblaze) with a configuration that minimizes area. We use embedded C code to describe the logic of GAM. The logic is an infinite loop that contains two steps. In the first step, it checks the done signal of all the accelerators. If any of them switched from 0 to 1 in this check, we will pass the task completion signal to the device driver through the mailbox FIFO. In this step, we know the status of all the accelerators, and that information would be stored in a table. In the second step, the GAM checks the incoming task FIFO one by one (we have one task FIFO associated with each type of accelerator) using the non-blocking FIFO test-read operation. When the incoming FIFO for the current type has task objects, the GAM further checks to see whether any accelerator that is compatible with task is idle (by looking at the status table). If so, it reads the task information which is sent out through the FIFO (including the page table along with the parameters of the task to the selected accelerator) to the actual accelerator. Otherwise, it will try out the next task FIFO. When all the task FIFOs are checked and processed, the GAM will return the first step to look for status change.

By having one task FIFO for each type of accelerator, the GAM is capable of performing

out-of-order task processing for tasks of different types.

6.5.5 Accelerator Implementation

The accelerator will keep a local page table, which is sent through the GAM. The accelerator performs burst read access to read the physical page into private on-chip scratchpad, and then starts computation to obtain the results; it then uses burst write access to write the physical output page.

A sample accelerator implementation for the *vecadd* design is listed here.

```
#define PAGE_TABLE_SIZE 512
#include <autopilot_tech.h>
void vecadd(volatile unsigned int * addrFIFO, volatile int* dataBus)
{
#pragma AP interface ap_fifo port=addrFIFO
#pragma AP interface ap_bus port=dataBus

//Define the pcore interfaces
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" \
                                                variable = return
#pragma AP resource core=FSL variable = addrFIFO
#pragma AP resource core=AXI4M variable = dataBus

    unsigned int lenPages=*addrFIFO;
    unsigned int A_pages[PAGE_TABLE_SIZE];
    unsigned int B_pages[PAGE_TABLE_SIZE];
    unsigned int C_pages[PAGE_TABLE_SIZE];
    int buf_A[1024];
    int buf_B[1024];
    int buf_C[1024];

int i,j;
  for(i=0;i<lenPages;i++)
  {
    A_pages[i]=*addrFIFO;
```

```
  }

  for(i=0;i<lenPages;i++)
  {
    B_pages[i]=*addrFIFO;
  }

  for(i=0;i<lenPages;i++)
  {
    C_pages[i]=*addrFIFO;
  }

  for(i=0;i<lenPages;i++)
  {
    memcpy(buf_A,dataBus+(A_pages[i]>>2),1024*sizeof(int));
    memcpy(buf_B,dataBus+(B_pages[i]>>2),1024*sizeof(int));
    for(j=0;j<1024;j++)
#pragma AP pipeline II=1
        buf_C[j]=buf_A[j]+buf_B[j];
    memcpy(dataBus+(C_pages[i]>>2),buf_C,1024*sizeof(int));
  }
}
```

We can see that the accelerator contains the storage for the page table as well as the local storage for storing pages. In the beginning the accelerator reads in the parameters and the page table. The computation and data transfer are performed afterwards. Because the page tables are stored locally within the accelerators, the accelerators then obtain the mapping and perform data transfer for the virtually contiguous addresses.

6.6 Putting the Template in Action

We have implemented the proposed architecture template shown in Figure 6.6. The design includes two type of accelerators, *vecadd* and *vecsub*. We instantiated two accelerator

instances for each accelerator type. All the system components behave properly.

6.6.1 Single Process Execution

First, we test the system running a single process which invokes one accelerator. The kernel is able to flush the cache and perform page translation (virtual-to-physical) at a rate of about 3K cycles per page. The accelerator for *vecadd* needs about 4K to 5K cycles to complete the computation for one page (which needs to read two pages and write one page); while the translation for the three pages took about 9K cycles. Table 6.4 shows the timing breakdown for a single process that invokes the *vecadd* accelerator. We can see that in the current setup, the device driver which performs the address translation currently takes about 2X of the time spent in actual *vecadd* accelerator computation. Surprisingly, the most time-consuming part is the data initialization where pages are touched and page table entries are created or added. When a page is created, the OS will also zero-out the data for security reasons. The zero-out operation is more expensive than the page translation in the device driver because the zero-out operation operates on each data element.

