UNIVERSITY OF CALIFORNIA,
IRVINE

A Fine-grain Parallel Execution Model for Homogeneous/Heterogeneous Many-core Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Tongsheng Geng

2018

DEDICATION

To my beloved parents, sister and friends
For your endless love and support, without which I could never make it this far.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Professor Jean-Luc Gaudiot, who guided me in my research and encouraged me with all his passion, patience and immense knowledge. It has been an honor to work and study under his guidance. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Professor Nader Bagherzadeh, Professor Rainer Doemer, and Professor Stéphane Zuckerman for their insightful comments and encouragement, but also for the hard questions which incentivized me to expand the horizons of my research. Thanks also to Professor Tony Givargis and Professor Mohammad Al Faruque for serving on my Ph.D. qualifying examination committee.

I would like to share a special word of gratitude with professor Stéphane Zuckerman, from the Université Paris-Seine, Université de Cergy-Pontoise, for the generous sharing of his time and expertise in the course of my Ph.D study and related research. He initiated the DARTS project on which most this works is based and he spent much time passing his knowledge on to me.

I am extremely grateful to professor Alfredo Goldman, from the University of São Paulo, and Professor Sami Habib, from Kuwait University, who gave me extensive personal and professional guidance and taught me a great deal about both scientific research and life in general.

My sincere thanks also goes to professor Guang R. Gao, from the University of Delaware, who provided me with an opportunity to participate in the activities of his extraordinary multi-disciplinary research group. Many thanks to the University of Delaware CAPSL research group members who granted me access to the laboratory and research facilities and provided technical support. Without they precious support it would not have been possible to conduct this research.

I thank all my fellow labmates in the PArallel Systems & Computer Architecture Lab (PASCAL) for the stimulating discussions, for their support during the sleepless nights we spent working together before deadlines, and for all the fun we had through the years.

Professor Gaudiot's wife, Teresa, took me under her wing and taught me so much about American culture and language. She has given me more than I could ever give her credit for here. I am immensely thankful to my "American mother."

A work of grateful acknowledgement to the National Science Foundation who supported my research under Grants No. XPS-1439165, XPS-1439097 and CCF-1763793. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

CURRICULUM VITAE

Tongsheng Geng

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2018**
University of California, Irvine

Master of Science in Institute of Microelectronics **2011**
Tsinghua University

Bachelor of Science in Mechanical Engineering **2006**
Zhengzhou University

RESEARCH EXPERIENCE

Graduate Research Assistant **2012–2018**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2015–2018**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

The Importance of Efficient Fine-Grain Synchronization for Many-Core Systems **September 2016**
In Languages and Compilers for Parallel Computing-29 th International Workshop,LCPC2016,Rochester, NY.USA, September 28-30,2016,pp203-217

SOFTWARE

Fine-grain Execution Model Run-time system `https://github.com/gengtsh/darts`
Runtime system that implements fine-grain parallel execution execution model

ABSTRACT OF THE DISSERTATION

A Fine-grain Parallel Execution Model
for Homogeneous/Heterogeneous Many-core Systems

By

Tongsheng Geng

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2018

Professor Jean-Luc Gaudiot, Chair

Computing systems have undergone a fundamental transformation from single core devices to devices with homogeneous/heterogeneous many-cores connected within a single or multiple chips. However, while the core count per chip continues to increase dramatically, the available on-chip memory per core is only getting marginally bigger. How to successfully explore parallelism and deliver scalability is a major research issue and we have successfully attacked three main problems:

First, it is well known that, in homogeneous shared-memory many-core systems, traditional coarse-grain multithreading models are reaching their limits. We have thus proposed and designed a fine-grain event-driven multithreading execution model that will deliver the parallelism required to efficiently operate with dependence-heavy applications in shared-memory systems. By performing finer-grained and hierarchical synchronization, even "almost embarrassingly parallel" workloads can obtain large performance improvement.

Second, it has been recognized that, in heterogeneous High Performance Computing systems, the performance depends on how well the scheduler can allocate workloads to the appropriate computing devices and make communication and computation to overlap efficiently. With different types of resources integrated into one system, the complexity of the scheduler correspondingly increases. Moreover, when the applications have varying problem sizes on different heterogeneous resources, the complexity of the scheduler grows accordingly. Our proposed profile-based Iterative Dynamic Adaptive Work-Load balance scheduling approach (IDAWL) combines offline machine learning with online scheduling offers a general approach to efficiently utilize, in a dynamic fashion, available heterogeneous resources.

Finally, for those applications where the computation can be naturally expressed as streams, our Stream-based fine-grain program execution model, was developed to explore parallelism of hierarchical heterogeneous resources. It can exploit two levels (coarse- and fine-grain) of parallelism, efficiently utilizing locally available heterogeneous resources to construct streaming pipeline stages and minimize data movement to enhancing data locality.

Chapter 1

Introduction

Computing systems have undergone a fundamental transformation from single core devices to devices with homogeneous/heterogeneous many-cores connected within a single or multiple chips. In the past decade, The number of Processing Elements (PEs) found in general-purpose high-performance processors has increased hundred-fold, as demonstrated by, *e.g.*, the latest processors from Intel® and IBM®. Further, heterogeneous resources(GPUs, FPGAs, storage, *etc.*) are widely used in High-Performance Computing (HPC) platforms. For instance, the number of platforms of the Top500 equipped with accelerators has significantly increased during the last years [1]. In the future it is expected that the nodes' heterogeneity and count will increase even more: hybrid computing nodes mixing general purpose units with accelerators, I/O nodes, nodes specialized in data analytics, *etc.* The interconnect of a huge number of such nodes will also lead to more heterogeneity. Many issues must be envisioned in new software/hardware systems, including programmability, scalability, performance evaluation, and power efficiency [2]. Better performance and power consumption obtained from the use of more appropriate resources according to the computations to perform will be obtained at the cost of code development and more complex resource management. How to successfully exploit parallelism and deliver scalability is a major research issue in both Homogeneous and heterogeneous many-core systems.

In contrast to the rapid development of the hardware, the programming models and program execution models (`PXMs`), in the area of software parallel computing in homogeneous shared-memory many-core systems used by scientific applications, have remained mostly the same: `MPI` is used for inter-node communication and *OpenMP* is still favored for shared-memory computations. However, while the *OpenMP* standard has evolved to provide ways to define fine-grain task-dependence graphs in *OpenMP*4 [3] and *OpenMP*4.5 [4], a large majority of application programmers use *OpenMP* features that are mostly related to parallel `for` loops, a coarse-grain style to express parallelism, *i.e.*, a programming style which requires the insertion of global barriers rather than finer-grain point-to-point synchronizations between individual threads. As long as the core count remained low in terms of shared-memory compute nodes, global barriers were reasonable. However, this approach is not scalable: as the core count increases, the stress sustained by the memory subsystem leads to unacceptable contention on the various memory banks (both at the cache and DRAM levels). Moreover, the coarse-grain approach is still sustainable for CPU-bound workloads, but, with memory-bound applications, global barriers may kill performance due to the hardware synchronization mechanism, while high-performance based synchronization constructs rely on some sophisticated variation of busy-waiting (potentially mitigated with a sleep policy) which can hog the memory subsystem, particularly in the case of "almost embarrassingly parallel" algorithms and programs.

In heterogeneous High Performance Computing systems, the performance depends on how well the scheduler can allocate workloads to the appropriate computing devices and make communication and computation efficiently overlap. With different types of resources integrated into one system, the complexity of the scheduler correspondingly increases. Manual scheduling workloads is time-consuming, error-prone and becomes nearly infeasible for developers since any changes of hardware may render the original scheduling approach useless.

A good heterogeneity-aware scheduler must leverage load-balancing techniques in order to obtain the best workload partition between CPUs and general-purpose accelerators—*e.g.*, a GPU. Naïve heuristics may result in worsened performance and power consumption. This is particularly the case for iterative algorithms, such as stencil-based computations which require regular host-accelerator synchronizations. An unbalanced workload may cause a huge drag in performance. At the same time, stencil-based computations are at the core of many essential scientific applications: stencils are used in image processing algorithms, *e.g.*, convolutions; partial differential equation solvers, Laplacian transforms, or computational fluid dynamics; linear algebra, the Jacobi method; *etc.*

Finally, streaming applications (where the computation can be naturally expressed as streams) include scientific computations, embedded applications, as well as the emerging field of social media processing. Program execution models centered on streams have been studied by many researchers and have been an active field of research for the past 30 years [5–9]. The most relevant early work on streams is the data flow execution model pioneered by Dennis [10, 11], the Synchronous Data Flow (SDF) model [12, 13] and Program Dependence Graph(PDG) model [14]. Other work include data-flow software pipelines [15, 16, 16–18]. However, these models do not address the parallelism and resources utilization problems existing in highly heterogeneous and hierarchical system. Moreover, it should be noted that core count per chip continues to increase dramatically while the available on-chip memory per core is only getting marginally bigger. This means that data locality, already a must-have in high-performance computing, will become an even critical point in streaming processing since smooth data movement will be a must in streaming processing.

The goal of this work is to propose a fine-grain event-driven parallel execution model to solve parallel computing, resource utilization and scalability issues coming with the new era of High-Performance computing. Specifically, it deals with the coarse-grain synchronization issues in homogeneous shared-memory many-core system, workload balance and streaming parallelism issues in hierarchical heterogeneous many-core system. The structure of this document is as follows. Section 2 reviews a fine-grain event-driven program execution model, abstract machine and correspondingly runtime system as background information; Section 3 proposes to demonstrate the need for fine-grain synchronization in homogeneous shared-memory many-core systems; Section 4 proposes an approach, combining offline machine learning with online scheduling, to offers a general approach to efficiently utilize, in a dynamic fashion, available heterogeneous resources; Section 5 proposes a stream-based fine-grain program execution model for streaming applications to explore parallelism of hierarchical heterogeneous resources; Section 6 concludes this work and presents the planned future work.

Chapter 2

The *Codelet* Abstract Machine and Runtime System

The *Codelet* Model [19] is a fine-grain event-driven program execution model which targets current and future multi- and many-core architectures[1]. In essence, it is inspired by data flow model of computation [20] and dynamic [21] models.

The quantum of execution is the *Codelet*, a fine-grain task that executes a sequence of machine instructions until completion and runs on a *von Neumann* type of processing element. A *Codelet* `fires` when all its dependencies (data and resource requirements) are met. A *Codelet* cannot be preempted while it is firing, *i.e.*, while it is executing its instructions.

Each time a *Codelet* produces data items or releases a shared resource, it signals the other *Codelet*s which depend on such data item(s) and/or resource(s). Such a group of *Codelet*s can be modeled as a directed graph called a *Codelet Directed Graph* (CDG) where *Codelet*s are the nodes and their dependencies are the edges. In general, a given CDG statically specifies the dependencies between the *Codelet*s it contains.

[1]A short introduction is available at `http://www.capsl.udel.edu/codelets.shtml`.

A *Threaded Procedure* (TP) is a container that comprises a CDG and data to be accessed by the *Codelet*s it contains. A TP is essentially an asynchronous function: once it is invoked, its caller resumes its execution. The TP itself can be scheduled to run anywhere in one of the clusters of the *Codelet Abstract Machine*, and the *Codelet*s can run on any of the cluster's Computation Units. However, once scheduled, a TP and its content (data and code) must remain allocated in its cluster. However, individual *Codelet*s may be scheduled for execution in any of the computation units comprised in the cluster.

The *Codelet* model relies on a *Codelet Abstract Machine* (`CAM`), which models a general purpose many-core architecture with two types of cores: scheduling units (SUs), which perform resource management and scheduling, and computation units (CUs), which carry out the computation. Compared to CUs, SUs have two more functions: control and synchronize all the CUs. A CAM is an extensible, scalable and hierarchy model. One cluster contains at least one SU, one or more CUs, and some local memory. Clusters can be grouped together to form a chip, which itself has access to some memory modules; Multiple chips consist of a node, and multiple nodes form a full CAM. The communication of between and within components of each level of hierarchy is done by the interconnection network.

A CAM is meant to be mapped on real hardware: the number of clusters, and computation units per cluster will be directly influenced by the actual hardware architecture on which a codelet program should be running. Further, different configurations may be used on the same target hardware, depending on the nature of the application.

The Delaware Adaptive Run-Time System (*DARTS*) [22–24] is a faithful implementation of the CAM. It is written in portable C++, and runs on any UNIX-based distribution. It targets shared-memory nodes. *DARTS* executes on regular multi-core chips and assigns a role to each core or thread: each processing element is either a SU or a CU. It also implements the configurable CAM, which can be configured at run-time by the user (or in code by the programmer). By default, *DARTS*'s CAM considers each socket to be a cluster, and assigns a single SU per cluster. Furthermore, there are two queues or pools, ready queue and waiting queue, to store these *Codelets*. When all the requirements of one *Codelet* are met, this *Codelet* will be moved from waiting queue to ready queue and SUs will push it to CUs ready queue to execute or execute by himself if all the CUs are busy or their ready queues are full. In specific case, to reduce data movement and utilize the data locality character, *DARTS* can pin the *Codelet*s into CUs when the same *Codelet*s are invoked repeatedly or periodically. *DARTS* is also extendable. Section 4.2.2 introduces *Heterogeneous-DARTS* which extend *DARTS* to support both CPU and GPU resources parallelism computing. Section 5 introduces *Streaming-DARTS* to support stream processing.

Chapter 3

Exploiting Fine-Grain Event-Driven Multithreading

3.1 Introduction and Motivation

In the past decade, the number of Processing Elements (PEs) found in general-purpose high-performance processors has increased between forty and a hundred times, as demonstrated by, *e.g.*, the latest processors from Intel® and IBM®. Further, the recent appearance of "accelerators" have reached even higher PE counts in recent years.

In the meantime, the programming models and Program execution Models (PXMs) used by scientific applications have remained mostly the same: *MPI* is used for inter-node communication and *OpenMP* is still favored for shared-memory computations. However, while the *OpenMP* standard has evolved to include finer-grain tasks with *OpenMP*3 [25], and even to provide ways to define task-dependence graphs in *OpenMP*4 [3] and *OpenMP*4.5 [4], a large majority of application programmers use *OpenMP* features that are mostly related to parallel `for` loops, sometimes exploiting the nature of their scheduling and the size of their iteration blocks. In turn, this approach tends to favor a rather coarse-grain style to express parallelism, *i.e.*, a programming style which requires the insertion of global barriers rather than finer-grain point-to-point synchronizations between individual threads.

As long as the core count remains low in terms of shared-memory compute nodes, global barriers is reasonable. However, it is not scalable: as the core count increases, the stress sustained by the memory subsystem leads to unacceptable contention on the various memory banks (both at the cache and DRAM levels). Moreover, the coarse-grain approach is still sustainable for CPU-bound workloads, but, with memory-bound applications, global barriers may kill performance due to the underlying hardware: on x86 machines, synchronization usually leverages the use of atomic operations, which can seriously hamper performance in a multi-core, multi-socket environment [26]. In particular, memory-bound workloads tend to tax the interconnection network which links sockets together. In general, high-performance based synchronization constructs rely on some sophisticated variation of busy-waiting (potentially mitigated with a sleep policy) which can hog the memory subsystem, as the system software designer expects contention to be low and the workload to be well-balanced—particularly in the case of ”almost embarrassingly parallel” algorithms and programs. However, recent compute nodes feature a high core and hardware thread count: cores and hardware threads nowadays share more and more resources, such as functional units, caches, and DRAM banks. As a result, it could be tedious and error-prone to parallelism even ”almost embarrassingly parallel” workloads with a high compute-to-memory operations ratio, such as matrix multiplication. On more memory-intensive kernels, the same problem arises, but on a larger scale. One such example is the use of partial differential equation iterative solvers for linear equation systems, in particular the application of Jacobi or Gauss-Seidel methods to a linear system by resorting to a stencil-based iterative solver: every element of an n-dimensional grid depends on its immediate neighbors, and potentially more remote ones. Such algorithms are used in a multitude of applications, *e.g.*, to solve Laplace equations used in heat conduction and computational fluid dynamics solvers.

Section 3 demonstrates the need for fine-grain synchronization even in the presence of rather coarse-grained workload partitioning using a stencil-based iterative solver, 5-point 2D stencil kernel, as an example. Different variants of 2D stencil(section 3.2), including coarse-grain variants and fine-grain variants, are running on two different types of machines featuring x86 processors, with a different number of processing elements per chip, but also a different number of sockets per node. Furthermore, a realistic stencil-based computation mini-app, LULESH(Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [27–29], is introduced in section 3.2.4 to support the idea that in a dependence-heavy context, yet with a uniform amount of work per thread, *fine-grain synchronization matters*, even in "regular" general-purpose systems.

3.2 Methodology: Apply Fine-Grain Parallelism

This section starts from a coarse-grain "parallel for" loop implementation of a simple, naïve 5-point 2D stencil computation expressed in *OpenMP4* and ported it to leverage the *Codelet* Model in *DARTS*. After that, Fine-grain task mechanism introduced in *OpenMP4.5* [4] and fine-grain model of *Codelet* will be described including data dependence and synchronization constructs. A realistic application, *LULESH* [27–29], will be used as an example to evaluate the differences of fine-grain and coarse-grain synchronization.

3.2.1 Basic Implementation of a Parallel Coarse-Grain 5-Point 2D Stencil Computation

The code presented in Listing 3.1 is a naïve *OpenMP* version of a coarse-grain multithreaded 5-point 2D stencil computation. To simplify the problem, a given number of time step instead of convergence test is considered. This version of the stencil code privatizes everything,

```
 1  void  stencil_5pt(double* restrict dst,    double* restrict src,
 2                    const size_t      n_rows, const size_t      n_cols,
 3                    size_t            n_steps)
 4  {
 5    typedef double (*Array2D)[n_cols];
 6  # pragma omp parallel default(none) shared(src, dst) \
 7                firstprivate(n_rows, n_cols, n_tsteps)
 8    {
 9      Array2D  D = (Array2D) dst, S = (Array2D) src;
10      size_t n_ts   = n_tsteps;
11      while (n_ts-- > 0) {
12  #     pragma omp for nowait
13        for (size_t i=1; i<n_rows-1; ++i)
14          for (size_t j=1; j<n_cols-1; ++j)
15            D[i][j] = 0.25 * (S[i-1][j]+S[i+1][j] + S[i][j-1]+S[i][j
                 +1]);
16        SWAP_PTR(&D,&S);
17  #     pragma omp barrier
18      }
19    }
20  }
```

Listing 3.1: Naïve 5-Point 2D Stencil kernel—*OpenMP* version. Everything has been privatized, but threads can only proceed to the next time step if they all have swapped their array pointers.

so that each thread can perform all computations including pointer swapping and moving forward to the next time step. The computation itself is located in a parallel `for` loop (see line 13). There is no the implicit barrier at the end of the loop because of using `for nowait` clause, so that threads that finish processing their own iteration chunk may proceed to swap their source and destination pointers for the next time step. The only required synchronization is the global barrier (line 17) before looping to the next iteration in the `while` loop, to ensure that all threads have properly swapped their array pointers before resuming the computation.

A direct translation of Listing 3.1's code into a *DARTS* framework can be found in Listing 3.2 and 3.3. Obviously, comparing with *Openmp* version, the *DARTS* version of code is more verbose using *DARTS* runtime system API. The various keywords emphasized in bold red are macros defined to simplify the writing of *DARTS* programs. A short description of the various keywords is provided in Table 3.1. Listing 3.2 defines a Threaded Procedure(*TP*)

Keyword	Description
DEF_TP	Defines a new threaded procedure
DEF_CODELET	Defines a new codelet
DEF_CODELET_ITER	Defines a new codelet with a specific ID
SYNC	Signals a codelet within the same TP frame
SIGNAL	Signals a codelet in another TP frame
SIGNAL_CODELET	Signals a codelet from a TP setup phase
LOAD_FRAME	Loads the threaded procedure frame
FIRE(CodeletName)	Code to run when `CodeletName` is fired
INVOKE(TPName,...)	Invokes a new TP from a codelet

Table 3.1: Codelet Model macros and their meaning.

```
DEF_CODELET_ITER ( Compute, 0, NO_META_DATA );
DEF_CODELET      ( Barrier, 2, NO_META_DATA );
DEF_TP(Stencil) {
//  Data
  double *dst, *src;
  size_t  n_rows, n_cols, n_tsteps;
//  Code
  Compute* compute;
  Barrier  barrier;

  Stencil(double* restrict p_dst,   double* restrict p_src,
          size_t           p_nRows, size_t           p_nCols,
          size_t           p_nTSteps)
  : dst(p_dst), src(p_src)
  , n_rows(p_nRows), n_cols(p_nCols), n_tsteps(p_nTSteps)
  , compute(new Compute[g_nCU])
  , barrier(g_nCU,g_nCU,this,NO_META_DATA)
  {
    for (size_t cid = 0; i < g_nCU; ++cid) {
      compute[cid] = Compute{1,1,this,NO_META_DATA,cid};
      SIGNAL_CODELET(compute[cid]);
    }
  }
};
```

Listing 3.2: Coarse-Grain 5-Point 2D Stencil kernel—*DARTS* version. `Stencil` *TP* definition and its associated codelets.

named `Stencil` and two *Codelets*, named `Compute` and `Barrier`. `Compute` *Codelet* is defined with default 0 dependence counts and `Barrier` *Codelet* is defined with default 2 dependence counts. The dependence count can be overridden when the *Codelet* is instantiated in *TP*.