Clearly, the data shown in Table 6.4 is an extreme case where the OS and software portion completely shadow the accelerator execution. We now further consider two synthetic cases. The first case is to repeatedly invoke the accelerator in the application code. The second case is to use a single invocation, but realize multiple rounds of the computation inside the accelerator kernel. Both cases try to emulate an iterative computation where computation works on a same data array multiple times. The number of iterations is set to 100. With 100 iterations, the overhead in touching pages is almost gone. However, case 1 still runs much longer than case 2, because the page translations in device drivers are performed in each invocation. Currently, we are building up an on-demand page translation scheme to reduce the overhead.

The execution time numbers where we use microblaze to perform computation are also listed in the Table 6.4. The simple accelerator is able to obtain about 8X to 9X speedup over the software version on microblaze.

Table 6.4: Timing breakdown for *vecadd* example

	Basic (1 iteration) (with acc)	Synthetic case 1 (100 iterations) (with acc)	Synthetic case 2 (100 iterations) (with acc)
malloc	0.002s	0.002s	0.002s
touch pages	0.157s	0.157s	0.157s
invoke driver	0.027s	2.7s	0.027s
wait for completion	0.015s	1.5s	1.5s
verify output	0.003s	0.003s	0.003s
Total	0.21s	4.4s	1.7s
	Basic (1 iteration) (without acc)		Synthetic case 2 (100 iteration) (without acc)
malloc	0.002s		0.002s
touch pages	0.157s		0.157s
computation	0.122s		12.2s
verify output	0.003s		0.003s
Total	0.29s		12.4s

Table 6.5: Timing for multiple processes of *vecadd (using Synthetic case 2)*

num of processes	elapsed time	increment of elapsed time
1	1.72s	1.72s
2	1.99s	0.27s
3	3.54s	1.55s
4	3.85s	0.31s
5	5.41s	1.56s
6	5.72s	0.31s

6.6.2 Multiple Processes with Accelerator Sharing

Note that our template supports multi-process parallelism and arbitration. Table 6.5 shows the execution time when we have multiple processes which run in parallel (and start at exactly the same timepoint). The microblaze processor is single-core, thus the time spent in software (touching pages and device drivers) will be accumulated.[8] However, we can see that the elapsed time of 2 processes is smaller than the 2X of the elapsed time of single process, which suggests the GAM is operating and distributing the workload into 2 *vecadd* accelerators. The elapsed time will have a steep increase when the system has an odd number of processes. That is because we have two *vecadd* accelerators, and an odd number of processes would create load-balancing issues.

6.6.3 Multiple Processes that Invoke Multiple Accelerators

When the system has multiple accelerators, the GAM will be able to manage them as well. Table 6.6 presents the results. Again the load-balancing will affect the total makespan of the execution. For example, if we have four concurrent processes in total, the makespan of a configuration that has one *vecadd* invocation and three *vecsub* invocations would run longer than the one with two *vecadd* invocations and two *vecsub* invocations.

[8]Current Xilinx Linux on Microblaze only support single core. The template should work for multi-core system like Xilinx Zynq extensible processing platforms (dual-core ARM A9) as well.

Table 6.6: Timing for multiple processes and multiple accelerators of *vecadd and vecsub(using Synthetic case 2)*

#vecadd \ *#vecsub*	0	1	2	3
0	0	1.73s	1.97s	3.59s
1	1.73s	1.98s	2.78s	4.28s
2	1.99s	2.73s	3.62s	5.10s
3	3.55s	4.30s	5.10s	5.45s

We also see that the results are obscured by the accumulation of the software and OS driver execution on the single-core microblaze. The sharing would be more effective when the system has a large number of cores, and the number should be at least larger than the number of accelerators. Running multiple processes concurrently on a single-core will bring in significant context switching overhead. When we have 3 or more processes, the makespan goes up a lot, likely due to more frequent context switchings. The template presented in the section could be smoothly transited to the Xilinx Zynq system as well.

6.6.4 Benefits over Conventional Invocation Scheme

In the conventional design scheme, each accelerator would have its own device node and device driver. Applications need to explicitly select the accelerator it wants to use. Multiple processes can invoke the same accelerator, but the request will be sequentialized by the device driver of the particular accelerator. Our proposed scheme can further dynamically manage multiple accelerators and distribute tasks onto the accelerators.

As a reference point, we also implemented the conventional scheme. Figure 6.7 shows the system diagram of the design. Instead of sending parameters and page table to GAM, these data are directly passed from application to the accelerator through corresponding device drivers.

Table 6.7 shows the results for the conventional scheme. Suppose we have several process-

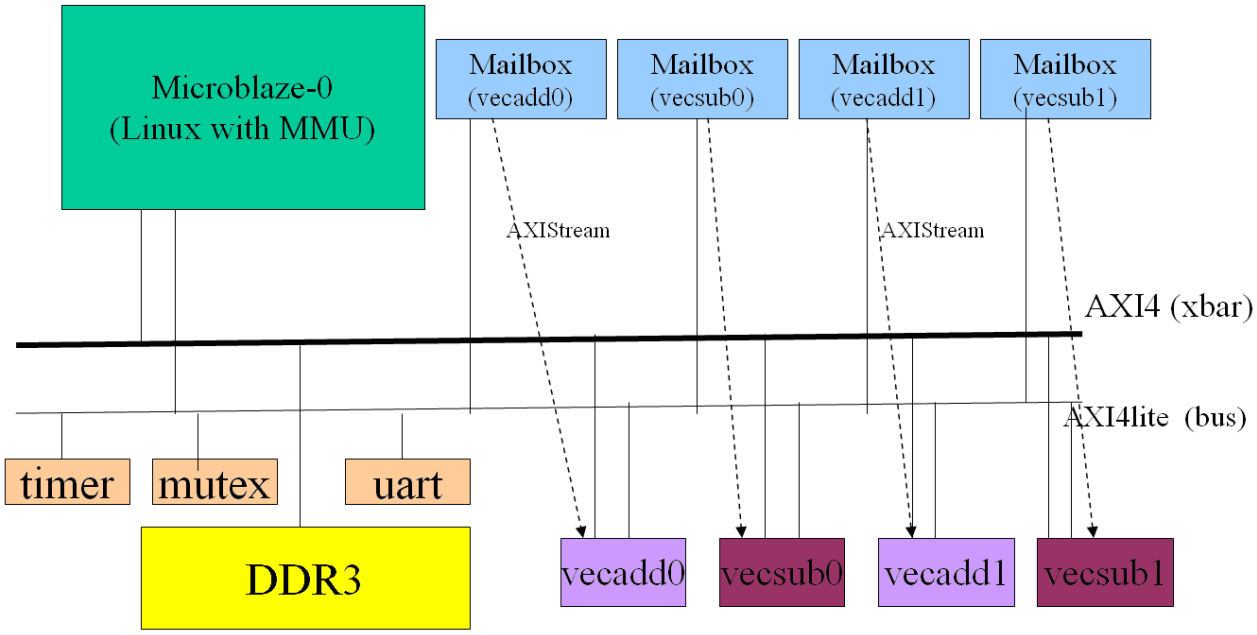


Figure 6.7: Conventional scheme (without GAM) on Xilinx ML605 board

es (*a* invoking *vecadd* and *b* invoking *vecsub*), when we statically select the non-conflicting accelerators (best-case), we can achieve similar makespan as our dynamic case using GAM. When the processes selected the conflicting accelerators (worst-case), the makespan is worse because of an unbalanced workload.