The `Stencil` *TP* is essentially a C++ `struct` which allocates the right amount of *Codelets*

```
FIRE(Compute) {
  LOAD_FRAME(Stencil);
  typedef double (*Array2D)[n_cols];
  Array2D      D      = (Array2D) FRAME(dst),
               S      = (Array2D) FRAME(src);
  const size_t n_rows  = FRAME(n_rows),
               n_cols  = FRAME(n_cols),
               n_steps = FRAME(n_steps);

  // current codelet's ID
  size_t cid = getID(),
         lo  = lower_bound(n_cols,cid),
         hi  = upper_bound(n_cols,cid);
  for (size_t i = lo; i < hi-1; ++i)
    for (size_t j = 1; j < n_cols-1; ++j)
      D[i][j] = 0.25 * (S[i-1][j]+S[i+1][j]+ S[i][j-1]+S[i][j+1]);
  SYNC(barrier);
  EXIT_TP();
}

FIRE(Barrier) {
  LOAD_FRAME(Stencil);
  if ( FRAME(n_tstep) == 0 ) {
    SIGNAL(done);
    EXIT_TP();
  }

  double *src = FRAME(dst), *dst = FRAME(src);
  size_t n_rows = FRAME(n_rows), n_cols = FRAME(n_cols),
         n_tsteps = FRAME(n_tsteps);
  Codelet *done = FRAME(done);

  INVOKE(Stencil, src, dst, n_rows, n_cols, n_steps-1,
         done);
  EXIT_TP();
}
```

Listing 3.3: Coarse-Grain 5-Point 2D Stencil kernel—*DARTS* version. Compute *Codelet* and barrier *Codelet* definition. a new *TP* is invoked at each new iteration step.

for a given cluster of cores, and holds the data which the *Codelet*s can access. The `Compute` *Codelet* proceeds to execute the stencil operation for one time step over a chunk of the data. When it is done firing, it signals the `Barrier` *Codelet*, which collects all the signals of all firing `Compute`s. `Barrier` then proceeds to invoke a new `Stencil` *TP* where the source and destination arrays are swapped in the parameters list, and the time step is decreased. This variant performs poorly compared to *OpenMP* since creating a new *TP* in every time step yields a rather high overhead which involves dynamically allocating and deallocating intermediate data structures to hold the *TP* frame, as well as creating a set of *Codelet*s to process portions of iteration space.

```
FIRE(Compute) {
  LOAD_FRAME(Stencil);
  typedef double (*Array2D)[n_cols];
  Array2D D = (Array2D) FRAME(dst), S = (Array2D) FRAME(src);
  const size_t n_rows  = FRAME(n_rows), n_cols  = FRAME(n_cols),
               n_steps = FRAME(n_steps);

  size_t cid = getID(), // current codelet's ID
         lo  = lower_bound(n_cols,cid),
         hi  = upper_bound(n_cols,cid);

  RESET(compute[cid]);
  for (size_t i = lo; i < hi-1; ++i)
    for (size_t j = 1; j < n_cols-1; ++j)
      D[i][j] = 0.25 * (S[i-1][j]+S[i+1][j] + S[i][j-1]+S[i][j+1]);
  SYNC(barrier);
  EXIT_TP();
}

FIRE(Barrier) {
  LOAD_FRAME(Stencil);
  if ( FRAME(n_tstep) == 0 ) SIGNAL(done), EXIT_TP();

  RESET(barrier);
  for (size_t i = 0; i < g_nCU; ++i) SYNC(compute[i]);
  EXIT_TP();
}
```

Listing 3.4: Coarse-Grain 5-Point 2D Stencil kernel—*DARTS* version. Compute *Codelet* and barrier *Codelet* definition. *Codelet*s reset themselves until the last iteration step is reached.

To reduce the overhead, a better version of the same coarse-grain behavior is provided in Listing 3.4. Adding `RESET` function to *Codelet* help reuse the same *TP* frame. The `SYNC` call allows a *Codelet* to signal a sibling contained within the same *TP* frame. In new version, `Compute` *Codelet*s reset their dependence count when they are fired. `Barrier` signals the end of the computation if there are no more time steps, or it resets itself, and then signals `Compute` *Codelet*s. This version is the "base" code we will be using to compare to *OpenMP* and refine in the section 3.3. Figure 3.1 illustrates this approach.

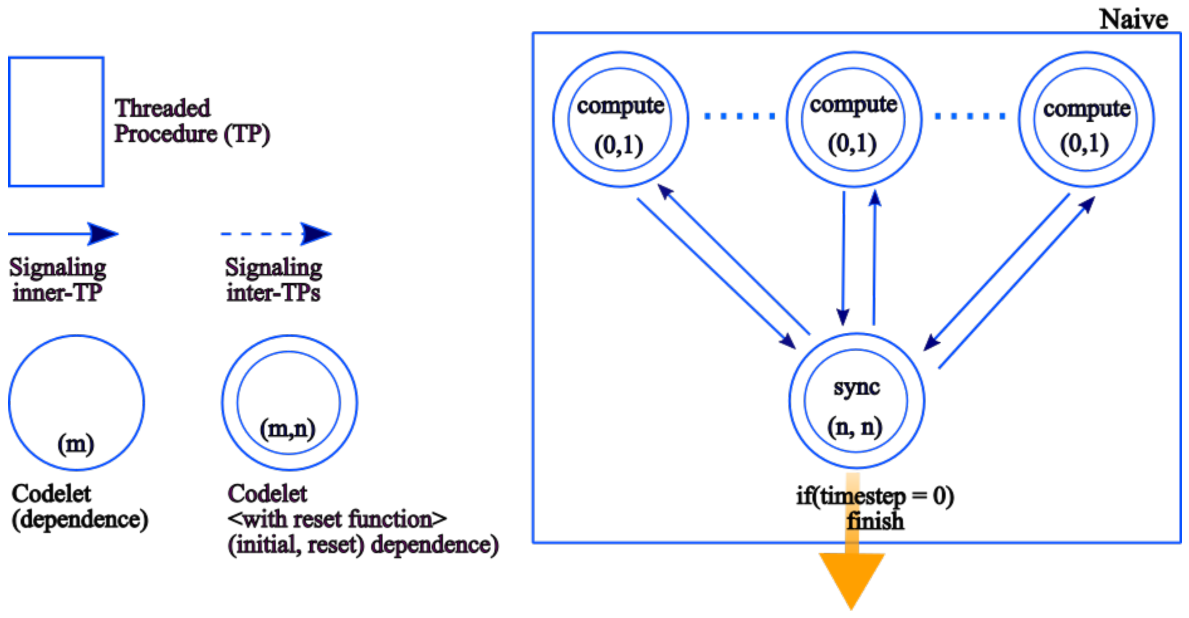

Figure 3.1: A Coarse-Grain Version of a Naïve Stencil Computation. Each codelet resets itself if there are remaining iteration steps.

3.2.2 Basic Implementation of a Parallel fine-Grain 5-Point 2D Stencil with *OpenMP*

The naïve *OpenMP* code leverages the regular coarse-grain fork-join execution model to parallelism code. As the loop is scheduled statically, the same *OpenMP* thread is tasked to process the same iteration chunk for each time step. Tiling method can be used to optimize the naïve code. As a result, even though as much asynchrony as possible was added to the code, there is still a need to issue a global barrier to wait between two time steps, to ensure each thread can start processing the new time step with the most up-to-date rows during the kernel's execution.

Figure 3.2 illustrates the fine-grain *OpenMP4.5*'s tasks Data Flow Graph(DFG). *OpenMP4.5* tasks directives, *i.e.*,`task depends: in`, `task depends:out` can help configure the connections between tasks. There are two types of tasks, `task-comp` and `task-scomp`. `task-comp` is a "regular" computing task, while `task-scomp` combines both a regular computation and a pointer swapping steps. `ck(i)` stands for $chunk_i$ (the i^{th} iteration chunk in the loop/block of matrix rows to process), and `ts(j)` stands for "time step j," the j^{th} time step in the iterative computation. The 2D stencil computation is partitioned into different chunks. Each chunk features the same number of columns and a similar amount of rows (the last chunk may feature a few more or a few less lines than the others, which mimics the way *OpenMP*'s parallel loops work). For each time step, there are different sets of dependencies to resolve, depending on which neighboring iteration chunks are being processed. Hence, `task-comp` for $chunk_i$ of time step j cannot begin computing until `task-scomps` of both $chunk_{i-1}$ (the "upper chunk") and $chunk_{i+2}$ (the "lower chunk") finish computing at time step $j-2$.

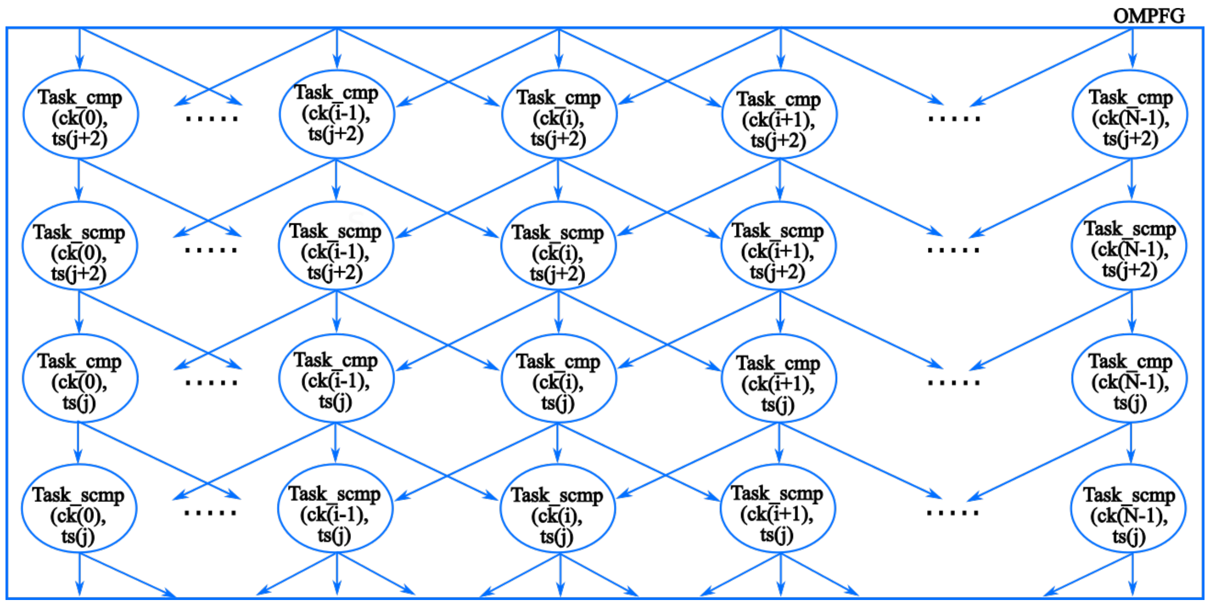
OMPFG
Task_cmp (ck(0), ts(j+2)
Task_cmp (ck(i-1), ts(j+2)
Task_cmp (ck(i), ts(j+2)
Task_cmp (ck(i+1), ts(j+2)
Task_cmp (ck(N-1), ts(j+2)
Task_scmp (ck(0), ts(j+2)
Task_scmp (ck(i-1), ts(j+2)
Task_scmp (ck(i), ts(j+2)
Task_scmp (ck(i+1), ts(j+2)
Task_scmp (ck(N-1), ts(j+2)
Task_cmp (ck(0), ts(j)
Task_cmp (ck(i-1), ts(j)
Task_cmp (ck(i), ts(j)
Task_cmp (ck(i+1), ts(j)
Task_cmp (ck(N-1), ts(j)
Task_scmp (ck(0), ts(j)
Task_scmp (ck(i-1), ts(j)
Task_scmp (ck(i), ts(j)
Task_scmp (ck(i+1), ts(j)
Task_scmp (ck(N-1), ts(j)

Figure 3.2: A Fine-Grain Version of 2D Stencil Computation with *OpenMP*.

3.2.3 Parallel Stencil Computations Using the *Codelet* Model

This section presents the various steps which were followed to produce a parallel fine-grained version of the 5-point 2D stencil code in the *Codelet* Model. *DARTS* explicitly specify *how* parallelism is created, orchestrated, and ended. It is necessary for fine-grain synchronization control including creating *Codelets* data dependencies and scheduling *Codelets* on specific threads.

3.2.3.1 Distributing The Computation Over Multiple Clusters In The *Codelet Abstract Machine*

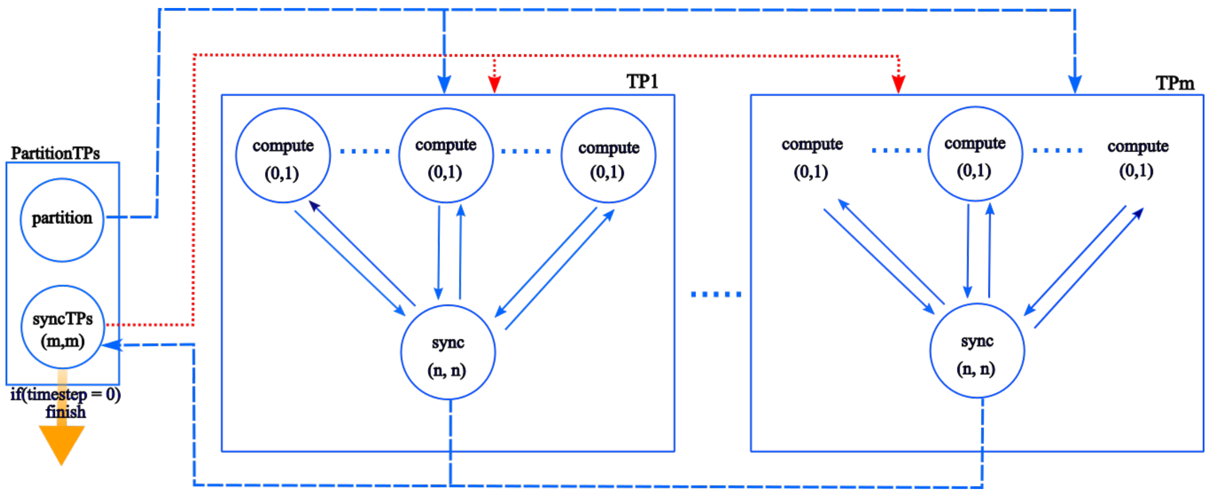

Figure 3.3: A Medium-Grain Version of a Naive 5-Point 2D Stencil Computation. The computation is decomposed into several sub-*Codelet* graphs, allowing a machine to hold multiple synchronization units for a better workload balance.

As described in section 2, *DARTS*, by default, maps each single socket to a *CAM*'s cluster of cores comprising one `SU` as control element and a serial of CUs as processing elements. As a consequence, parallelism is inherently hierarchical in this setting. Programming *Codelet* applications thus leads to building "natural" hierarchical barriers. The new naïve version with `RESET` function, shown in Listing 3.4 and Figure 3.1, is faithfully implementing one SU *CAM*. However, this configuration centralizes all *Codelet* graph creations onto a single processing element. This has several drawbacks. First among them, it effectively forces all cores to issue an atomic operation on the same memory location, thus forcing the serialization signals when a time step has been achieved. Second, it prevents the system from performing load-balancing when needed. To ensure a better load-balancing on a multi-socket shared-memory node, it is preferable to map multiple clusters from the *CAM*, each with its own SU.

The way shown in Listing 3.3 partitions the *Codelet* graph into sub-graphs, each contained within its own *TP*, and each confined to a given cluster of cores, hence maintaining locality. To avoid paying the overhead cost of dynamically allocating and deallocating *Codelet*s array when create new *TP* in each time step, mentioned in section 3.2.1, the same array of *Codelet*s is passed from invocation to invocation: the *Codelet*s are created only once the first iteration step has been started and destroyed only once the last iteration step has been reached. Figure 3.3 provides a high-level view of the resulting *codelet* graph. This results in a somewhat *medium-grained* version of the stencil computation, shown in section 3.3.

3.2.3.2 Toward a Finer-Grain Approach

The goal is to allow portions of work to proceed with the next iteration step, as long as the shared rows they require to update their portion of the matrix are available. To make it simple, new version of code is still still decomposing the work along the rows of the matrices. However, each *Codelet* simply signals its neighbors when it is done updating the rows they depend on to move to the next iteration step. Hence, some *Codelet*s may proceed to update the system at step S_{t+1} while others are still finishing step S_t. Figure 3.4 provides a diagram of the resulting *Codelet* graph where only one *TP* is created to hold the whole *Codelet* graph, where all dependencies are statically determined. The stress on the memory subsystem is not expected to be excessive, however, since signals are now only sent between "neighboring" cores.

3.2.3.3 Reducing the Stencil Computation's Footprint

To reduce the memory footprint of the computation, instead of systematically using two matrices to iterative compute new values at each time step (subsequently requiring to exchange array pointers), it is possible to allocate a small buffer per *Codelet* in each invoked *TP*. Each buffer must be large enough to hold a set of at least three full rows in the matrix. As a result, The original naive loop thus becomes more complex, as each *Codelet* must

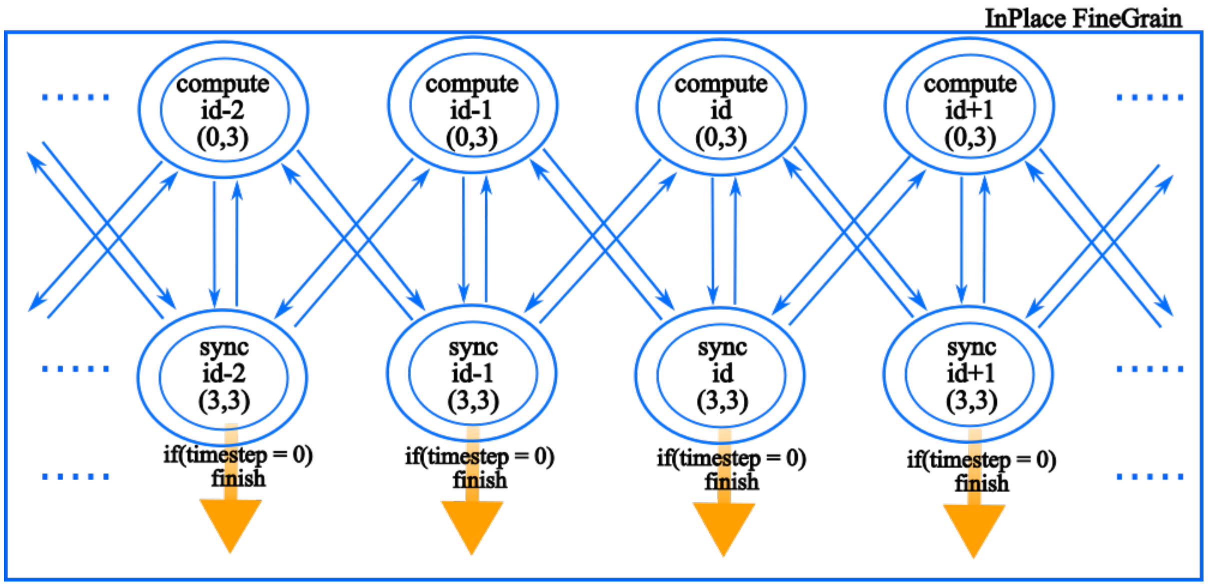

Figure 3.4: A Fine-Grain Version of a Naive 5-point 2D Stencil Computation. A single *TP* is generated, which holds the full *Codelet* graph. *Codelets* only signal the neighbors which read and write shared rows.

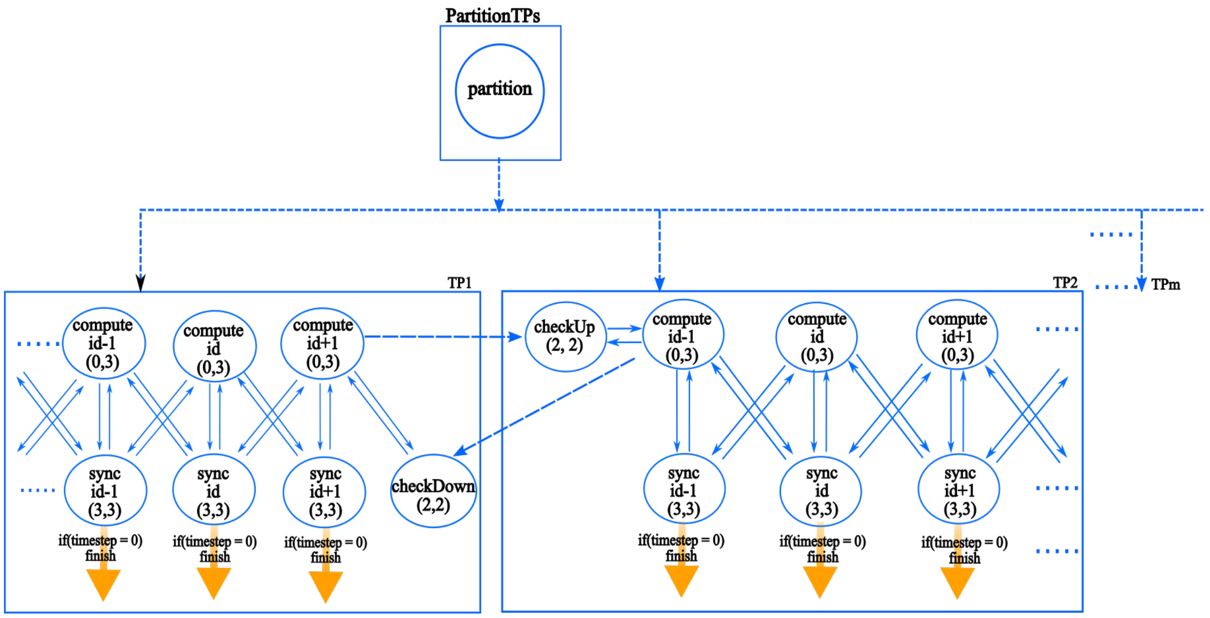

Figure 3.5: A Fine-Grain In-Place Version of a Naïve 2D Stencil Computation. Multiple *TPs* can be generated, which hold a portion of the overall *Codelet* graph. *Codelets* only signal the neighbors which read and write shared rows. A single matrix is required.

now first write the new values of the system to its local buffer first, then must write the newly updated row(s) back to the original matrix. However, this scheme lends itself well to fine-grain synchronization.Indeed, as Figure 3.4 only features *TPs*, *Codelets*, and their dependencies, but not the actual code or data that are held in the *TP* frames, then it is also an adequate representation of an "in-place" version of a fine-grain version of an n-point stencil computation. However, this version suffers from the same limitation as the previous fine-grain variant: it requires to invoke a single *TP*, thus forcing the *CAM* to be mapped with a single SU for the whole machine, and, in turn, to accept that all *TP* creations will involve a potentially heavy serial step. Note also that since this implementation requires to allocate enough space for three full rows of the original matrix, there is no guarantee these buffers will fit in individual core's L1 data caches, or even L2 caches.

Hence, a final refinement is to allow for the distribution of the fine-grain "in-place" variant over multiple *TP*s. While the previous variants, including the initial fine-grain one, were relatively easy to implement, this specific implementation requires some careful coding when setting up the overall *Codelet* graph, as *Codelet*s will reset themselves and signal each other not only within the same *TP* frame, but also *across* frames. However, the basic structure remains the same, and it clearly can be automated by a compiler. The resulting *Codelet* graph is shown in Figure 3.5. In this last variant, each *Codelet* graph features three types of *Codelet*s: `Compute` performs the actual computation, as before. The `CheckDown` and `CheckUp` *Codelet*s are signaled when rows shared by "upper" and/or "lower" neighbors are ready to be updated. In turn, they also signal other compute *Codelet*s to let them know that the rows they are sharing with their neighbors are cleared for reading. Note we elected to partition the matrices row-wise to keep the case study simple, but further partitioning (along both rows and columns) would follow the same principles.

3.2.4 A More Realistic Stencil-based Computation: LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics, or LULESH, is a "proxy app," *i.e.*, an application that is representative of a more complete and more complex type of application currently in use in national laboratories all over the world and in particular in the US. Specifically, such applications are used to model deformation events, and in the particular case of LULESH, the Sedov blast wave problem for one material in three dimensions. It is a hexahedral mesh-based physics code with two centerings, Nodal centering and element centering, and time simulation constraints [27, 29]. As with all other "proxy" (or "mini") applications published by US national laboratories, the implementation prioritizes clarity over optimization. As a result, while the application is representative of more complex (and more complete) shock hydrodynamics currently deployed in production in various laboratories, *e.g.*, in terms of computation steps, data movements, *etc.*, it lacks many of the optimization that can be found in production-level implementations.

Well known code transformations, such as tiling, loop fusion, or loop distribution, is not the purpose of this section. The impact of transitioning from a coarse-grain implementation of the code toward a fine-grain one, and, in the context of this mini-app, how much performance can be hoped to be gained when "drowned" within a more complex application environment is the interesting part.

The "official" LULESH application [29] uses *OpenMP*'s coarse-grain synchronization constructs, as shown in Figure 3.6. For example, the `Nodal` centering function, one of most time consuming functions of LULESH, copes with all nodes' kinematics value such as force, acceleration, positions, velocities *etc.* The synchronization barrier will be used for every kinematics value calculation. The synchronization cost is proportional to the number of nodes.

To avoid global barrier, a tree barrier approach can be used to control synchronization granularity. The tree can be balanced or unbalanced for a given arity k. The tree structure barrier impact the overall performance by reducing atomic operations in the overall computation. For instance, there are 8 processing elements in the hardware platform, if the arity of each node is set to 2 which ensures a balanced tree, as shown in Figure 3.7, then every

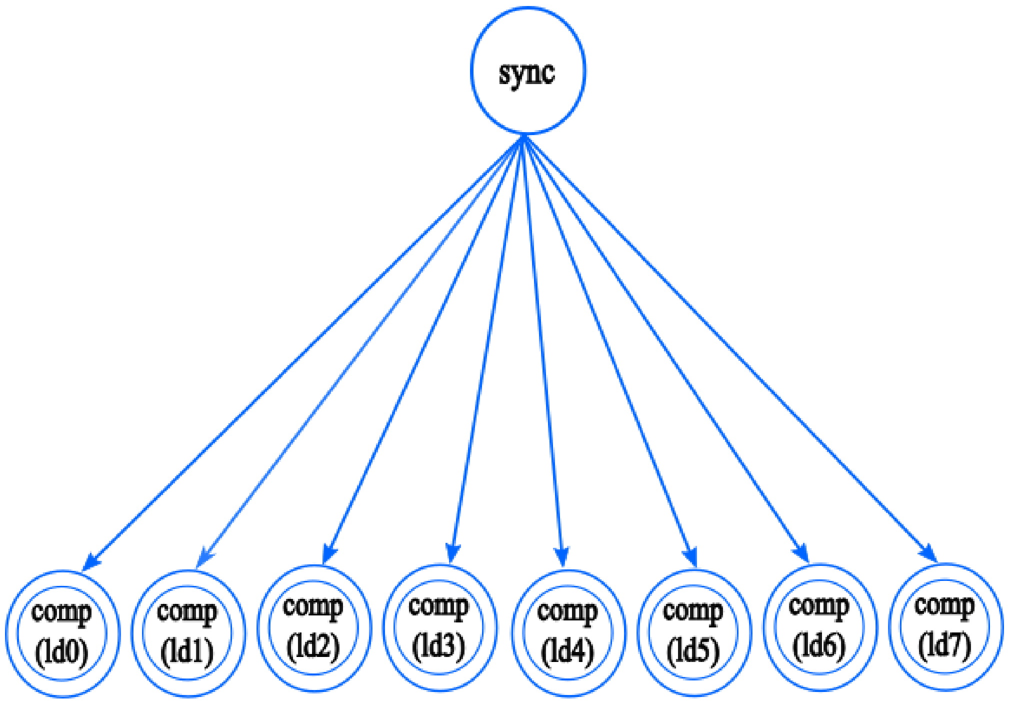

Figure 3.6: LULESH Compute-Sync Graph. – *OpenMP* version – Coarse Grain

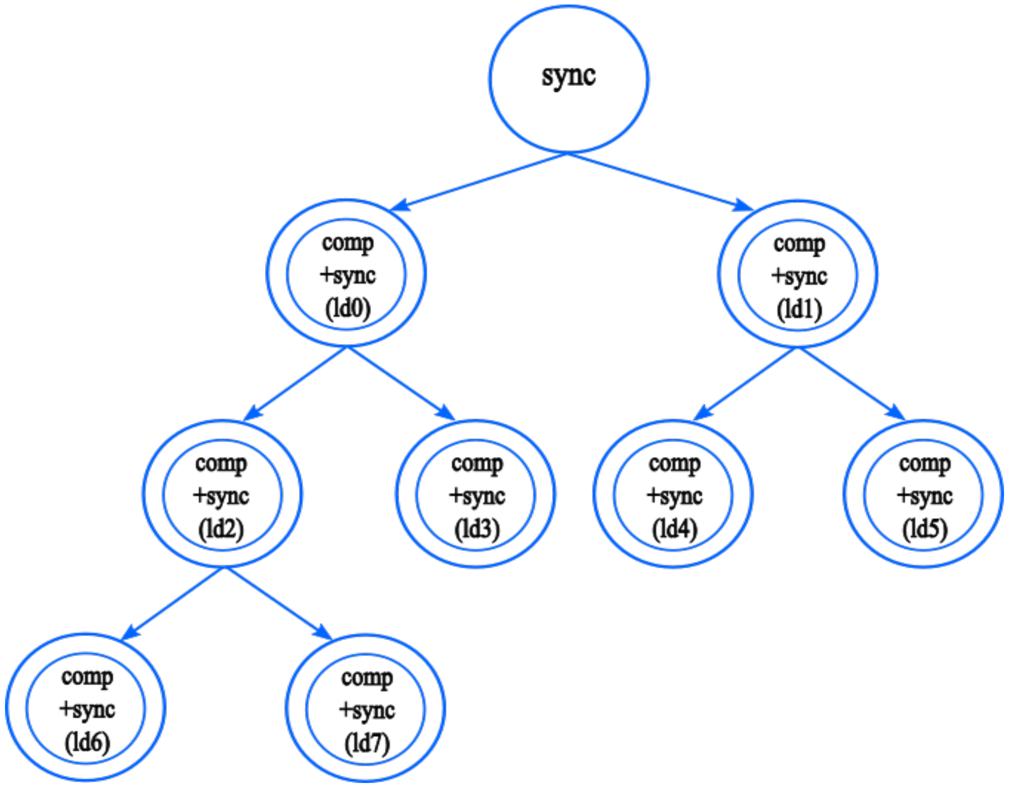

Figure 3.7: LULESH Compute-Sync Graph. – *DARTS* version, – Balanced Compute-Sync Tree

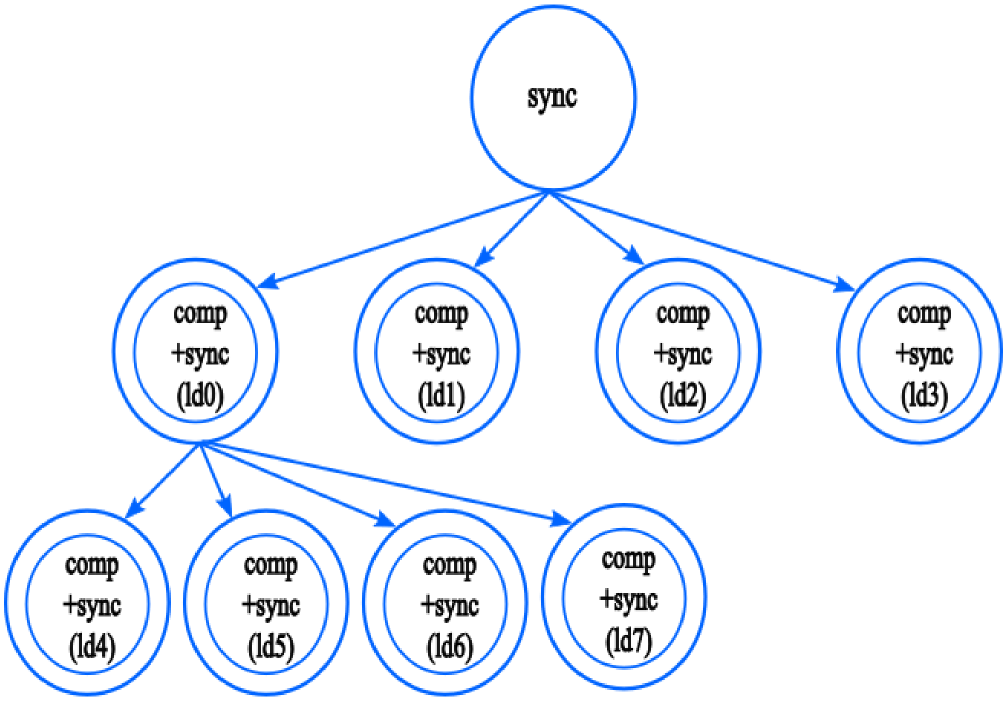

Figure 3.8: LULESH Compute-Sync Graph. – *DARTS* version, – Unbalanced Compute-Sync Tree

inner code will have 2 children; if the arity of each name is set to 4, as shown in Figure 3.7, then this tree is unbalanced. There is no standard criteria to determine which type of tree is more efficient. The hardware,especially the number of available cores in one cluster, the structure of clusters and the memory hierarchy, affect the finally performance. In this tree structure, every *Codelet* fulfills two functions: computation and synchronization, excepted the root *Codelet*, which only performs synchronizations.

3.3 Experiments

This section describes the results of each specific computation—naïve coarse-grain 2D stencil, fine-grain 2D stencil, and LULESH—implemented using *OpenMP* and *DARTS*. We compare the results obtained in each case.

3.3.1 Experimental Testbed

Platform	Processor type	# Sockets	# PEs per Socket	Total PEs	L1D (KiB)	L2 (KiB)	L3 (MiB)	Comments
A	Intel Sandy Bridge	2	16	32	32	256	20	Private L2 ; Hyperthreading
B	Intel Sandy Bridge	4	12	48	32	256	15	Private L2 ; Hyperthreading

Table 3.2: Compute Nodes Characteristics. "PE" = "Processing element." L2 and L3 caches are all unified. Hyperthreaded cores feature two threads per core. Platform A features 64 GB of DRAM; Platform B features 128 GB.

Platform	Linux distribution	Kernel version	GCC version
A	CentOS 7.1	3.10.0	8.1
B	Ubuntu 16.04 LTS	3.13.0	8.1

Table 3.3: System Software Stack used for the experiments.

The hardware platforms characteristics are described in Table 3.2. Table 3.3 provides the information related to the system software running on each compute node.Each platform offers a relatively varied system software layer, with compilers and OS kernels being slightly (or even widely) different from node to node. All experiments are run by pinning threads to a given processing element, and by setting the `OMP_PROC_BIND` environment variable to `true` (for *OpenMP*). *DARTS* automatically pins its work queues to the underlying processing elements. [1]

3.3.2 Experimental Protocol

Eight different variants of stencil code: `Seq` is the baseline and is a benchmark that runs sequentially on one CPU core; `OpenMP` runs the same code as `Seq` with added *OpenMP* directives running on all the available CPU cores; `Naïve` is a single *TP* implementation of the stencil computation, both `OpenMP` and `Naïve` are described in Section 3.2.1); `OpenMPFG` is

[1] All the code is available on the Git repository: `https://github.com/gengtsh/darts-heterogeneous`.

the fine-grain variant which makes use of the tasking mechanisms available in *OpenMP4.5*, described in section 3.2.2; `NaïveTPsPtr` implements the same logic as `Naïve`, but distributes the work across several *TP*s, illustrated in Figure 3.3; `FineGrain` implements the fine-grain synchronization scheme and illustrated in Figure 3.4; `InPlace`'s *Codelet* graph is identical to `FineGrain`'s. However, `FineGrain` allocates two (dst and src) matrices, while `InPlace` only allocates one matrix and a small 3-row buffer within a compute *Codelet*. Hence, the synchronization logic of `InPlace` is more complex than `FineGrain`'s. `InPlaceTPs` implements the same in-place variant, but distributes the computation across multiple *TP*s, described in Section 3.2.3.3 and illustrated in Figure 3.5. To summarize, `OpenMP` and `Naïve` implement coarse-grain synchronization scheme, `NaïveTPsPtr` and `InPlaceTPs` implement medium-grain synchronization scheme, and `FineGrain` and `InPlace` implement Fine-Grain synchronization scheme.

The experiments utilize the following protocol:

1. All 2D stencil computations run for 30 time steps
2. Each variant instance is run 20 times to increase the stability of the run, then the accumulated times are averaged after removing the 2 most extreme values (min and max).
3. Each binary containing a variant is run 10 times from the command line, and average the accumulated times once again.

The reason why computing the the average of different invocations of the binaries for each variant is because the overall system environment introduces enough noise to generate timings that can significantly differ, for sequential, *CAM*, as well as *OpenMP* model variants—especially for smaller input sizes.

LULESH experiments were done in a similar way as 2D Stencil: All the computations were repeated 20 times and the whole executable were run 10 times each.

3.3.3 Experiment Results — 5-Point 2D Stencil Kernel

The results for strong scaling are shown in Figures: 3.9, 3.10, 3.11, 3.12, 3.13, and 3.14. Since there are eight different variants, as described in section 3.3.2, to maintain the readability of the Figures, only `Seq`, `OpenMP`, `OpenMPFG` and two best *DARTS* variants will be shown in these Figures.

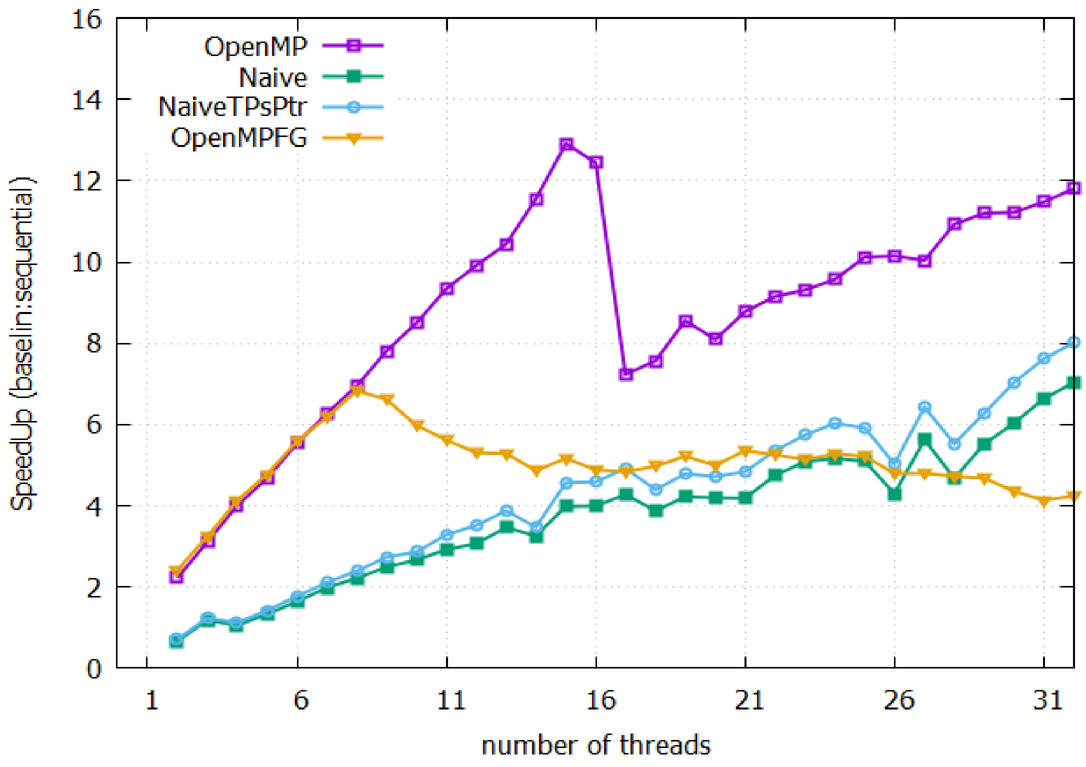

Figure 3.9: platform A: Strong Scaling– Matrix size: 1000 × 1000.

The default CAM is used in the case of DARTS: each socket of the target platform is mapped to a cluster of cores. Each cluster thus features $n-1$ Compute Units (`CU`s) and one Scheduling Unit (`SU`). Hence all `CU`s are physically close to each other, ensuring that a *TP* allocated to a cluster displays some level of locality (at least at the L3 cache level). In other words, `compact` mapping polices, allocating software threads as closely as possible on the available

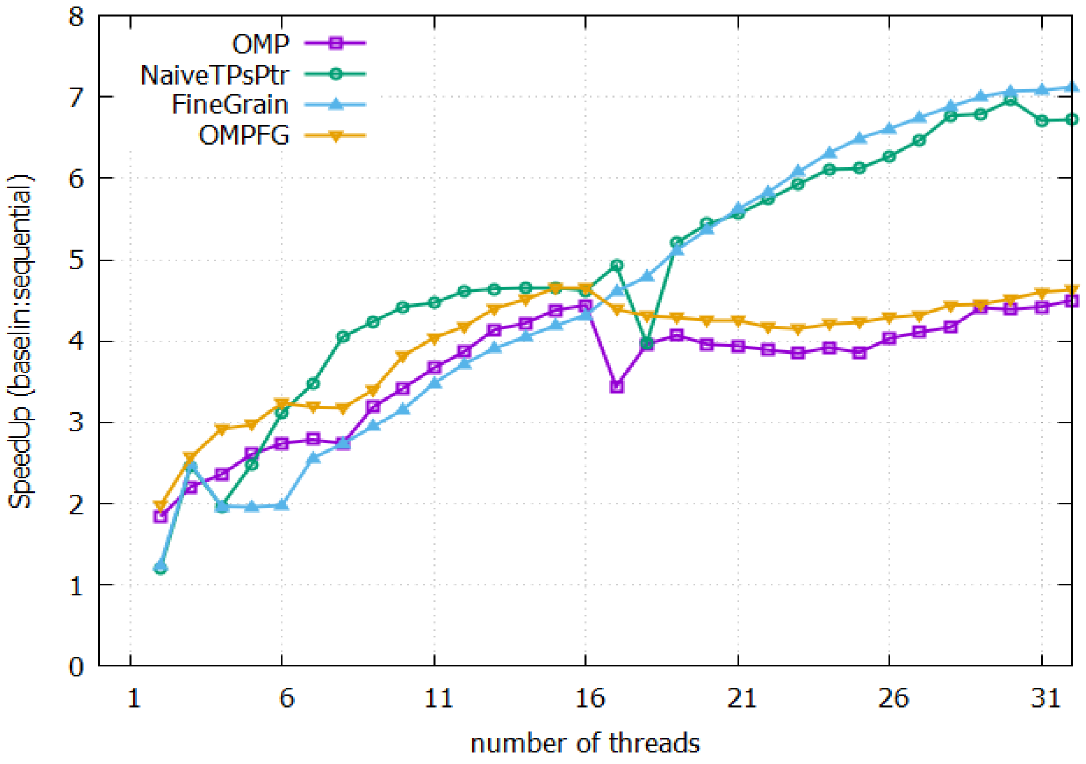

Figure 3.10: platform A: Strong Scaling– Matrix size: 3000 × 3000.