Currently, the scheduling logic implemented in the GAM is very simple and minimal. We expect to further extend that to handle complex management such as accelerator chaining and composition. Although this simple GAM logic can also be incorporated in the device drivers to achieve a similar effect, we choose to enforce the separation to maintain the extensibility.

6.7 Resolving Potential Deadlocks in the Template

Concurrent systems may be subject to deadlock situations if not designed carefully. The CnC computation model is deterministic and it will deadlock only if the unrolled task graph of initial application description has a cycle. In that case, any scheduling of the tasks will

Table 6.7: Timing for multiple processes and multiple accelerators of *vecadd and vecsub(using Synthetic case 2)* in conventional template

(#*vecadd*,#*vecadd*)	worst-case	best-case	with GAM
(1,1)	1.98	1.98	1.98
(2,0)	3.41	1.99	1.98
(2,1)	3.54	2.72	2.73
(2,2)	3.88	3.68	3.62

create deadlock as the cycle exists in the unrolled task graph. If the initial application description does not have those cycles, any scheduling would be deadlock-free [5].

The templates for *dynamic loops* is deadlock-free because the templates work like a streaming fashion and there is no circular wait in the system. We design the template for accelerator management to disallow the "hold and wait" situation. Processes would use one accelerator at any particular time-point and the accelerator is given back to GAM immediately after the task finishes (we do not allow the process to hold accelerator A and request for accelerator B), and thus we avoid potential deadlock as well. In case the accelerator requested is not available, the process can either wait or execute other program segments (e.g, pure software version of the function). Special attention need to be paid once we further extend the template to allow accelerator chaining or composition [110], because that may create "hold and wait" scenarios.

6.8 Conclusions and Future work

In this chapter we presented several architecture templates that can potentially aid the management of coprocessor acceleration. We start from a pure software-based approach that is powerful and supports the cross-device stealing and scheduling. Then we discuss a pure hardware-based manager that only works on a narrowed set of applications called *dynamic loops*. After that, we present a design template that can support the coprocessor management at the system-level.

Still, the template for accelerator-rich CMP is far from complete. The features it can potentially support are still quite minimum compared to the pure software-based approach. The control code in the GAM can also be extended to handle more complex resource management such as buffer allocation, etc.

CHAPTER 7

Concluding Remarks

Future energy-efficient compute systems will incorporate more coprocessors or accelerators. In this thesis, we use the GPU and FPGA to showcase the effectiveness of coprocessor acceleration.

Some major parts of this thesis document many stories on using high-level synthesis tools to perform reconfigurable computing. We are the earliest user of the AutoPilot tool, and many of our designs inspired new features that are further integrated by the tool. As the tool further evolved and strengthened, the startup company AutoESL was acquired by Xilinx in early 2011. This also shows that the HLS approach is viable and will bring in a world-wide user-base. As we see from the thesis, many of the hardware-oriented optimizations still need to be performed manually. The research conducted in this thesis provides concrete examples to inspire further developments of the field of high-level synthesis in general. For example, the research in Chapter 3 inspired the automatic memory partitioning [63], which is later integrated into AutoESL tool. The research in Chapter 4 inspired the automatic buffer allocation, memory prefetch and reuse analysis [55, 85, 86]. The memory partitioning for reuse buffer in Chapter 4 also motivates research to improve the memory partitioning passes for *mod* operations[80].

The journey just begins. As we improved the flow to perform component-specific implementation, more optimizations can be done at system-level and architecture level. We need to develop more automated flow for the coprocessor acceleration, in particular, system-level optimizations and automations. Architecture templates can further augment the HLS-based approach to provide a more complete development environment. We believe that accelerator-rich architecture is the viable approach to address the utilization wall and achieve the goal

of domain-specific customizable computing. The thesis provides a set of building-blocks (at both the component-level and architecture/system level) to realize the futuristic accelerator-rich architectures.