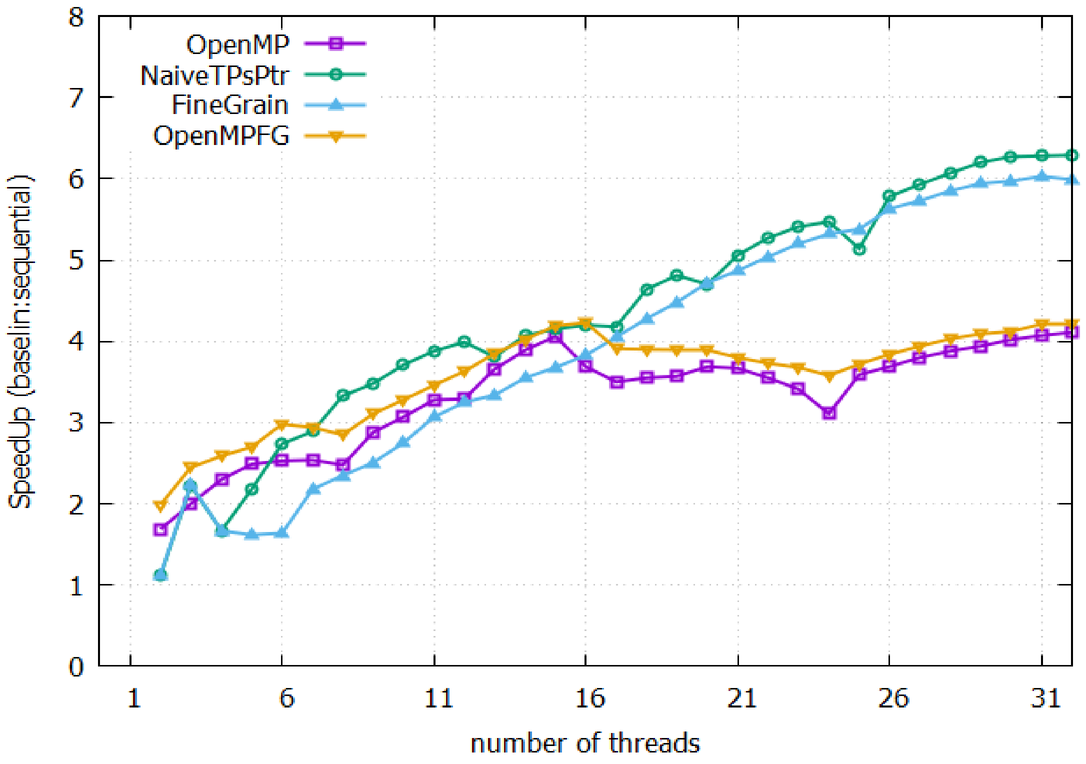

Figure 3.11: platform A: Strong Scaling– Matrix size: 5000 × 5000.

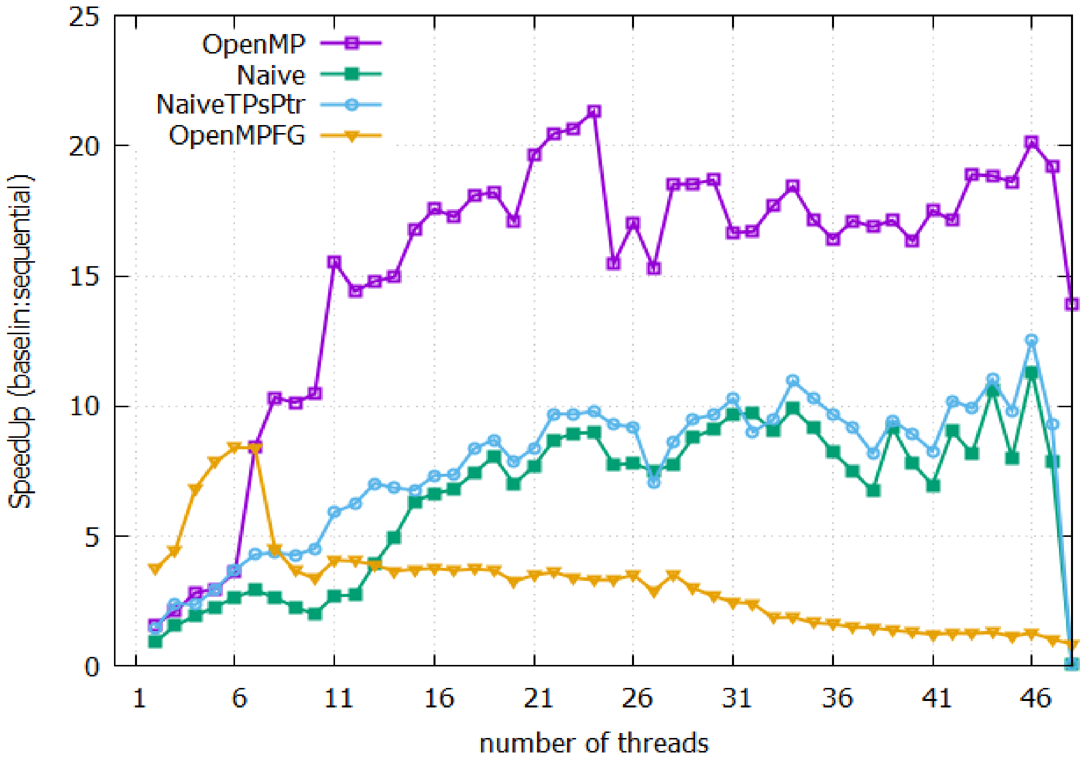

Figure 3.12: PlatformB: Strong Scaling– Matrix size: 1000 × 1000.

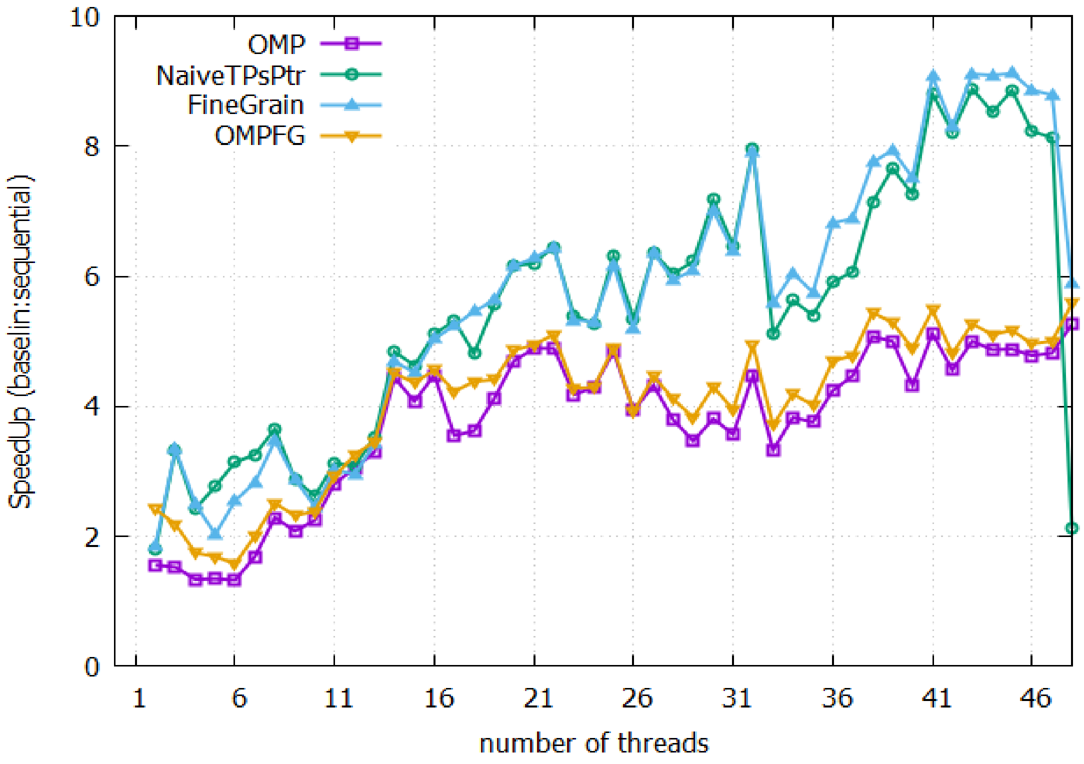

Figure 3.13: PlatformB: Strong Scaling– Matrix size: 3000 × 3000.

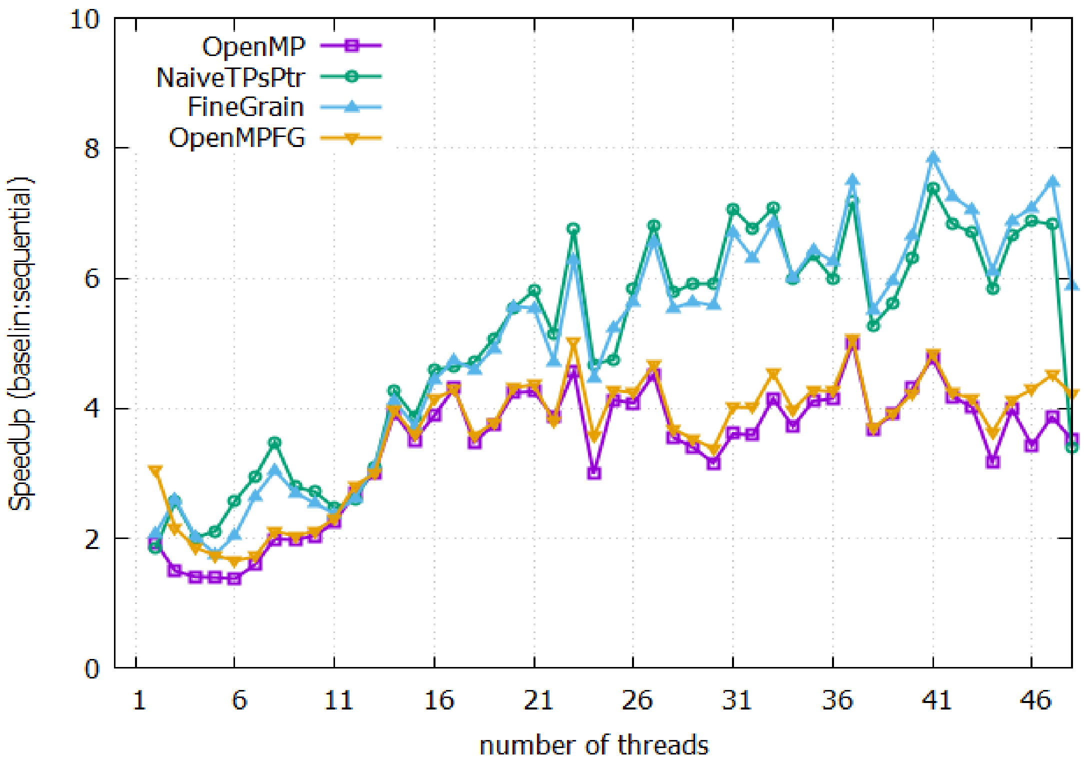

Figure 3.14: PlatformB: Strong Scaling– Matrix size: 5000 × 5000.

Processing Elements (PEs) according to the underlying physical topology, is used in strong scaling test. As a result, in low-CU count case, not all the available aggregated cache will be used, especially, on a on a 2-socket compute node, if less than half of processing element(*i.e.*, one cluster/socket), then only one L3 module will be utilized.As described in section 3.3.2, the average execution time will be used as the final result since the execution times followed a normal distribution. For workloads that were mostly memory-bound, on Platform A, the standard deviation using *DARTS* is at most 5%, and less than 1% on average. On Platform B, the highest standard deviation reaches 18%, with an average of 10%.For cache-bound workloads, the standard deviation is much higher. For example, for 1000 × 1000 matrices, the standard deviation reaches 11% on Platform A (with an average of 5%), and 27% (with an average of 24%) on platform B. This is in part due to the dynamic scheduling algorithm which are used in the *DARTS*, which cannot guarantee that the same chunk of data will be processed by the same processing element.

In the *OpenMP* case, for both `OpenMP` and `OpenMPFG`, `OMP_PROC_BIND` are set to true to ensure that threads are pinned to a given PE. However, the *OpenMP* run-time system and the underlying OS are in charge of assigning a given (*OpenMP*) thread to the physical PEs, which results in threads being able to use all available L3 caches (when distributed over several sockets). Still, when resources start to be saturated, *i.e.*, when more than half of the processing elements (which in the case of both platforms are hardware threads) are used, and when they start to compete for FPUs, caches, *etc.*, the *DARTS* variants outperform the OpenMP version. As the PE count increases, so does the performance gap, as shown in Figure 3.10, 3.11, 3.13 and 3.14.In Platform A, figure 3.9 shows that the *OpenMP* coarse-grain variant has a clear advantage over *DARTS* and `OpenMPFG` fine-grain when the workload fits in the caches (*i.e.*, when the matrix size is 1000, or possibly 2000, as it still partially fits in the caches). Once the data grow beyond the capacity of L3 caches, as shown in Figure 3.10 and 3.11, *DARTS* Medium and Fine-Grain variants get the upper hand, and the `OpenMPFG` yields slightly better performance than `OpenMP`. The same trend can be found in Platform-B (Figure 3.12, 3.13 3.14).

In weak scaling, shown in Figure 3.15 and Figure 3.16, `FineGrain` and `NaiveTPsPtr` achieve the best performance, with speedups reaching up to 3× compared to two *OpenMP* variants. The `OpenMP` has a clear advantage over *DARTS* when the workload fits in the caches (*i.e.*, when the matrix size is 1000×1000 on Platform A, or 1000×1000 and 2000×2000 on Platform B, as it still partially fits in the L3 caches). In the *OpenMP* coarse-grain case , loops are statically scheduled, thus ensuring that the same PE processes the same chunk of data, and hence minimizing cache misses. In the *OpenMP* fine-grain case(`OpenMPFG`), all the tasks, as described in Figure 3.2, are in the tasks pool, and will be invoked when their dependencies are satisfied. Tasks are dynamically assigned to available PEs and cluster. In contrast, *DARTS*'s scheduling policy is fully dynamic, and thus *Codelet*s can be run by any PE belonging to the same cluster of cores. Hence a given data chunk may be processed by different PEs over two successive iteration steps, resulting in additional cache trashing.

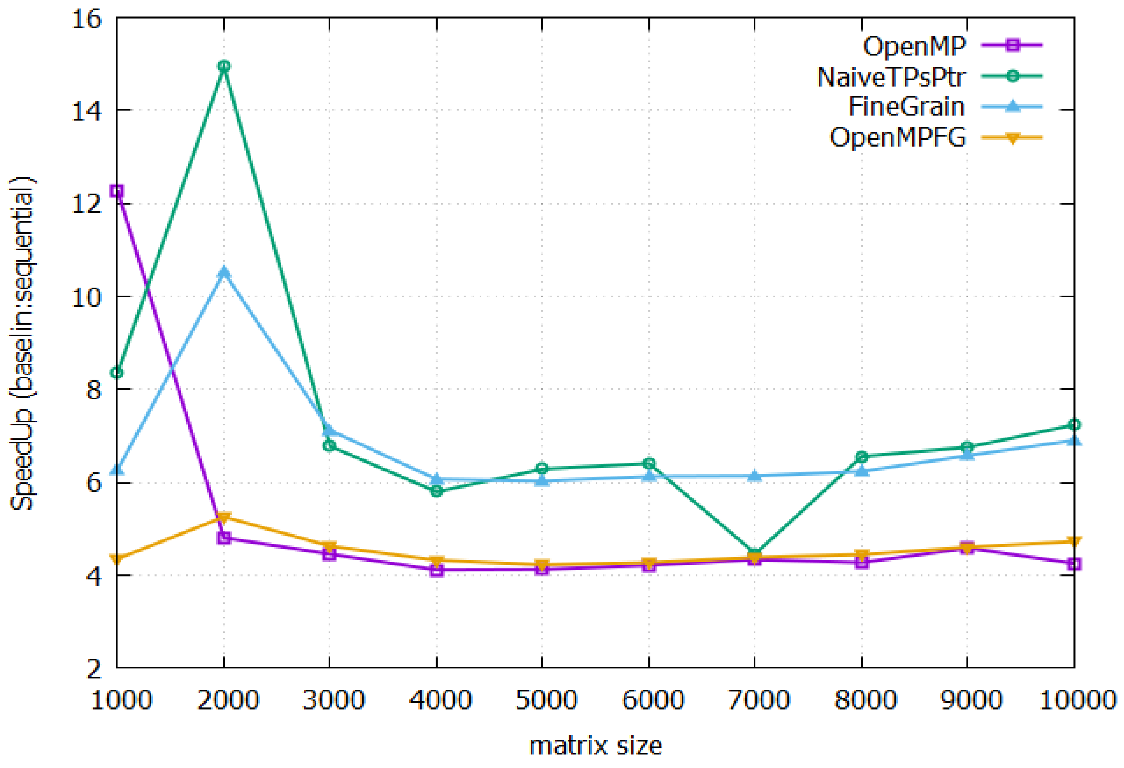

Figure 3.15: Platform A: Weak Scaling– Thread Number: 32

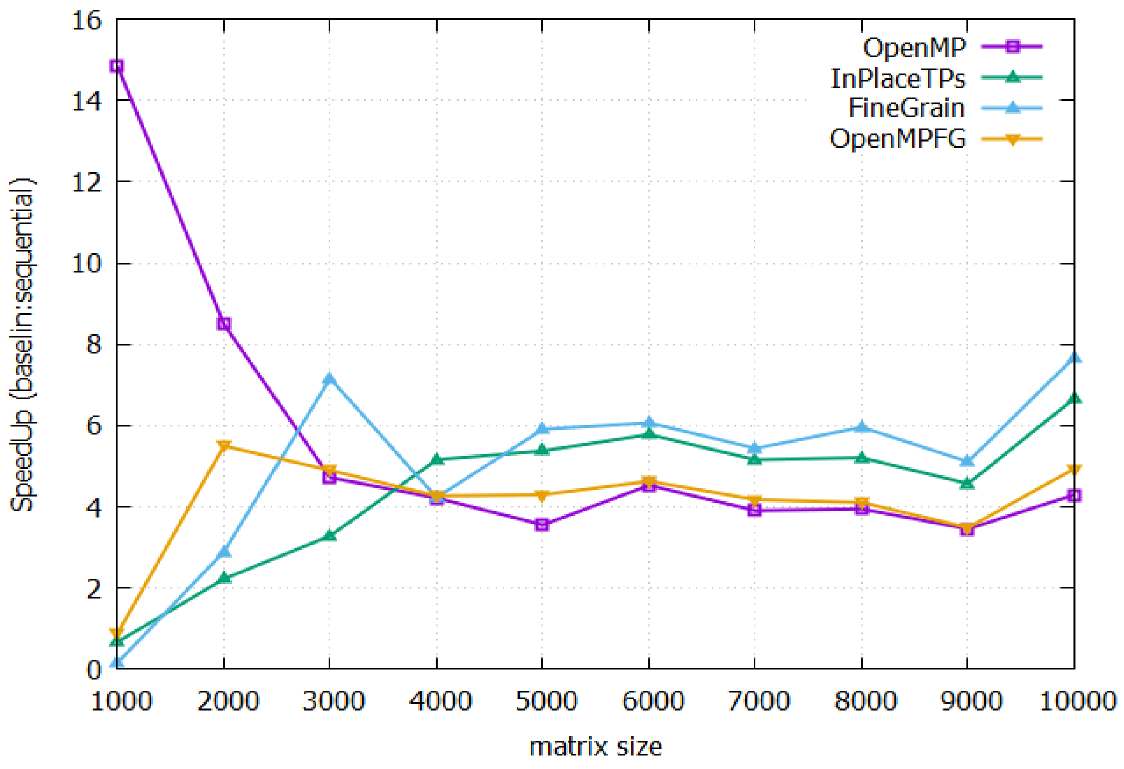

Figure 3.16: Platform B: Weak Scaling–Thread Number: 48

Once the data grows beyond the capacity of L3 caches, *DARTS* gets the upper hand: the finer-grain variants either issue "local" atomic operations between neighbors (as with the `FineGrain` variant), or at least provide a hierarchical way to maintain some locality within their cluster of cores, thus reducing the overall memory traffic. In particular, the inclusive nature of the caches in Intel processors allows the hardware to recognize when a given memory location is owned by the "local" L3, and thus avoids a costly request for ownership across sockets. Fine-grain *DARTS* variants are always better than coarse grain ones. When problem sizes fit in the L3 cache(s), the *OpenMP* variant yield much better performance than all the *DARTS* variants, no matter the granularity. When problem sizes are larger than the L3 cache, the *DARTS*'s `FineGrain` and `NaiveTPsPtr` variants yield better performance.

3.3.4 5-Point 2D Stencil Kernel Results — Discussion

Coarse-grain synchronizations (*e.g.*, barriers) tend to be implemented with a single memory location, even in state-of-the-art run-time systems (for example: *GCC*'s *OpenMP*; Intel's implementation offers both linear and tree-based barriers). This has several negative consequences: (1) all processing elements issue an atomic operation to the same location, forcing the other PEs to flush their write buffers, sometimes more than once; (2) there is a "natural" contention due to the target single location. By contrast, finer-grain synchronization makes use of more locations with better locality effects. Write buffer flushes still occur, but tend to be limited to writing back in L3 (at least in the Intel case). In addition, *Codelet*s can better exploit the "slack" that exists when a core is done running a thread, due to their event-driven nature.

Finer-grain synchronization clearly *does* provide better results on general-purpose many-core systems, as shown in Figures from 3.9 to 3.16. However, which variant works best varies significantly depending on which platform running tests. On Intel-based compute nodes, the most refined variants did not perform very well in the end: the `InPlace` and `InPlaceTPs` variants under performed compared to their most simple counterparts, and even compared to the coarse-grain *OpenMP* variant. The main reason is the implementation is too naïve: while the `InPlace` variant does require less memory than the original code, its implementation is too simplistic. When computation *Codelet*s are being fired, `OS` will allocate *Codelet*s to different Hard threads. However, when a huge amount of *Codelet*s are fired within a very small time range, some serialization operations, such as accessing OS's "memory allocator" , will drop the overall performance.As Intel-based nodes feature inclusive caches, the data can only be as big as the L3s of the system. The situation maybe different, if the experiment run on the AMD-based Processor since AMD system cache are exclusive: the aggregated size of the L2 caches equals the aggregated size of the L3s, effectively doubling the overall size of the data that can be held in the caches.The AMD system also relies on write-through L1D caches (compared to Intel's write-back L1Ds), which allows for a better utilization of the L1D (there is roughly four times more reads than writes in the stencil computation)

Moreover, the purpose of this section is to show the benefits of "pure" fine-grain synchronization, without resorting to classical loop transformations such as tiling or loop skewing, even the allocation of just three complete rows is enough to quickly fill L1D caches. For example, the smallest input size for a matrix, 1000×1000, requires three rows of a thousand elements to implement the current in-place variants. However, this represents already $\approx 2/3$ of the L1D cache of the compute nodes. Hence, to obtain an efficient in-place variant, additional blocking and tiling techniques need to be applied.

3.3.5 Experiment Results — LULESH

Similarly to the 5-points 2D stencil kernel, the average execution time is used as final result since the execution times followed a normal distribution. The largest standard deviation was

2% for Platform A, and 1% for Platform B.

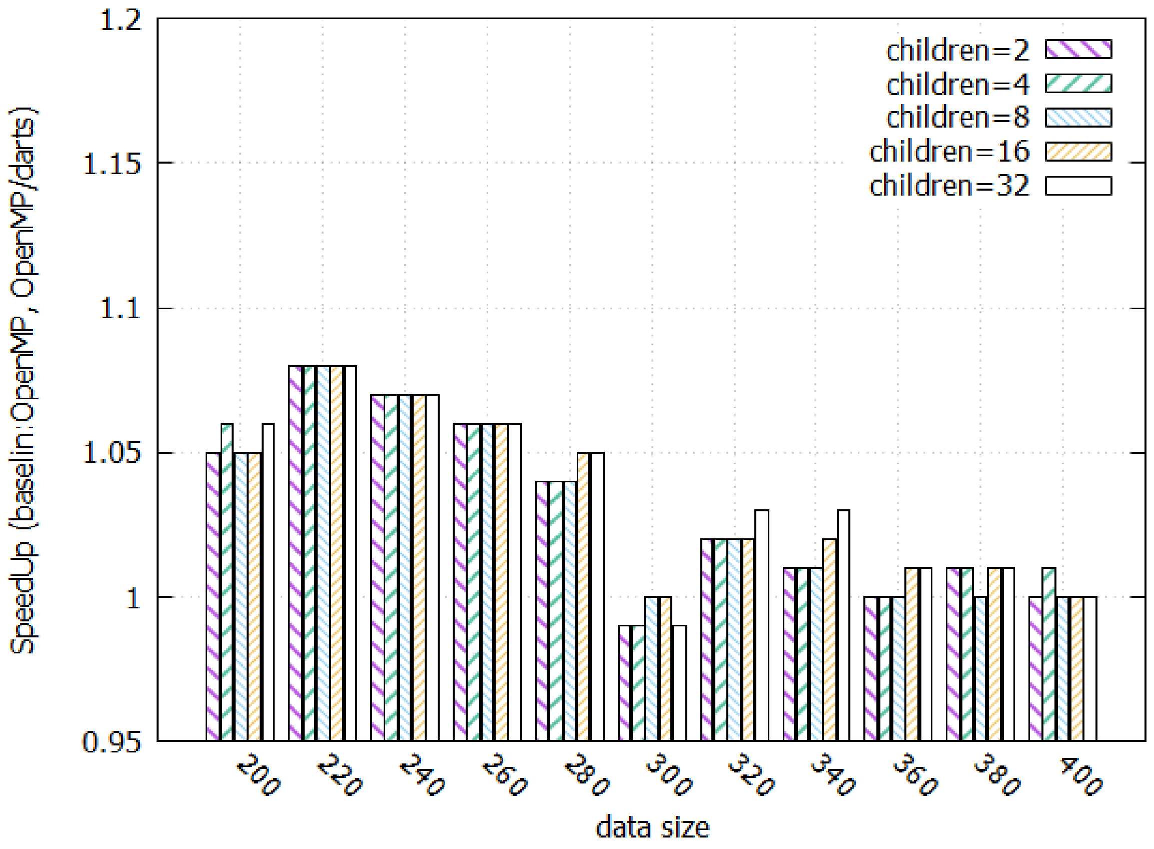

Figure 3.17: Platform A: LULESH on *DARTS*, vs *OpenMP*, children n is the arity of each node in the tree, *i.e.*, the number of children a node can have in the tree.

The overall performance of `LULESH` on *DARTS*, relies on three parts: the underlying computer architecture (in particular the memory aspects), the synchronization granularity, and input data size. As shown in Figure 3.18 (Platform B – see Table 3.2), compared to the reference *OpenMP* implementation, which uses coarse grain synchronization, the *DARTS*, Medium-Grain synchronization variant gets relatively good performance for small data sizes. For instance, when the input size is less than 320^3, *i.e.*, the resolution of 3D LULESH is either over 320^3 elements or 321^3 nodes per time step. Medium grain variants (where the arity of each node is denoted by $children = 6$ and $children = 12$) fare relatively better than the coarse grain version (*i.e.*, $children = 24$ and $children = 48$) and the *OpenMP* reference code, but not by much. The main reason is that data fits in the various L3 caches, which

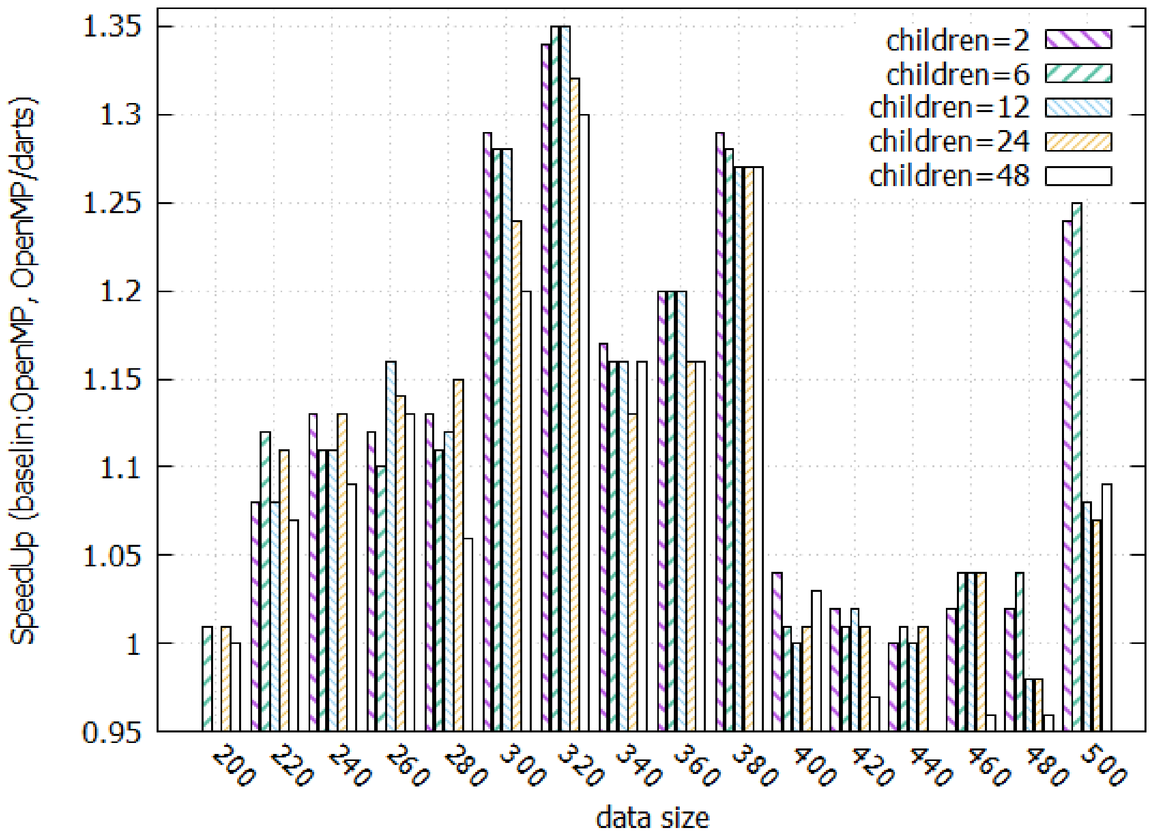

Figure 3.18: Platform B:LULESH on *DARTS*, vs *OpenMP*, children n is the arity of each node in the tree, *i.e.*, the number of children a node can have in the tree.

then allows for all synchronizations to occur relatively seamlessly.

For larger matrix sizes, Fine-Grain variants (with $children = 2$, which builds a binary computational tree, and $children = 6$) fare much better. For example, performance jumps by a wide margin when the data size reaches 500^3. This is in part due to the fact that when the data set size increases, each individual *Codelet* has more work to perform, so ratio of computation-communication cost increase while communication belongs to serialization part and computation belongs to paralleling part. Furthermore, the *Codelet* graph builds a fine grain synchronization tree, as shown in Figures 3.7 and 3.8. In it, the non-root, non-terminal *Codelets* have two functions: computation and synchronization. The synchronization tree structure can help split this large data set into a series of small and relatively independent ones. The amount of computation is the same among all *Codelets*, but the communication cost will be reduced compared to a coarse-grain variant leveraging a barrier. Because children *Codelet*s only need to communicate/synchronize with their parent, data access conflicts are reduced within one cluster and between sockets/clusters, especially in the case that the computation spreads across all the PEs and all the clusters, fully utilizes the L3 cache, and needs to access data in main memory. Hence, the tree structure helps control the data flow, data transfers, and further reduces data access conflicts. For medium-sized computations, the situation is more complex. When the data set size reaches the L2 cache boundary, *e.g.*, data sizes from 400^3 to 480^3 as shown in Figure 3.18 (Platform B), the overall performance will rely on multiple factors, such as how *Codelet*s were bound to cores or clusters, whether the leaves and their parent are allocated in the same socket or not, how deep of tree structure was, *etc.* For this data set, the best granularity cannot be easily predicted, and some fine-tuning is in order.

DARTS assign *Codelet*s to different PEs and clusters by using the `hwloc` library and bind units to specific cores using their ID. In this experiments, the binding method is based on the granularity of synchronization and number of cores in one cluster. The basic rule is to try to put parent and children *Codelet*s in one cluster/socket to reduce the data transformation time.

In the Figure 3.17 and 3.18, there are some special points, called *changing points*, which correspond to the medium sizes always somewhere “just above” the L3 cache sizes. The boundary between coarse-grain and fine-grain is vague since the cost of tree hierarchy communication and coarse-grain communication are similar during these changing cost. Different architectures have different changing points. For example, in Platform A (see table 3.2 and Figure 3.17), the changing point occurs at data size 300^3, but in Platform B, the changing points are range from 400^3 to 480^3. This is of course directly related to the sizes of the platforms’ L3 size (individual and aggregated), as well as the way the workload is partitioned among the PEs.

3.4 Related Work

This section presents work pertaining to fine-grain and event-driven multithreading, as well as relevant approaches to run parallel executions of stencil computations.

3.4.1 Fine-Grain Multithreading Execution and Programming Models

In recent years, several attempts at providing more dynamic ways to create parallel work have been proposed. Many such attempts are inspired by data flow models of computation. Among them, we can mention Concurrent Collection [30–33], which implements a dynamic data flow inspired execution and programming model to orchestrate parallel programs execution. Cnc was used to run workloads that expose extremely fine-grain parallel algorithms, such as stencil computations in the LULESH application [34], using classical optimizations such as loop fusion and tiling to coarsen granularity and enhance the application's scalability. However, the authors lacked a cache-specific tuner and had to suffer a large overhead due to the data collection phase.

XKaapi [35, 36] is a macro-data flow run-time which targets multi and many core (possibly heterogeneous) systems. Much like most modern run-time systems (including *DARTS*), it relies on the use of work stealing for dynamic load-balancing, as well as work over-subscription to ensure the system is always usefully busy.

The Open Community Run-time [37] (OCR) system is an event-driven multithreading system in part inspired by the *Codelet* model. It runs on both shared and distributed memory systems, and requires the programmer to pass data and events through data blocks and event slots. Each data block or event-driven task is assigned a global unique identifier. While this approach may introduce additional complexities for the parallel programmer, especially in shared-memory systems, it results in a more seamless execution across computation nodes in a distributed memory system.

The SWift Adaptive Run time Machine (SWARM) system is another implementation of the *Codelet* model [38]. Just like *DARTS*, it is initially a run-time system with a hardware layer abstraction. It can run on both shared and distributed memory systems. However, it is not completely faithful to the initial *Codelet* model.

XKaapi, OCR, and SWARM all propose a fine-grain event-driven approach to multithreading. However, they do not provide an explicit way to group data flow tasks to ensure they execute on a specific portion of the hardware (for example, to maintain spacial and temporal locality), contrary to *DARTS* (which uses threaded procedures to enforce *Codelet* grouping). SWARM does get close to this concept by providing ways to "bind" *Codelet*s to a specific physical portion of the machine, in a manner similar to Hierarchical Tiled Array's locales [39]. As a result, most of these solutions tend to resort to very dynamic ways to spawn parallelism as a whole.

The Cilk programming and execution model [40] and its subsequent evolution, is a parallel programming language that favors fine-grain multithreading, and encourages a divide-and-conquer approach using a fully-strict approach to evaluate a program's computation graph, *i.e.*, children tasks must synchronize back with their parents.

The Habanero parallel programming language [41–43] also relies on fine-grain synchronization mechanisms, such as phasers, data-driven futures, and async/finish constructs. Contrary to Cilk and CilkPlus, Habanero relies on a terminally strict synchronization approach, *i.e.*, a child task must synchronize back with *any* of its ancestors (not necessarily its parent task). While most of the research pertaining to Habanero relies on the Java virtual machine, the Habanero programming model has also been ported to a C-like language, Habanero C.

Finally, the latest version of the *OpenMP* standard proposes a way to describe task dependencies in a program [3] [4]. The way task dependencies are expressed is directly inspired by StarSs and OmpSs [44, 45]. The resulting task dependence graph is obtained in a fully dynamic manner. By contrast, *DARTS*'s *Codelet*t graphs tend to dynamically allocate chunks of *Codelet*s which feature statically-defined dependencies.

3.4.2 Other Approaches to Optimize and Parallel Stencil Computations

Classical loop optimization techniques provide very efficient ways to improve sequential stencil computation. Loop tiling, locality optimization and parallelization are the main methodology to improve stencil computation performance. Loop tiling [46–48] manipulates hyperplanes from the iteration space to determine the tile shapes for a given computation, as well as the scheduling order.

From a parallel optimization angle, diamond-tiling [49] focuses on concurrent start-up as well as perfect load-balance. It enables tiles to start being processed simultaneously to improve cache reuse and provide a high degree of concurrency. However, this technique is often used manually, as it requires a complex mapping of tiles to different cores. Hence, this technique is limited in that it involves a complex control flow, an architecture-specific tile size and tile shape and an overall lower portability. Some of these limitations are addressed by Bertolacci *et al.* [50] by proposing a parameterized diamond tiling technique to better schedule tile processing.

More recently, the manipulation of the iteration space has led to better work scheduling for many-core devices. For example, Shrestha *et al.* propose to perform transformations on the iteration space using jagged-tiling to allow for a better concurrent start for processing tiles in parallel [51, 52].

Several works have proposed to automate the tuning and code generation of stencil-based applications. Among them, Pochoir is a domain-specific language that allows the user to specify a given type of stencil computation to be generated automatically for parallel execution [53]. The Pochoir compiler then translates that specification into Cilk code to be executed on a parallel multi/many core machine.

Kamil *et al.* [54, 55] propose a code generation and auto-tuning framework for stencil computations targeted at multi- and many-core processors. It makes it possible to leverages the auto-tuning methodology to optimize strategy-dependent parameters for a given hardware architecture.

Muranushi and Makino introduced the PiTCH tiling method [56]. It leverages a temporal blocking methodology which can achieve a target's optimal memory bandwidth ratio well-suited for multidimensional stencil computations.

Lesniak introduced a block-based wave-front synchronization technique for parallel stencil calculation [57]. The matrix is divided into blocks; each diagonal block can be calculated independently by different threads. New threads cannot be invoked until all blocks in the current diagonal have been calculated. The iteration is completed only if the last block, located in the lower-right corner of matrix, has been calculated. In general, the wave-front synchronization limits the level of parallelism. Hence the parallelism level is reduced from the center of the diagonal to the upper-left and lower-right corners of matrix.

Rawat *et al.* [58] have introduced their Stencil Domain Specific Language (SDSL), which provides a target-independent description and optimization strategies for stencil computation on multi-core CPUs with short-vector SIMD instructions, GPUs, and FPGAs. The purpose of SDSL is to provide a programming language which can generate high-performance portable stencil computation running on multiple platforms. It adopts both nested split-tiling and hybrid split-tiling methods in conjunction with dimension-lifting transformation.

None of the upper described works on stencils are directly competing with the objective, which is to advocate for finer-grain synchronization, even faced with "almost embarrassingly parallel" and well-balanced workloads. However, all these techniques could clearly be used on top of the event-driven multithreading run-time system, to improve the overall performance of the stencil application.

3.5 Observations

Fine-grain synchronization with our event-driven multithreading model, as described in section 3, has the advantage of exploiting the parallelism of dependence-heavy applications compared to the coarse-grain synchronization in current high-performance general purpose many-core shared-memory compute nodes. Several variants, coarse-grain variants implemented with *OpenMP*4, fine-grain variants implemented with *OpenMP*4.5, as well as several variants using fine-grain event-driven execution run-time system(*DARTS*), were developed to leverage the granularity of the synchronization.

While there are various ways to optimize stencil-based kernels, as described in section 3.4, the experiments, described in section 3, demonstrate that even only with a simple hierarchical synchronization scheme, the reduction in the number of atomic operations and amount of memory traffic in general benefits the overall execution of the program. By leveraging such a synchronization scheme, a transformation method from a coarse-grain program into a fine-grain one has been demonstrated. The cost of such a transformation is that what was initially expressed as an "almost embarrassingly parallel" loop (within a time step) now becomes a more complex computation graph. However, the advantages of using finer-grained synchronization show that even initially "almost embarrassingly parallel" workloads such as stencil kernels, performance can improve by up to 3.5× using regular work distribution among processing elements. Furthermore, a realistic stencil-based LULESH application was used to evaluate the idea that fine-grain synchronization matters even in "regular" general-purpose many-cores systems for applications which are dependence-heavy. Compared to the reference coarse-grain *OpenMP* version, the speed up of the fine-grain tree-based approach version reaches $1.35\times$.

This fine-grain synchronization work has relied on hand-coded *Codelet* programs. For the type of applications studied in section 3, this only means that the code is slightly more verbose than its *OpenMP* counterpart. However, for more complex parallel applications, it can be unwieldy to apply the same methodology. Future work includes the use of an *OpenMP-to-Codelet* compiler, `omp2cd` to observe how our fine-grain partitioning can be automated through the use of *OpenMP4.5*'s compilation directives. To do so, expanding paper [23] will be done by adding the missing directives related to `task` (*e.g.*, `taskloop`) so that fine-grain *OpenMP*4.5 code may be generated into a multi-level synchronization scheme *DARTS* program.

Chapter 4

Profile-Based Dynamic Adaptive Work-Load Scheduler on Heterogeneous Architecture

4.1 Introduction and Motivation

Nowadays, most High-Performance Computing (HPC) platforms feature heterogeneous hardware resources (CPUs, GPUs, FPGAs, storage, etc.). For instance, the number of platforms of the Top500 equipped with accelerators has significantly increased during the last years [1]. In the future it is expected that the nodes of such platforms' heterogeneity will increase even more: they will be composed of fast computing nodes, hybrid computing nodes mixing general purpose units with accelerators, I/O nodes, nodes specialized in data analytic, etc. The interconnect of a huge number of such nodes will also lead to more heterogeneity. Resorting to heterogeneous platforms can lead to better performance and power consumption through the use of more appropriate resources according to the computations to perform. However, it has a cost in terms of code development and more complex resource management.

Moreover, GPU boards are integrated with multi-core chips on a single compute node to boost the overall computational processing power. The scientific applications tend to rely on large amounts of data. Hence, heterogeneous systems pose some restrictions on how some of the computation can be offloaded to an accelerator, *e.g.*, GPUs, as their memory capacity is limited, and data transfers incur very limited latancy and bandwidth [59]. A heterogeneity-aware scheduler must leverage load-balancing techniques in order to obtain the best workload partition between CPUs and general-purpose accelerators—to be specific, a GPU. Naïve heuristics may result in worsened performance and power consumption. Furthermore, due to the complex and dynamic interplay between the program and hardware system [60], efficiently executing parallel programs on many-cores continues to be a challenging problem, where efficient execution requires dynamically and continuously matching the parallelism programs with the instantaneous resources. It is non-trivial work since neither the programs demands nor the system resources remain constant during the execution time.