References

[1] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010, pp. 205–218.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11, 2011, pp. 365–376.

[3] J. Cong, G. Reinman, A. Bui, and V. Sarkar, "Customizable domain-specific computing," *IEEE Design Test of Computers*, vol. 28, no. 2, pp. 6 –15, march-april 2011.

[4] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD '09, 2009, pp. 681–688.

[5] "Intel concurrent collections for c++." [Online]. Available: http://softwarecommunity.intel.com/articles/eng/3862.htm

[6] Y. Zhang, L. Peng, B. Li, J.-K. Peir, and J. Chen, "Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, nov. 2011, pp. 205 –215.

[7] Khronos OpenCL Working Group, "The OpenCL Specification - Version 1.0," The Khronos Group, Tech. Rep., 2009.

[8] G. Estrin and C. R. Viswanathan, "Organization of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices," *J. ACM*, vol. 9, no. 1, pp. 41–60, Jan. 1962.

[9] G. Estrin, "Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer," *IEEE Ann. Hist. Comput.*, vol. 24, no. 4, pp. 3–9, Oct. 2002.

[10] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1997, pp. 12–21.

[11] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[12] S. Hauck and A. A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, ser. Systems on Silicon, S. Hauck and A. Dehon, Eds. Morgan Kaufmann, Nov. 2007.

[13] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on fpgas," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '09, 2009, pp. 219–228.

[14] J. Diaz, E. Ros, F. Pelayo, E. Ortigosa, and S. Mota, "Fpga-based real-time optical-flow system," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 16, no. 2, pp. 274 – 279, feb. 2006.

[15] D. Thomas, J. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial monte-carlo simulations," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, july 2007, pp. 168 –173.

[16] M. Singh and B. Leonhardi, "Introduction to the ibm netezza warehouse appliance," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '11. Riverton, NJ, USA: IBM Corp., 2011, pp. 385–386.

[17] M. C. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. VanCourt, "Single pass, blast-like, approximate string matching on fpgas," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 217–226.

[18] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Crescini, C. de Jesus, C. Dick, P. Droz, D. MacMahon, K. Meder, J. Mock, V. Nagpal, B. Nikolic, A. Parsa, B. Richards, A. Siemion, J. Wawrzynek, D. Werthimer, and M. Wright, "Petaop/second fpga signal processing for seti and radio astronomy," in *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, 29 2006-nov. 1 2006, pp. 2031 –2035.

[19] K. H. Tsoi and W. Luk, "Axel, a hetergenous cluster with FPGAs and GPUs," in *FPGA '10: Proc. International Symposium on Field Programmable Gate Arrays*, February 2010.

[20] M. Showerman, W.-M. Hwu, J. Enos, A. Pant, V. Kindratenko, C. Steffen, and R. Pennington, "QP: A heterogeneous multi-accelerator cluster," in *10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[21] "Impulse C." [Online]. Available: http://www.impulsec.com

[22] T. Bollaert, "Catapult Synthesis: A Practical Introduction to Interactive C Synthesis High-Level Synthesis," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Dordrecht: Springer Netherlands, 2008.

[23] "Agility DK design suite." [Online]. Available: http://www.agilityds.com/products/c_based_products/default.aspx

[24] M. Meredith, "High-level SystemC synthesis with Forte's Cynthesizer," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Dordrecht: Springer Netherlands, 2008.

[25] AutoPilot, "http://www.autoesl.com, AutoESL Design Technologies," 2008.

[26] S. Aditya and V. Kathail, "Algorithmic synthesis using PICO," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Dordrecht: Springer Netherlands, 2008.

[27] "C-to-silicon compiler." [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx

[28] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *IEEE International SOC Conference,*, sept. 2006, pp. 199 –202.

[29] "The autoesl autopilot high-level synthesis tool." [Online]. Available: www.bdti.com/MyBDTI/pubs/AutoPilot.pdf

[30] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[31] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proc. of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08, 2008, pp. 260–269.