Meanwhile, whole sectors of scientific computing rely on iterative algorithms. In particular, stencil-based computations are at the core of many essential scientific applications: stencils are used in image processing algorithms, *e.g.*, convolutions; partial differential equation solvers, Laplacian transforms, or computational fluid dynamics; linear algebra, to apply the Jacobi method; etc. the iterative nature of a stencil kernel is what makes it an interesting type of computation kernel. As a result, it must make sure everything is correctly synchronized between two time steps. Thus, if such a kernel is to be used in a heterogeneous context, the application will be required to perform host-accelerator synchronizations regularly, which will make dynamic workload scheduling even more complicated.

Section 4 proposes an approach to solve the workload balance problems between heterogeneous resource during run time to obtain higher performance. The following questions are attempted to solve in section 4 :

1. How can a system dynamically adapt its scheduling policy according to availability of heterogeneous resources? To answer this question, a novel approach to dynamic scheduling of data-driven tasks on heterogeneous systems is approached relying on the concept of *co-running*, as defined by Zhang et al. [61]: a system has enabled co-running, if it runs applications decomposed in tasks which can run simultaneously on both CPUs and general purpose accelerators. In a co-running mode, it is possible that two instances of the *same task* run on both CPU and GPU processing different subsets of the input data.*Co-running* friendly applications, *i.e.*, which can run on both GPU and CPU concurrently, tend to have low memory/communication bandwidth requirements, compared to applications which run the workload on only one part of the system. Hence, the computation-memory ratio and computation patterns can help identify the suitability of the workload to a resource.

2. Are using all the computing resources simultaneously the necessary to obtain the highest performance? To answer this question, a serial experiments were set up. As described in section 4.4.2.2, there is no clear-cut answer, and it all depends on a wealth of parameters, both hardware and software

3. How to build a accurate estimation model? Many researchers such as Van Craeynest *et al.* [62], Power *et al.* [63], García *et al.* [64], Zhang *et al.* [61], Chen *et al.* [65], and Yang *et al.* [66] proved that heterogeneous system architectures are impacted significantly by several parameters, such as number and type of cores, their topology (cores and memory hierarchy), bandwidth, the communication congestion and synchronization mechanism, as well as other hardware or software factors. However, the growing variety of hardware devices increase the difficulty of building a mathematics estimation model while keeping higher accuracy. Moreover, the mathematics model need to be rebuilt once the Hardware changes. The GPU concurrent stream technique furthermore increase the complexity, as described in paper [67]. A profile-based ML approach are proposed to reduce the

complexity of establishing an estimation model while promoting its accuracy, more detail will be described in section 4.3.3.

4.2 Background

4.2.1 Heterogeneous Computing and Co-running Applications

Heterogeneous computing is about efficiently using all computing resources in the system, including CPUs and GPUs. Usually, GPUs have been connected to a host machine (CPU) by a high bandwidth interconnect such as PCI Express (PCIe). Here, host and device have different memory address spaces, and data must be copied back and forth between the two memory pools explicitly by the data transfer functions. One of the most important challenges of Heterogeneous computing is how to fully utilize heterogeneous resources while minimizing the communication costs between different resources by leveraging communication-computation overlap.

Co-running friendly application [61] can run on both GPU and CPU concurrently, and tend to have low memory/communication bandwidth requirements compared to applications which run the workload on only one part of the system. Hence, the computation-memory ratio and computation patterns can help identify the suitability of the workload to a resource. For example, if the application is characterized, such that the communication time (data transfers) between CPUs and GPUs is far higher than the computation time of a given workload, and if there is no overlap between computation and communication, then this application will belong to the "co-running unfriendly class". However, the categorization may change when the application is developed in different hardware architectures or even in same hardware architecture with different dataset sizes.

4.2.1.1 Heterogeneous Hardware Communication cost

On integrated CPU/GPU architectures, Zhang *et al.* [61] suggested that the architecture differences between CPUs and GPUs, as well as limited shared memory bandwidth, are two main factors affecting co-running performance.

Lee *et al.* [68] analyzed a set of important throughput computing kernels on both CPUs and GPUs. They showed the differences of optimization features, contributing to performance improvements on these architectures. According to their conclusions, CPUs can have comparable performance to GPUs, if they fully utilize CPU optimization techniques, such as cache blocking, and reorganization of memory accesses for SIMD units, among others. On SMP (Symmetric Multi-Proces-sing) systems connected to GPU architectures, beside architectural differences, communications between CPUs and GPUs is another important factor. GPUs and CPUs are bridged by a PCIe bus. Data are copied from the CPU's host memory to PCIe memory first, and are then transferred to the GPU's global memory.The PCIe bandwidth is always a crucial performance bottleneck to be improved. Nvidia provides ways to pin memory to lower data transfer latency [69]. However, performance may be degraded if the allocated pined memory size is too big.

Congestion control mechanisms have a significant impact on communications. Moreover, the PCIe congestion behavior varies significantly depending on the conflicts created by communication. Martinasso *et al.* [70] have explored the impact of PCIe topology, a major parameter influencing the available bandwidth. They developed a congestion-aware performance model for PCIe communication. They found that bandwidth distribution behavior is sensitive to the transfer message size. PCIe congestion can be hidden if the overlapping communications transfer very small message sizes. However, when the message size reaches some limit, congestion will significantly reduce the theoretical transfer bandwidth efficiency.

4.2.1.2 Concurrent Streams on GPUs

Starting with CUDA v7 (published in 2015), CUDA's programming model was augmented with stream-based constructs, able to schedule multiple kernels concurrently, while overlapping computation and communication. This allows the system to hide host-accelerator data transfer latency. A CUDA stream can encapsulate multiple kernels, which must be scheduled in a particular order. CUDA streams are usable as long as the target GPU has more than one copy engine and one kernel engine—which is true of most recent GPUs.

A main optimization of the developed application was to overlap data transfers across the PCIe bus [70]. This is only possible using CUDA streams and pinned memory in the host. Using pinned host memory enables asynchronous memory copies, lowers latency, and increases bandwidth. This way, streams can run concurrently. However, this goal is constrained by the number of available kernels and copy engines exposed by GPUs. Also, synchronization must be explicit in the stream kernels.

There are GPUs with only a single copy engine and a single kernel engine. In this case, data transfer overlapping is not possible. Different streams may execute their commands concurrently or out of order with respect to each other. When an asynchronous CUDA stream is executed without specifying a stream, the CUDA runtime uses the default stream 0; but when a set of independent streams are executed with different ID numbers, these streams avoid serialization, achieving concurrency between kernels and data copies.

Figure 4.1 explains the streaming model which are used to improve the performance of the target GPU application. This figure compares the sequential computation of two different kernels with their respective data transfers: one single stream vs. 3 different kernels with their respective data transfers using 3 streams. The second method is only possible in GPUs with two copy engines, one for host-to-device transfers and another for device-to-host transfers.

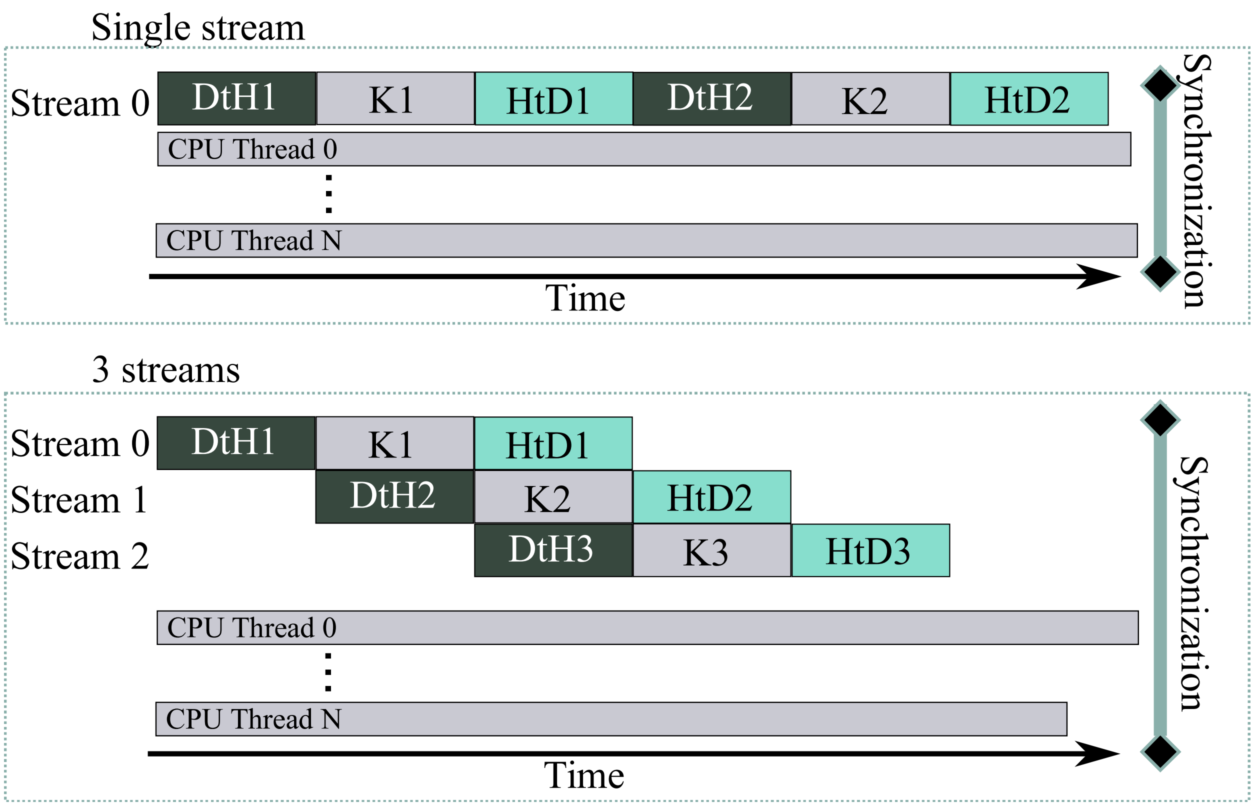

Figure 4.1: Concurrent Streams overlap data transfer

4.2.2 Heterogeneous-DARTS Run-time System

As described in Section 2, *DARTS*[22–24] run-time system is an implementation of the Codelet Model and the *Codelet Abstract Machine*(CAM) on which it relies [19]. A CAM is an extensible, scalable and hierarchical parallel machine model. It is a many-core architecture with two types of units: scheduling units (SUs), which perform resource management and scheduling, and computation units (CUs), which carry out the computation. CUs and SUs are grouped into several clusters and they benefit from data locality. *DARTS* maps these "abstract cores" to physical processing elements (PEs).

To target CPUs-GPUs heterogeneous system, a new scheduler named `CPU-GPU-Corunning` are designed to allocate/schedule computing on both CPUs and GPUs. A new type of `Codelet` named `GPU_Codelet` is created to control/configure/run computing on GPU. Then,there are two main `Codelet` types: `CPU_Codelets` and `GPU_Codelets`. They can run concurrently on different cores.

4.2.2.1 `GPU_Codelet`

`GPU_Codelet` consists of two parts: CUDA host code and CUDA Kernel code. The host and kernel codes can run concurrently or sequentially, see section 4.2.1.2.

Similar with `CPU_Codelet`, `GPU_Codelet` have zero or multiple dependencies. `GPU_Codelet` will be first pushed to ready pool of SU when all its dependencies are satisfied. Then, instead of being pushed to normal CU by SU scheduler, `GPU_Codelet` will be pushed to Specific CU which is equipped with GPU scheduling policy.

`GPU_Codelet` include sub co-running policy to decide the synchronization/asynchronization between host and GPU kernels, whether to use concurrent streams, how many streams, and access which GPU(s) *etc.*.

4.2.2.2 `CPU-GPU-Corunning` scheduler

Theoretically, every CPU core (CU) can be the host of GPU or GPUs, which means every CU scheduler equips with two types of ready queue, one for `CPU_Codelet`s and one for `GPU_Codelet`s. However, to reduce the complexity of scheduler, the number of CUs which can interact with GPU, called binding CUs, depends on the number of GPUs in the system. Every binding CU can access any available GPU(s). In contrast, no-binding CUs can't directly interact with GPU(s). Binding CU also can run normal `CPU_Codelet` when there is no `GPU_Codelet` available or when `GPU_Codelet`'s kernel code asynchronously running on GPU(s) and its host code is finished.

`CPU-GPU-Corunning` scheduler have two functions: one is allocating `CPU_Codelet`s and `GPU_Codelet`s to matched CU ready pool; another is checking availability of computing resources. The scheduler will invoke a set of back up `CPU_Codelet`s which have the same function of the `GPU_Codelet` if GPU resources are not available and estimated waiting time is longer than running backup `CPU_Codelet` on CPUs including waiting time on CPUs. The `GPU_Codelet` will stay in the ready pool if the waiting time is shorter than the running backup `CPU_Codelet`s.

A monitor module is used to record running time of all `Codelet` in current CU. It will provide a reference to the scheduler when it need to estimate running time and waiting time for future `Codelet`s with same function but same or different workload.

4.3 Methodology: DAWL and IDAWL

This section consists two parts: propose a dynamic adaptive workload algorithm (DAWL) as a first step, and a profile-based machine-learning estimation model as an optimization over it (IDAWL).

4.3.1 Target: Dependence-heavy Iterative Applications

Implementing a parallel algorithm for heterogeneous computing can yield outstanding results, but there is still a lack of tools to get better performance [69]. Further, even if it is written with heterogeneity in mind, a parallel application may not exploit the parallelism of various computing resources. In particular, data-parallel algorithms dealing with large blocks of data (*i.e.*, algorithms featuring intense arithmetic with regular (array-based) data structures), can greatly benefit when they are implemented to run exclusively on GPUs [71, 72].

Applications targeting heterogeneous systems must be implemented with load-balancing in mind to better exploit the various compute units in a system. As a result, the workload behavior for both the host and devices must be carefully analyzed. This work focuses on parallel applications which require regular and/or periodic synchronization steps between the host and the devices on a heterogeneous platform. Stencil-based computations have an iterative nature, and expose heavy data-dependence patterns; they are well suited to evaluate the proposed scheduling strategies. Furthermore, such stencils make up the core computation kernels of some benchmarks such as Rodinia [73].

A 5-point 2D and a 7-point 3D stencil kernels will be used as case studies to explain the Dynamic Adaptive Work-Load (DAWL) scheduling algorithm below, which is outlined in algorithms 1, 2 and 3.

4.3.2 Dynamic Adaptive Work-Load Scheduler

A balance must be found between heterogeneous devices' computing potential and memory bandwidth/capacity.Equation 4.1 shows that the GPU execution time is split into two parts: round-trip data transfers ("Xfer") between the host and the device, and computing time.

$$GPU_{naive} = \mathrm{Xfer}_{H\rightarrow D} + \frac{\mathrm{Compute}_D}{\mathrm{NumThreads}_D} + \mathrm{Xfer}_{D\rightarrow H} \tag{4.1}$$

Using concurrent streams (see Section 4.2.1.2), data transfers can be partly or totally overlapped with computing, creating a pipeline of sorts. There are many parameters required to build an overlapping model, including the PCIe bandwidth the number of concurrent stream and copy engines *et al.* [67].

The model expressed in Equation 4.1 is too simple to integrate all features of multiple concurrent streams-based computations. Hence, Machine learning (ML) techniques are leveraged to build a model to support predictions in a CPU-GPU ratio (see Section 4.3.3.2). Equation 4.2

holds when programs run on multiple cores.

$$CPU_{naive} = \frac{\text{Compute}_H}{\text{NumThreads}_H} \tag{4.2}$$

Equation 4.3 estimates the ratio r of execution time when *applications* run on CPUs or GPUs.

$$r = \frac{\text{CPU\_naive}}{\text{GPU\_naive}} \tag{4.3}$$

Algorithm 1 consists of two steps: (1) Choose hardware (CPUs, GPUs, or both) ; and (2) run tasks on the selected hardware. Equation 4.3 was used in step 1. $r \gg 1$ means tasks running on CPUs are far slower than on GPUs. Hence, all computation will be carried on the device (GPUs); On the contrary, $r \ll 1$ indicates CPUs should be chosen to run tasks; when r is closer to 1, the tasks will be distributed in a co-running manner. In 2D and 3D stencil, with different time step settings, r's range may change significantly. In particular, data transfers between host and devices affect the performance when GPUs are used.

Algorithm 1: Dynamic adaptive workload balance between heterogeneous Resources

```
1 Function main(HW_Info, WL(problem_size), GPU_WL, CPU_WL, Limit_WL,
    GPU_Change_ratio, CPU_Change_ratio):
2     step1: decision = hardware_choose(HW_Info,WL)
3     step2: if decision = Co_running then
4         Co_running_WL_balance(HW_Info,total_WL, GPU_WL, CPU_WL, Limit_WL,
            GPU_Change_ratio, CPU_Change_ratio)
5     else if decision = CPU then
6         run_CPUs(HW_Info,WL)
7     else
8         run_GPU(HW_Info,WL)
9     end
```

How the workload must be partitioned between host and devices to enable co-running depends on two conditions: First, the static initial workload on GPU,which should be smaller than the available GPU memory considering the communication cost; second, the

Algorithm 2: DAWL: Hardware Choose and Run Function

```
Function hardware_choose(HW_Info, WL):
    if WL ¡ GPU_memory_available_size then
        if $r \gg 1$ then
            decision = GPU
        else
            decision = CPU
        end
    else
        decision = Co_running
    end
Function Run_Func(Hardware_Info, type, WL, Remaining_WL, Limit_WL,
  Change_ratio):
    if type = CPU then
        CPU_Func(HW_Info,WL)
    else
        GPU_Func(HW_Info,WL)
    end
    if Remaining_WL¡Limit_WL then
        WL = Remaining_WL;
    else
        (faster,Ratio) = check(CPU_status,GPU_status)
        if faster = CPU then
            TWL = WL * (1-change_ratio)
        else
            TWL = WL * (1+change_ratio)
        end
        if Remaining_WL¡TWL then
            WL = Remaining_WL * Ratio
        else
            WL = TWL
        end
    end
    Remaining_WL -= WL
    sync_GPU_CPU(Remaining_WL,MEM)
    renew(WL_min,WL_max)
    Run_Func(HW_Info, type, WL, Remaining_WL, Limit_WL)
```

initial workload can be obtained with the Profile-based Estimation Model, described in Section 4.3.3. It is based on profile information obtained from `OProfile` and `nvprof` and the compute node's hardware configuration, such as the number of CPU cores, LLC, the GPU

Algorithm 3: DAWL: Asynchronous Parallel function and load balance

```
1  Function SYNC_Rebalance_Func(HW_Info, CPU_WL_Info,GPU_WL_Info):
2      CPU_WL = Rebalance(CPU_WL_Info)
3      GPU_WL = Rebalance(GPU_WL_Info)
4      IsChange = check_Hardware(HW_Info)
5      if IsChange=true then
6          CPU_WL = CPU_Update(CPU_WL,HW_Info)
7          GPU_WL = GPU_Update(GPU_WL,HW_Info)
8      end
9  Function Co_running_WL_balance(HW_Info, total_WL, GPU_WL, CPU_WL,
     Limit_WL, GPU_Change_ratio, CPU_Change_ratio):
10     GPU_initialize(HW_Info,GPU_WL)
11     CPU_initialize(HW_Info,CPU_WL)
12     Remaining_WL = total_WL - GPU_WL-CPU_WL
13     do
14         PARALLEL EXECUTION: GPU and CPU
15         GPU: Run_Func(HW_Info, GPU, GPU_WL, Remaining_WL,
16                       Limit_WL, GPU_Change_ratio)
17         CPU: Run_Func(HW_Info, CPU, CPU_WL, Remaining_WL,
18                       Limit_WL, CPU_Change_ratio)
19         SYNC_Rebalance_Func(HW_Info, CPU_WL_Info, GPU_WL_Info)
20     while Iteration !=0
```

type *etc.*. The DAWL scheduling algorithm aims at minimizing workload imbalance between heterogeneous processing elements. DAWL dynamically adjusts the workload distribution on different computing resources based on real time information. DAWL is composed of six steps (steps 3 and 4 are detailed in Figure 4.2):

1. Initialize CPU, GPU configurations: determine the number of processing elements (PEs), their initial workload, *etc.*

2. Run the tasks with initial workload on CPUs and GPU simultaneously.

3. Monitor the computation on CPUs and GPU, record their respective execution times with their specific workload and adjust workload on PEs based on the all stored record information. PEs are given the same amount of work at first (see Figure 4.2). Once the GPU task is finished, it adds its execution time and current workload to the record.

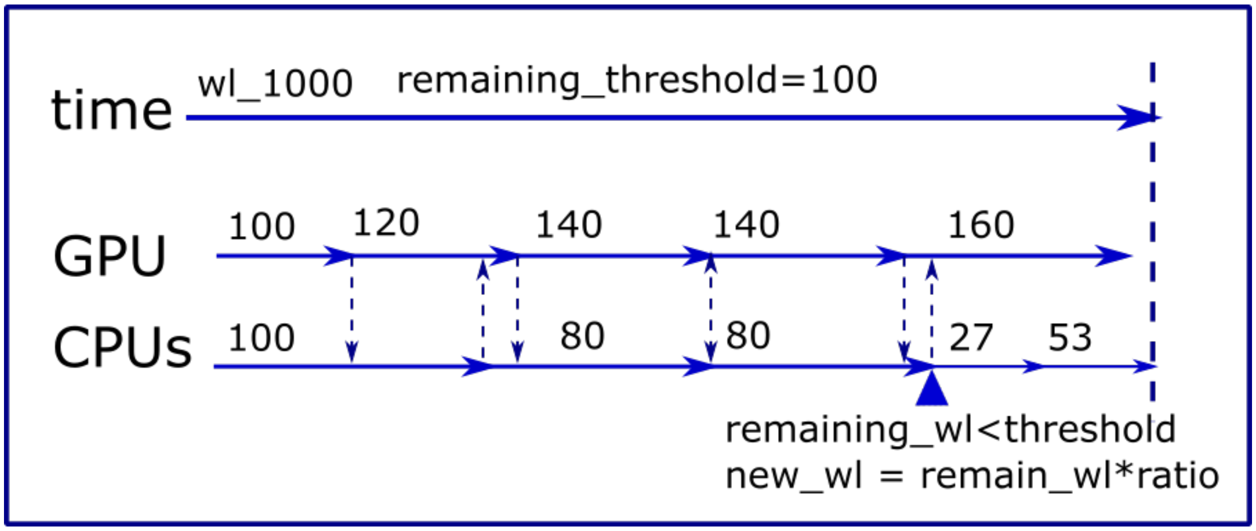

Figure 4.2: Example: CPU-GPU Workload Balancing with DAWL.

Because there is no CPUs record existing, then its computing potential is currently greater than the CPUs', and its next workload will be increased with a ratio of 10 to 20%. When CPUs' task finish, excepting adding its record (current workload and execution time), scheduler will also adjust next workload based on all the history record of both GPU and CPUs.

4. Adapt to borderline cases. $ratio = CPU_{cur}/(GPU_{cur} + GPU_{cur})$, where CPU_{cur} and GPU_{cur} represent the amount of work finished on CPUs (resp. GPUs). When the remaining workload is within 10% of the total workload (see Figure 4.2), the CPUs or the GPU only takes $\lfloor ratio \times \text{remaining workload} \rfloor$ (*e.g.*, 27 in the Figure 4.2 case) as a next task, no matter which one finishes first. The second part is allocated to whichever set of PEs finishes early. Thus, the first and second parts may run either type of PE.

5. Synchronize and re-balance the workload (see Algorithm 3) when all the compute tasks in one time step finish. The load-balancing function checks the workload information, and computes the mean workload for each PE type. If the hardware changed, *e.g.*, if several CPUs are unavailable, the system must adjust the workload on both CPUs and GPU.

6. Reset CPUs and GPU and free allocated memory.

4.3.3 Profile-based Estimation Model

4.3.3.1 Heterogeneous Systems and the Importance of their Initial Workload

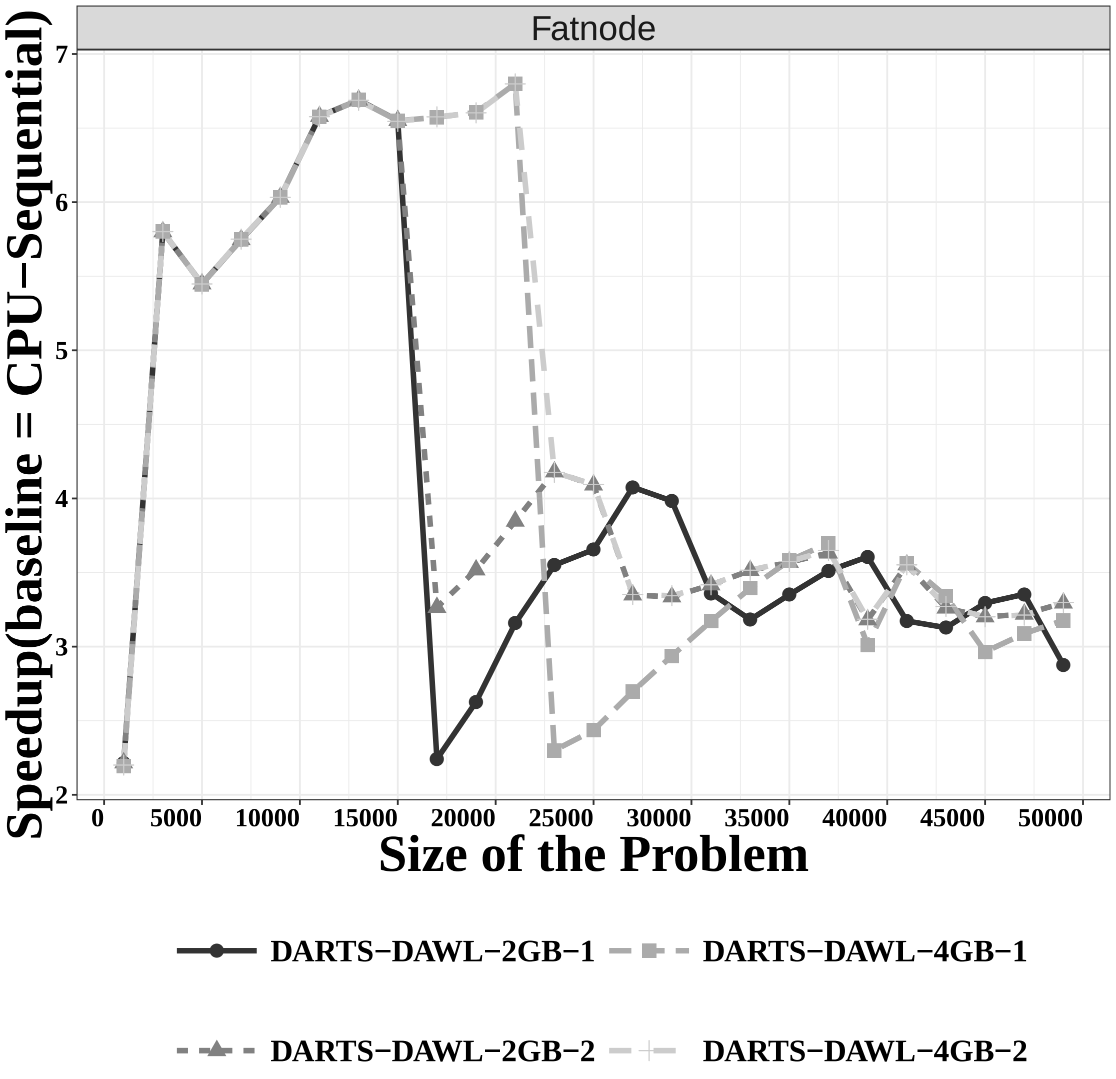

Figure 4.3: 2D stencil: speed up when GPU memory is 2 and 4GB with different initial workload (GPU=CPU): $0.5 \times av\_GPU$ (1) vs $2000 \times *$ (2)

While DAWL can dynamically adjust the workload according to real-time information, an unsuitable initial workload may drag down the performance when the problem size is relatively small since there are no enough time to adjust workload. As shown in Figure 4.3, when problem sizes are close to a specific drop point, an unsuitable initial workload (such as an initial workload on GPU close to the problem size) lowers the performance. It is not a guaranteed behavior; Figure 4.3 shows the performance of a small initial GPU workload (2 GiB, DARTS-DAWL-2GB-2 and 4 GiB, DARTS-DAWL-4GB-2) can be fare better than a big initial workload (2 GiB, DARTS-DAWL-2GB-1,4 GiB, DARTS-DAWL-4GB-1,). in Figure 4.3, (1) stands for the initial workload equals to 0.5$\times$ total workload,(2) stands for the initial workload equals to 2000 rows $\times$ columns of total workload, *e.g.* total workload equals to 4000 (rows)$\times$4000(columns), then (2) will equals to 2000$\times$4000. However, Assuming a small workload will yield better results by default is not always working, especially for stencil-based applications. Indeed, they feature heavy data dependencies, and thus must be synchronized when partitioning the workload into different tasks. A suitable initial workload can help fully utilize the computing resources with reasonable amounts of data transfers.

4.3.3.2 Profile-based Estimation Model for an Iterative DAWL (IDAWL)

In heterogeneous many/multi-core system, hardware configuration is one of the most important information to gather to estimate the performance of applications. However, the growing variety of hardware devices as well as their combinations increase the difficulty of building estimation model and reduce the accuracy of the established model. Furthermore, a tiny change in the hardware configuration may generate fantastic variations on performance. For heterogeneous systems, building an accurate transfer-computing mathematical model including concurrent streaming aspect is a huge challenge [67].

To solve the problem of building an accurate transfer-compute mathematical model on heterogeneous systems, a profile-based ML approach is proposed to reduce the complexity of establishing an estimation model while promoting its accuracy. Such a model works for the same type of application on same configured system. It will provide a reference for other types of application running on the same or different configured system.

2D and 3D stencil kernels are targeted as an example to explain the principle of the ML approach (See Section 4.3.1). The ML approach will help optimize the DAWL algorithm supporting execution times prediction. The resulting algorithm is called Iterative DAWL, or IDAWL for short. A black-box ML method, (*i.e.*, an automatic model without any user intervention, is used). It follows three phases:

1. Collect hardware architecture information as parameters to the estimation model. Table 4.1 and 4.2 list some parameters of the profile-based model. Besides these, the parameters also include the host's cache hierarchy (*e.g* L1, L2 and L3 cache parameters) information, the system's PCIe concurrent data transfer rate, and GPU's parameters, including the maximum number of concurrent streaming, GPU thread dimension information, the shared memory size *etc.*

2. Collect the application's profile information at run time as training data. Different combinations of CPU cores and different GPUs are run:

   - CPU: Since too many events can be obtained from `OProfile`, option 1 will be that collecting the events related to cache misses in the cache hierarchy; branch related events will be as option 2 and will run only when necessary. Option 3 is all the events left in `OProfile` (rarely chosen).
   - GPU: option 1 collects `gpu-trace` and `api-trace` information. Option 2 collects all the metrics in `nvprof` (rarely chosen for big workloads, as it is too time consuming).
   - Sampling (leave-one-out cross validation): three levels of workloads (small, middle and huge) are running on each PE type (purely) in the system. Here, small and middle tasks are used for training and validation, and huge workload is used for testing only. The transfer-computing model sample information is obtained by splitting the workload with different ratios on the host and devices.

3. Utilize the information gathered from steps 1 and 2 to build a profile-based estimation model for a given heterogeneous platform, and obtain a customized initial workload on

different heterogeneous architectures and number of devices necessary for such initial workload. The approach attempts to predict the overlapping and running model of large data set by using the small and middle data sets.

- Run a set of ML algorithms: regression (including linear and logistic, each using multiple meta-functions, such as polynomial, logarithmic, exponential functions), ensemble learning (*i.e.*, random forests), and Support Vector Machine (SVM).
- Choose the resulting model that fits best the measured data. If several models are a good fit, pick the model that is the least computationally intensive. To evaluate how well the model fits the data, the coefficient of determination, $R_{squared}$, is used. It is defined as the percentage of the response variation that is explained by a linear model: $R_{squared} = \frac{\text{Explained variation}}{\text{Total variation}}$, with $0\% \leq R_{squared} \leq 100\%$. 0% indicates the model explains none of the variability of the response data around its mean and 100% indicates the model explains all the variability of the response data around its mean.
- Build the ML estimation model: an estimation formula of the best matched statistical model can be built to predict an application's performance on this specific heterogeneous platform. For a given problem size, a minimization multi-variable function can be used to obtain an estimation of the initial workload. The specific parameters used to construct the formula are mentioned in Section 4.4.3

IDAWL adaptively adjusts the workload between CPUs and GPU depending on the real time execution situation, and can further compensate the insufficient off-line ML method.

Table 4.1: Hardware Platforms

Param.	**CPU Parameters**					**GPU Parameters**				PCIe
Machines	**Cores**	**Clock**	**# Socket**	**L3 Size**	**CPU Mem**	**# SM**	**Clock**	**L2 Size**	**GPU Mem**	
1. Fatnode (K20)	32	2.6 GHz	2	20 MB	64 GB	13	0.71 GHz	1.25 MB	4.8 GB	6.1 GB/s
2. Super Micro (K20)	40	3 GHz	2	25 MB	256 GB	13	0.71 GHz	1.25 MB	4.8 GB	6.1 GB/s
3. CCSL (Valinhos) (k40)	8	3.4 GHz	1	8 MB	16 GB	15	0.75 GHz	1.5 MB	12 GB	10.3 GB/s
4. Debian (Titan)	12	3.4 GHz	1	12 MB	31 GB	14	0.88 GHz	1.5 MB	6 GB	11.5 GB/s

Table 4.2: Software Environment.

Platform	**GCC**	**CUDA**
Fatnode (K20)	v6.2 / v8.1	v8.0
Super Micro (K20)	v4.8.5/v6.2	v8.0
CCSL (Valinhos) (K40)	v5.4	v9.0
Debian (Titan)	v4.9.2	v9.1

4.4 Experiment

4.4.1 Experimental Testbed

The experiments run on four heterogeneous systems, presented in Table 4.1 and Table 4.2. They all feature Intel processors and Nvidia GPUs. The number and names presented in the first column of Table 4.1 are used to describe the machines.

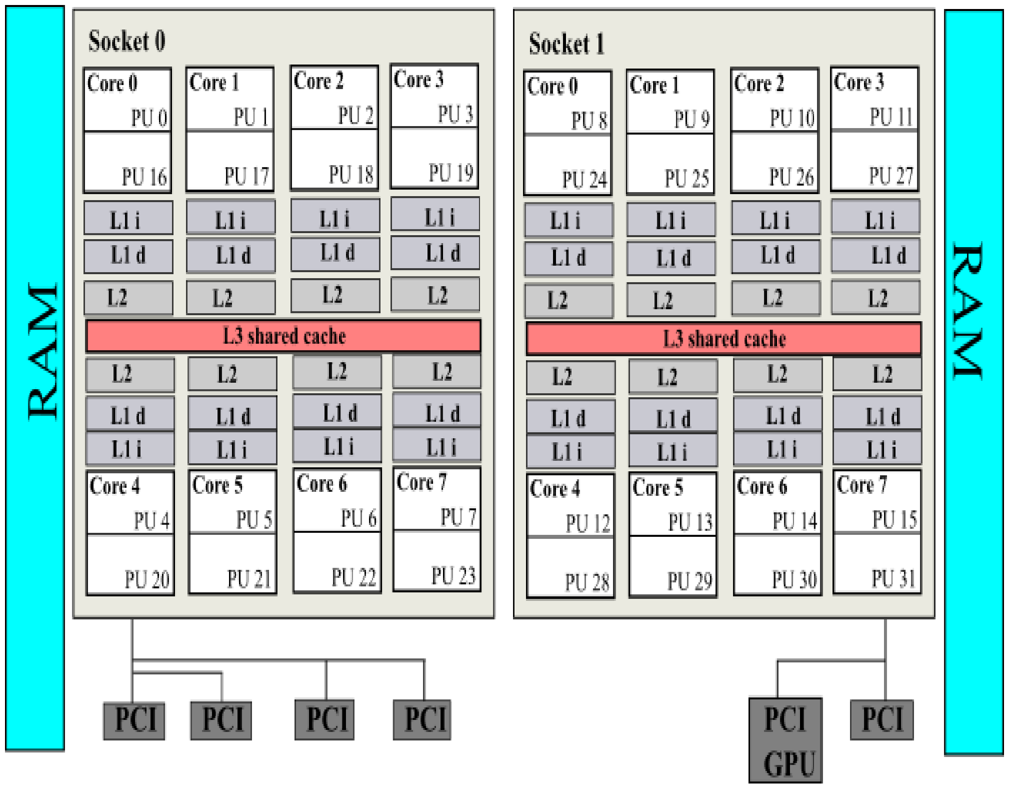

Figure 4.4: fatnode topology

`fatnode`'s general purpose CPUs are made of Intel Xeon® E5-2670, each CPU processor consists of 8 physical cores and 16 logical cores, considering hyper-threading, see Figure 4.4. processors are connected by socket and each socket has 32 GB of RAM for a total of 64 GB in this system, and one Nvidia Tesla K20® (Kepler architecture with Compute Capability 3.5) board with 5 GB of global memory. `Super Micro` equipped with two Intel Xeon®E7 v2,each processor has a total of 10 physical cores with hyper-threading of 3 GHz. This system has 256 GB of RAM, 128 over each socket. and embeds 4 Tesla K20 boards®with 5 GB of global memory each one. `CCSL Valinhos` equipped with one Intel i7-4770® processor consisting of 4 physical cores with hyper-threading of 3.4 GHz. This system has 16 GB of RAM. and one Tesla K40®. with 12 GB of memory. `Debian` equipped with one i7-4930K®processor consisting of 6 physical cores with hyper-threading of 3.4 GHz. This system has 20 GB of RAM, one Nvidia Titan®board with 6 GB of global memory and one GeForce GT 630®with 2GB of global memory.

The original DARTS main focus on homogeneous many-core systems, as explained in Section 4.2.2. an extend DARTS is used to support heterogeneous architectures to validate the workload algorithms: DAWL and IDAWL. The new scheduler is capable of controlling and monitoring `CPU_codelet`s and `GPU_codelet`s, so that the two types of tasks can synchronize with each other[1]

4.4.1.1 Target Applications

Stencil-based computations are chosen,see session 4.3.1, to evaluate the two scheduling algorithms: DWAL, and IDAWL.

To emphasize a worst-case scenario, stencil kernels without ghost cells are used, which enhances the need for synchronization.Specifically, two kernel variants: a 5-point 2D stencil, and a 7-point 3D stencil computing over double precision numbers are focused on. The

[1]The heterogeneous-DARTS source code is available at `https://github.com/gengtsh/darts/darts-heterogeneous`.

number of time steps is fixed to 30, removing the convergence test at the end of each time step to simplify the problem and make it more deterministic. Each experiment was repeated 20 times. To obtain objective test results, the mean of test results after removing the best and worst results are utilized. To make it worthwhile to run some of the workload on the GPU, CUDA's concurrent streaming described in Section 4.2.1.2 is used.

In particular, Exploiting the GPU's available scratchpad memory is very important to enhance data locality and speedup the GPU computations on each streaming multiprocessor (SM). Further, in 3D stencil, the geometry of the input arrays matters when it comes to workload partitioning, especially when considering the GPU's scratchpads. Hence, the dimensions of arrays should eked as much performance as possible from a single GPU SM. A micro-benchmarks are run to evaluate the performance of 2D stencil naïve kernels vs. L1-tiled kernels on Intel CPUs. L1-tiled kernels are used when running from DARTS, as they improved the overall performance by a wide margin (from $\approx 1.09\times$ to $\approx 2\times$, depending on the workload size).For 3D stencil kernels, a tile has the following dimensions: $16 \times 16 \times 10$, which represents roughly 20KiB and fits into L1 caches on Intel CPUs. When running tiled 3D stencil kernel however, the performance worsens compared to un-tiled naïve kernels. So, the un-tiled sequential 3D Stencil is chosen for the baseline of the experiments.

Beside tiling, the initial workload is statically partitioned along the first dimension, on both 2D and 3D stencils, to validate and verify DAWL. In the 2D case, the "slices" of 2000 rows (with a varying number of columns) is as a static block of elements to process, which denote as $slice \times *$. For instance, for the 2D stencils, a static partition could be a composition of 2000×2000 2D arrays. In the case of 3D stencils, as the overall memory footprint increases much faster as we increase the size of any of the three dimensions, we use $200 \times * \times *$ slices, (*e.g.*, $200 \times 100 \times 100$).

4.4.1.2 Parameter Space of Experiments

`numactl` are used to allocate memory in a round-robin fashion and avoid NUMA-related issues. All systems were configured so that only 2 GB were seen as available by the runtime system, in order to reduce the "parameters surface" to explore. Indeed, as shown in Figure 4.3, the "drop" which occurs in the charts once data does not fit in the GPU memory happens whether considering an artificial 2 GiB DRAM limit or if using the full DRAM capacity; the performance drop is "only" delayed in the latter case. Hence, the artificial constraint putting on the GPU DRAM capacity does not impact the overall methodology nor its results.

4.4.2 Performance Analysis

To comprehensively characterize DAWL, a series of workload performance analyses were performed. Five variants,the `DARTS-DAWL` performance with `GPU-Only`, `CPU-Seq`, `DARTS-CPU`, `DARTS-GPU` (see Table 4.3 for details) are compared in the experiment.

There are three different way to implement GPU version code: one is using concurrent streams for all size of workload,which is proved very inefficient when workload is less than available GPU available global memory since the cost of synchronization; second is transfer all the data to the GPU, which is also inefficient when the workload is larger than the available global memory since the huge data transfer cost; third way is only using concurrent streams method when the workload is larger than the available GPU global memory.`DARTS-GPU` is using first method for 2D Stencil and using the third method for 3D Stencil, more detail are described on session 4.4.2.1 . To keep the same amount of memory allocation of every stream, the number of stream changes with workload. For example, the number of stream equals to 4 when the workload is smaller than the 0.5 $\times$ GPU global memory, and the number of stream changes to 8 when the workload is closed to available GPU global memory. `GPU-Only` is using the third way.Both `DARTS-GPU` and `GPU-Only` code consist two parts: host and kernel. Here host code run on one CPU, kernel code run on GPU. Beside transferring data between CPU main memory and GPU global memory, host code also need to synchronize CPU and GPU and configure GPU's parameters, such as the number of concurrent streams, the grid structure of GPU and so on. `CPU-Seq` is the implementation of Sequential code on one CPU and is used as base line.`DARTS-CPU` is the implementation of multi-CPU-core code on DARTS. `DARTS-DAWL` is the implementation of DAWL on DARTS. Depending on parameters mentioned in session 4.3.2, `DARTS-DAWL` may run on multiple CPUs or GPU, or be co-running on both CPUs and GPU.