[32] J. Yeung, C. Tsang, K. Tsoi, B. Kwan, C. Cheung, A. Chan, and P. Leong, "Map-Reduce as a programming model for custom computing machines," in *Proc. of the 16th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '08, Apr. 2008, pp. 149 –159.

[33] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06, 2006.

[34] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Springer-Verlag, 2009, pp. 851–862.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, 2007, pp. 59–72.

[36] "https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C."

[37] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05, 2005, pp. 519–538.

[38] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[39] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, May 1998.

[40] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2009.

[41] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12, 2012, pp. 61–70.

[42] C. A. Mack, "Lithography simulation in semiconductor manufacturing," *Advanced Microlithography Technologies*, vol. 5645, no. 1, pp. 63–83, 2005.

[43] M. S. Yeung, "Fast and rigorous three-dimensional mask diffraction simulation using battle-lemarie wavelet-based multiresolution time-domain method," *Optical Microlithography XVI*, vol. 5040, no. 1, pp. 69–77, 2003.

[44] Y. C. Pati and T. Kailath, "Phase-shifting masks for microlithography: automated design and mask requirements," *J. Opt. Soc. Am. A*, vol. 11, no. 9, p. 2438, 1994.

[45] Y. Cao, Y.-W. Lu, L. Chen, and J. Ye, "Optimized hardware and software for fast full-chip simulation," *Optical Microlithography XVIII*, vol. 5754, no. 1, pp. 407–414, 2004.

[46] Mentor, "Datasheet of Calibre nmOPC, Mentor Graphics Corporation," 2004.

[47] Y.-T. Wang, C.-M. Tsai, and F.-C. Chang, "Lithographic simulations using graphical processing units," Patent 20 060 242 618, October, 2006, united States Patent Application 20060242618.

[48] N. B. Cobb and A. Zakhor, "Fast, low-complexity mask design," T. A. Brunner, Ed., vol. 2440, no. 1. SPIE, 1995, pp. 313–327.

[49] A. K.-K. Wong, *Optical imaging in projection microlithography.* Bellingham, WA: SPIE Press, 2005.

[50] P. Yu and D. Z. Pan, "A novel intensity based optical proximity correction algorithm with speedup in lithography simulation," in *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, 2007, pp. 854–859.

[51] N. B. Cobb, "Fast optical and process proximity correction algorithms for integrated circuit manufacturing," Ph.D. dissertation, UC Berkeley, 1998.

[52] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[53] I. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of fast fourier transforms for real-time signal and image processing," *IEE Proceedings - Vision, Image, and Signal Processing*, vol. 152, no. 3, pp. 283–296, 2005.

[54] A. K.-K. Wong, "Private communication," 2007, Magma Design Automation Inc.

[55] J. Cong, P. Zhang, and Y. Zou, "Combined loop transformation and hierarchy allocation for data reuse optimization," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11. IEEE Press, 2011, pp. 185–192.

[56] J. K. Tanskanen, T. Sihvo, and J. Niittylahti, "Byte and modulo addressable parallel memory architecture for video coding," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 14, no. 11, pp. 1270–1276, Nov. 2004.

[57] M. Doggett and M. Meissner, "A memory addressing and access design for real time volume rendering," in *ISCAS '99. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, 1999.*, Jul 1999, pp. 344–347 vol.4.

[58] Xtremedata, "http://www.xtremedatainc.com, XD1000 Development System," 2007.

[59] O. Mencer and R. G. Clapp, "Accelerating 2D FFTs and convolutions for seismic processing," 2007, brief Notes by Maxeler Technologies.

[60] V. Podlozhnyuk, "FFT based 2D convolution," 2007, nVIDIA white paper.

[61] J. W. Park, "An efficient buffer memory system for subarray access," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 3, pp. 316–335, 2001.

[62] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "Multimedia rectangularly addressable memory," *IEEE Transactions on Multimedia*, vol. 8, no. 2, 2006.

[63] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, nov. 2009, pp. 697 –704.

[64] C. Castro-Pareja, J. Jagadeesh, and R. Shekhar, "FAIR: a hardware architecture for real-time 3-D image registration," *IEEE Transactions on Information Technology in Biomedicine*, vol. 7, no. 4, pp. 426 –434, 2003.

[65] C. Castro-Pareja and R. Shekhar, "Hardware acceleration of mutual information-based 3-D image registration," *Journal of Image Science and Technology*, vol. 49, pp. 105 – 113, 2005.

[66] O. Dandekar and R. Shekhar, "FPGA-accelerated deformable image registration for improved target-delineation during CT-guided interventions," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 1, no. 2, pp. 116 –127, 2007.

[67] G. Christensen, R. Rabbitt, and M. Miller, "Deformable templates using large deformation kinematics," *IEEE Transactions on Image Processing*, vol. 5, no. 10, pp. 1435 –1447, Oct. 1996.

[68] M. Bro-Nielsen and C. Gramkow, "Fast fluid registration of medical images," in *Proc. VBC*, 1996, pp. 267–276.

[69] J.-P. Thirion, "Non-rigid matching using Demons," in *Proc. CVPR*, Jun. 1996, pp. 245 –251.

[70] E. D'Agostino, F. Maes, D. Vandermeulen, and P. Suetens, "A viscous fluid model for multimodal non-rigid image registration using mutual information," in *Proc. MICCAI*, 2002, pp. 541–548.

[71] I. Yanovsky, A. D. Leow, S. Lee, S. J. Osher, and P. M. Thompson, "Comparing registration methods for mapping brain change using tensor-based morphometry," *Medical Image Analysis*, vol. 13, no. 5, pp. 679–700, October 2009.

[72] J. Modersitzki, *Numerical Methods for Image Registration.* Oxford University Press, 2004.

[73] L. Alvarez and L. Mazorra, "Signal and image restoration using shock filters and anisotropic diffusion," *SIAM J. Numer. Anal.*, vol. 31, pp. 590–605, April 1994.

[74] I. T. Young and L. J. van Vliet, "Recursive implementation of the gaussian filter," *Signal Process.*, vol. 44, pp. 139–151, June 1995.

[75] R. Deriche, "Fast algorithms for low-level vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, pp. 78–87, January 1990.

[76] J. Cong, M. Huang, and Y. Zou, "3D recursive Gaussian IIR on GPU and FPGAs, a case study for accelerating bandwidth-bounded applications," in *Proc. SASP*, 2011.

[77] A. Roldao-Lopes, A. Shahzad, G. Constantinides, and E. Kerrigan, "More flops or more precision? accuracy parameterizable linear equation solvers for model predictive control," in *Proc. FCCM*, 2009, pp. 209 –216.

[78] H. Yu and M. Leeser, "Automatic sliding window operation optimization for FPGA-based computing boards," in *Proc. FCCM*, 2006, pp. 76–88.

[79] O. Dandekar, C. Castro-Pareja, and R. Shekhar, "FPGA-based real-time 3D image preprocessing for image-guided medical interventions," *Journal of Real-Time Image Processing*, vol. 1, pp. 285–301, 2007.

[80] Y. Wang, P. Zhang, X. Cheng, and J. Cong, "An integrated and automated memory optimization flow for fpga behavioral synthesis," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 30 2012-feb. 2 2012, pp. 257 –262.

[81] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *Proc. ICS*, 2009, pp. 256–265.

[82] S. van Haastregt and B. Kienhuis, "Automated synthesis of streaming C applications to process networks in hardware," in *Proc. DATE*, 2009, pp. 890–893.

[83] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07, 2007, pp. 235–244.

[84] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: A geometric programming framework," *IEEE TCAD*, vol. 28, no. 3, pp. 305 –315, 2009.

[85] J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching algorithm for scratchpad memory," in *Proc. DAC*, 2011.

[86] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *DAC'12, Proc. Design Automation Conference*, 2012.

[87] S. Coric, M. Leeser, E. Miller, and M. Trepanier, "Parallel-beam backprojection: an FPGA implementation optimized for medical imaging," in *Proceedings of International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '02, 2002, pp. 217–226.

[88] N. Gac, S. Mancini, M. Desvignes, and D. Houzet, "High speed 3D tomography on CPU, GPU, and FPGA," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 5:1–5:12, January 2008.

[89] J. Li, C. Papachristou, and R. Shekhar, "An FPGA-based computing platform for real-time 3D medical imaging and its application to cone-beam ct reconstruction," *J. Imaging Science and Technology*, vol. 49, pp. 237–245, 2005.

[90] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the common unified device architecture (CUDA)," in *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, vol. 6, 26 2007-nov. 3 2007, pp. 4464 –4466.

[91] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, "Impulse C vs. VHDL for accelerating tomographic reconstruction," in *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, may 2010, pp. 171 –174.

[92] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society, Series B*, vol. 39, no. 1, pp. 1–38, 1977.

[93] A. H. Andersen and A. C. Kak, "Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm," *Ultrason Imaging*, vol. 6, pp. 81–94, Jan 1984.

[94] B. Keck, H. Hofmann, H. Scherl, M. Kowarschik, and J. Hornegger, "GPU-accelerated SART reconstruction using the CUDA programming environment," in *Proceedings of SPIE*, E. Samei and J. Hsieh, Eds., vol. 7258, Lake Buena Vista, 2009.

[95] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *Nuclear Science, IEEE Transactions on*, vol. 52, no. 3, pp. 654 – 663, june 2005.

[96] D. Stsepankou, K. Kommesser, J. Hesser, and R. Manner, "Real-time 3D cone beam reconstruction," in *Nuclear Science Symposium Conference Record, 2004 IEEE*, vol. 6, oct. 2004, pp. 3648 – 3652 Vol. 6.

[97] M. Yan and L. A. Vese, "Expectation maximization and total variation-based model for computed tomography reconstruction from undersampled data," in *Proc. SPIE Conference on Medical Imaging: Physics of Medical Imaging*, 2011.

[98] A. Bui, K. Cheng, J. Cong, L. Vese, Y. Wang, B. n, and Y. Zou, "Platform Characterization for Domain-Specific Computing," in *Proceedings of the 17th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '12, 2012.

[99] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07, 2007, pp. 162–173.

[100] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10. New York, NY, USA: ACM, 2010, pp. 311–322.

[101] M. Wolfe, "More iteration space tiling," in *Proc. Conference on Supercomputing*, 1989, pp. 655–664.

[102] A. Lim and M. S. Lam, "Communication-free parallelization via affine transformations," in *Proc. International Workshop on Languages and Compilers for Parallel Computing*, 1995, pp. 92–106.

[103] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, "Map-reduce as a programming model for custom computing machines," in *FCCM '08: Proc. International Symposium on Field-Programmable Custom Computing Machines*, April 2008, pp. 149–159.

[104] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA - a case study of RankBoost acceleration," in *Proc. International Symposium on Field Programmable Gate Arrays*, Feb. 2010.

[105] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. International Symposium on Field Programmable Gate Arrays*, 2005, pp. 63–74.

[106] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *Proc. International Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 349–352.

[107] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proc. International Symposium on Field Programmable Gate Arrays*, 2005, pp. 75–85.

[108] Y. Elkurdi, D. Fernández, E. Souleimanov, D. Giannacopoulos, and W. J. Gross, "FPGA architecture and implementation of sparse matrix vector multiplication for the finite element method," *Computer Physics Communications*, vol. 178, pp. 558–570, Apr. 2008.

[109] T. A. Davis, "University of Florida sparse matrix collection," *NA Digest*, vol. 92, 1994.

[110] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proceedings of the Design Automation Conference*, ser. DAC '12, 2012.