Table 4.3: Stencil kernel implementation

Implementation	Illustration
`CPU-Seq`	Sequential c++ code
`GPU-Only`	CUDA code
`DARTS-CPU`	Multi-threads c++ code
`DARTS-GPU`	CUDA code on DARTS (concurrent streams)
`DARTS-DAWL`	DAWL hybrid code on DARTS

4.4.2.1 Full Resource Usage

Figure 4.5 shows the speedup of different 2D Stencils, and Figure 4.7 does the same for 3D Stencils, using `CPU-Seq` version as a baseline. Here, all CPU related versions, `DARTS-DAWL` (may also use GPU depending the ratio r mentioned in section 4.3.2) and `DARTS-CPU`, are using all the CPU cores as computing resources. Even though there are a lot of differences, overall `GPU-Only`'s performance drops dramatically at $drop_{2D} = 17000 \times 17000$ for 2D stencil, and the same happens in the 3D Stencil, while $drop_{3D} = 400 \times 800 \times 800$. The available GPU global memory of the four different machines were setting to 2GB, that's the reason why the four machines have the same drop point. Before drop point, all the data of `GPU-Only` are copied to GPU global memory. In this case, there are only two data transfer and two

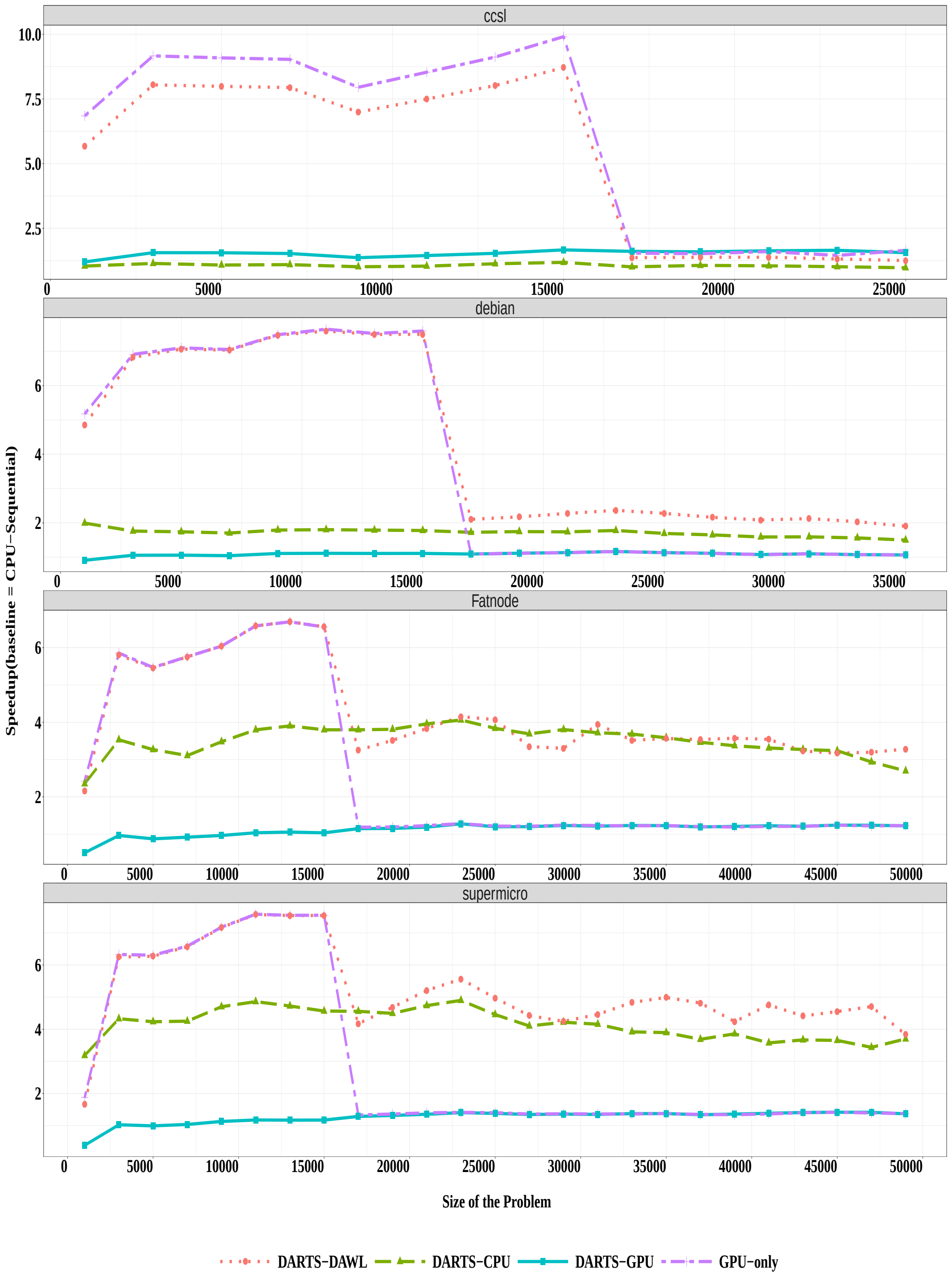

Figure 4.5: 2D stencil: Speedup of the different versions

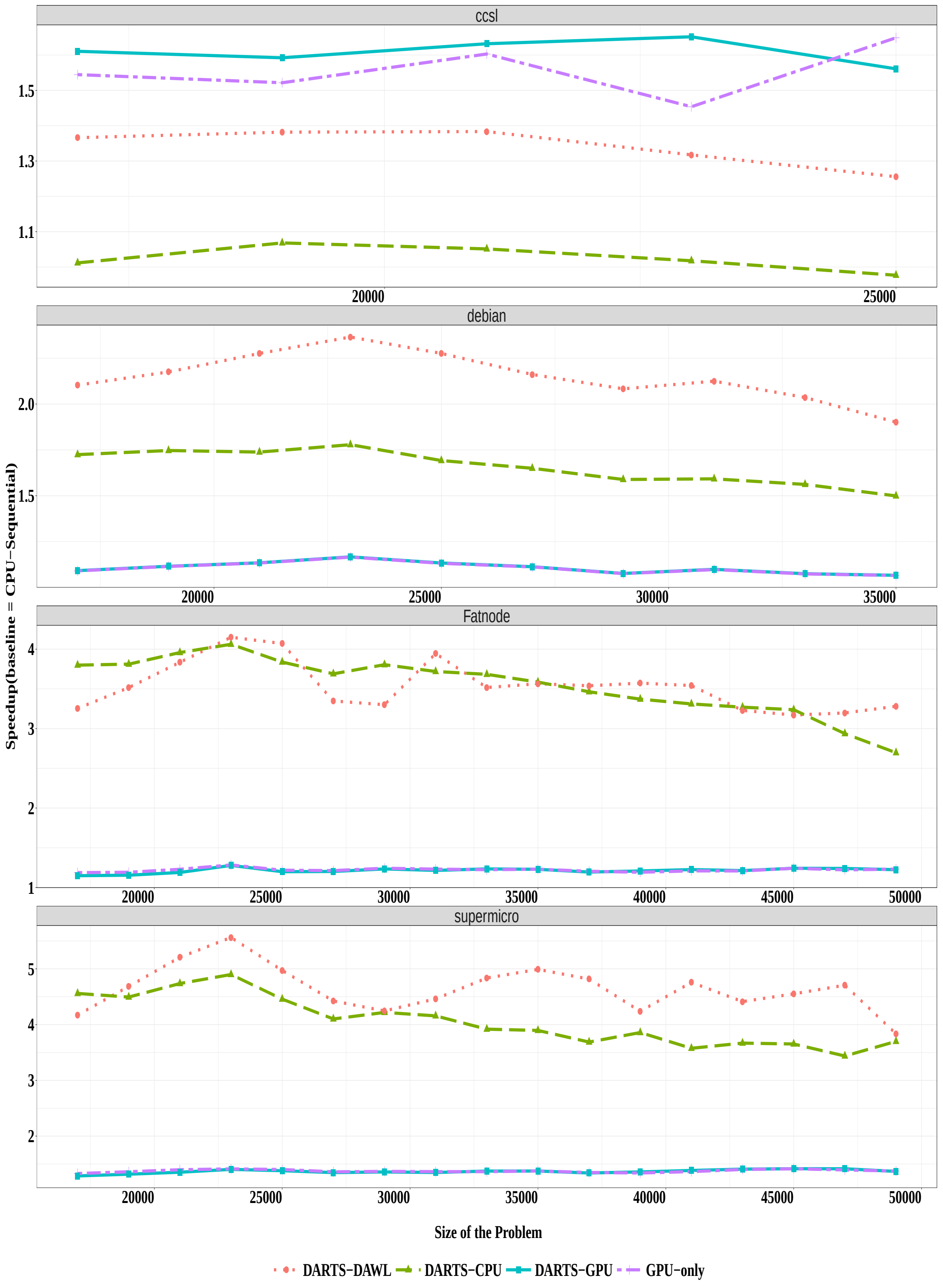

Figure 4.6: 2D stencil: Speedup when matrices are larger than 17K

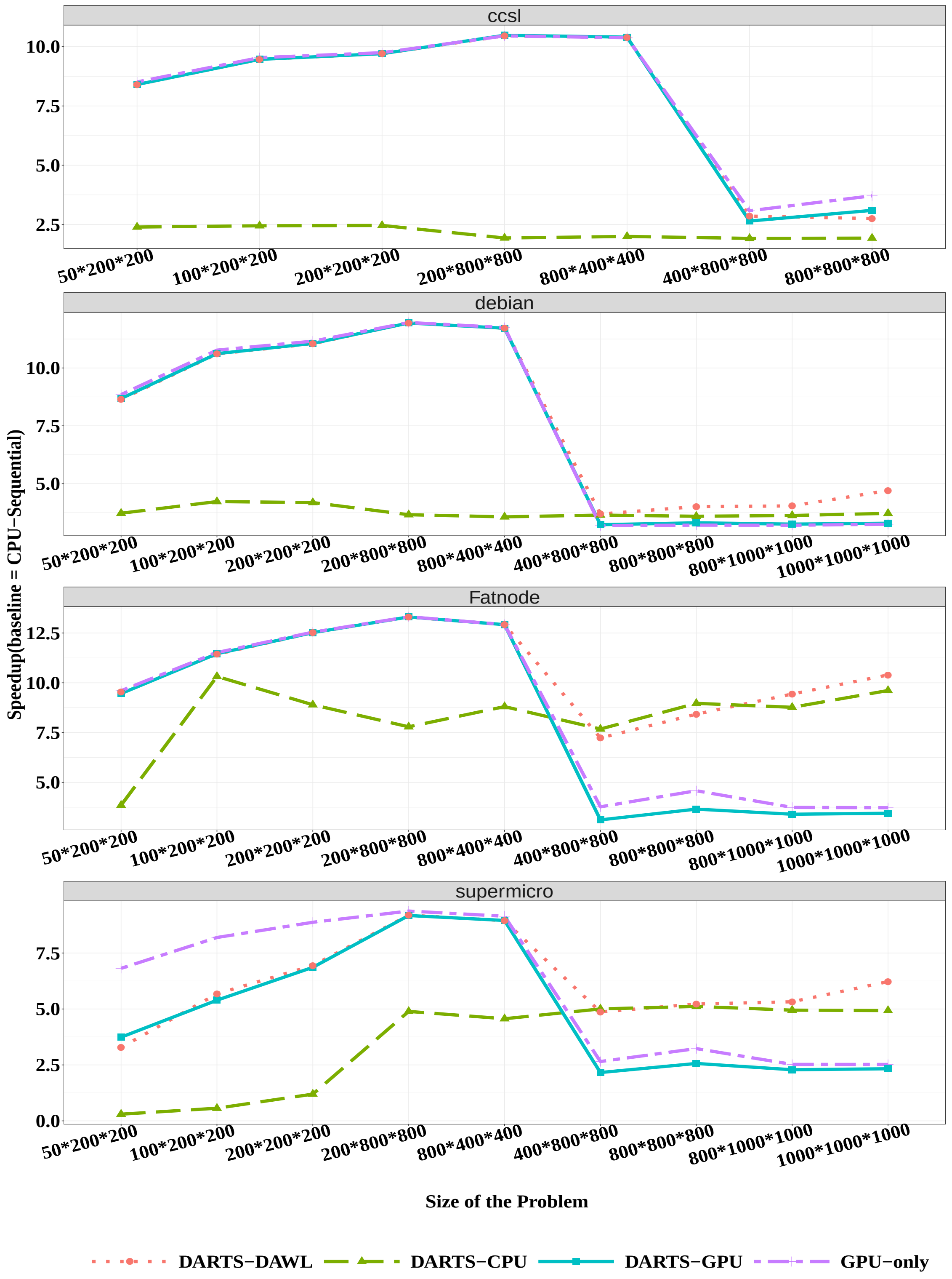

Figure 4.7: 3D stencil: Speedup with different versions

synchronization operations between CPU and GPU, one is at the beginning, another is at the end. When the workloads are larger than the GPU available global memory which means after drop point, concurrent streams approach is utilized by `GPU-Only`. Because of communication and synchronization cost, the performance of `GPU-Only` drop dramatically. For 2D Stencil, as shown in Figure 4.5, `DARTS-GPU` is using concurrent streams all the time and its performance on all the machines are pretty stable. The experiment that the performance of `DARTS-GPU` after drop points part are very closed to `GPU-Only` has proved that the new `Heterogeneous-DARTS` run-time system overlap can be overlooked. Since the theory that concurrent stream method is slower than the simple one stream method when workload is less then the available GPU Global memory is already verified or proved by 2D Stencil, `DARTS-GPU` for 3D stencil will use single stream for smaller workload and concurrent streams method for larger workload. For 2D Stencil, `DARTS-CPU`'s performance are stable. Except on `CCSL Valinhos`, `DARTS-CPU`'s performance are lower than `GPU-Only`'s before drop point and higher than `GPU-Only` after drop point and `DARTS-GPU`. Because of 3D Stencil geometry structure affect performance a lot, that's why comparing to 2D, 3D `DARTS-CPU` performance fluctuate a little.

Based on the *hardware_choose* function (Algorithm 1), when the problem size is smaller than $drop_{2D}$ or $drop_{3D}$, the application belongs to the GPU-friendly category and all the workload will be run on GPU. When the problem size is larger than $drop_{2D}$ or $drop_{3D}$, the application changes to the co-running friendly category, and computation will run on both CPUs and GPU. Figures 4.5 and 4.7 validate the mathematical model. On all systems, `DARTS-DAWL`'s performance is very close to `GPU-Only` when the input set fits in the GPU global memory capacity. As shown in Figure 4.3, the $drop_{2D}$ shifts from 17000×17000 to 23000×23000 when the GPU's memory capacity changes from 2GB to 4GB.

Figure 4.6 zooms in Figure 4.5 for matrix sizes 17000×17000 and onward. As shown in Figure 4.6 the speedup ratios are quite different on different systems with different workload.

On `fatnode` and `Super Micro`, `DARTS-DAWL` and `DARTS-CPU` are alternately faster; on `CCSL Valinhos`, `DARTS-DAWL` and `GPU-Only` are similar; on `debian`, `DARTS-DAWL` is faster than `DARTS-CPU`. The changes are affected by the differences of Hardware architecture, see Table 4.1,`CCSL Valinhos`'s GPU is a Tesla-K40, which has a slightly higher clock and memory frequency than Tesla-K20, as well as improved floating-point processing capability. Further more, comparing to other machines, `CCSL Valinhos` only equip 8 CPU cores with only 8 MB L3 cache. That's why `DARTS-CPU` on `CCSL Valinhos` is far slower than GPU related version, such as `DARTS-GPU` and `GPU-Only`. `DARTS-DAWL`'s performance is in between since the GPU have to wait for CPUs to synchronize in some extent.

4.4.2.2 Varying the CPU resources

Are using all the computing resources simultaneously the necessary to obtain the highest performance? Figure 4.8, 4.9, 4.10 and 4.11 show answer for this question. `fatnode` server stands for a classical type of hardware configuration: a "regular" GPU and a two-way SMP chip multiprocessing system,see Figure 4.4. Two different mapping policies are used to pin compute units to physical processing elements: `spread` and `compact`. The `spread` policy attempts to map compute units to a processing element (PE, *i.e.*, a core or a thread) as far as possible from each other according to the underlying physical topology. This policy tends to yield good results when the application features a large memory-to-computation ratio, there is little temporal locality to be expected, and there are possibly fewer compute units than there are actual physical PEs, as the cache is then "owned" by a single PE. On the contrary, `compact` attempts to allocate software threads as closely as possible on the available processing elements. `compact` is useful when data (and caches) are shared between threads; it ensures that locality is maximal. However, should the application's potential for cache locality be low, the sharing threads may end up trashing each others cache lines.

Even though the *compact* and *spread* methods affect plenty the performance, the rough trend

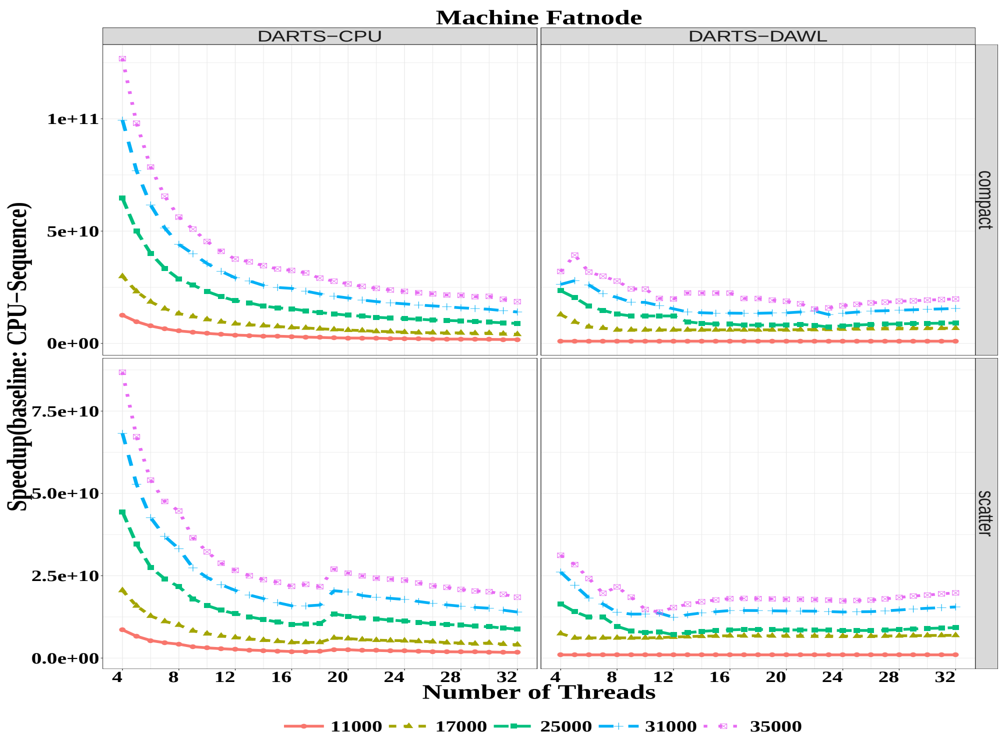

Figure 4.8: 2D stencil: Performance with a varying number of HW threads on `fatnode`. Time in nanoseconds.

is the same. When the threads number reaches a given threshold, increasing the number of threads does not improve performance—which is expected, because of memory conflicts. An important observation is that DAWL manages to quickly make use of the GPU to lower the overall execution time, as soon as it can. Section 4.4.3 will detail how the estimation model can help obtain a suitable threads number based on the application and hardware configuration.

4.4.3 Result of Profile-based Estimation Model

As described in session 4.3.3, the first two steps of building estimation model is collecting hardware architecture and application's run-time profile information. The required hardware information can be obtained through hardware spec or operating system command. To obtain application's run-time profile information for Machine Learning estimation model,

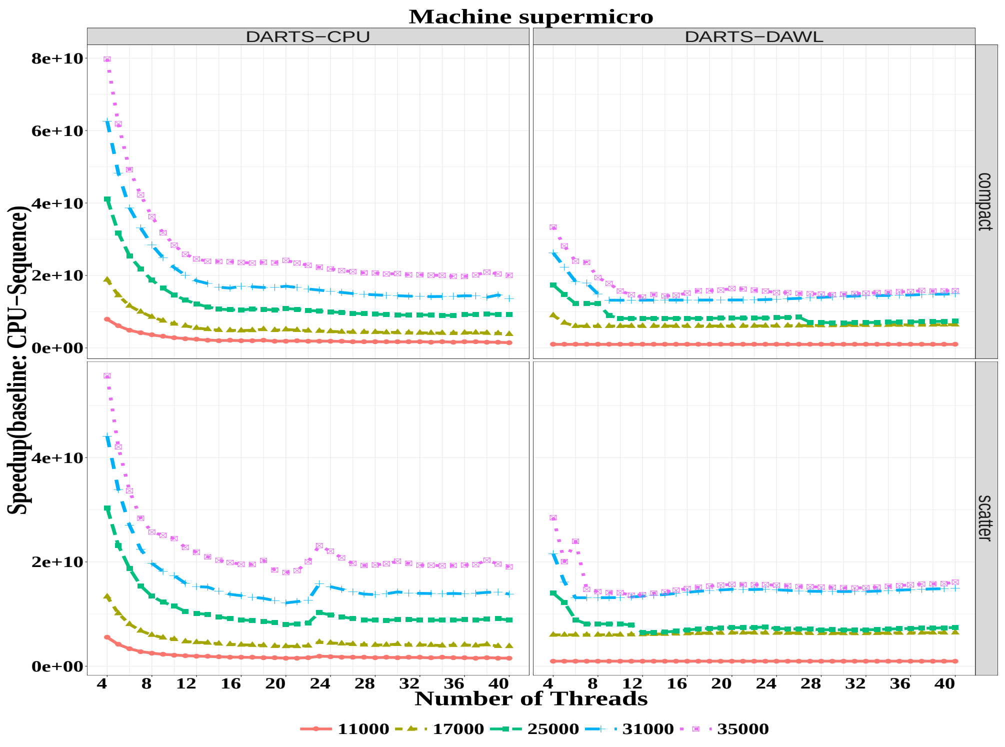

Figure 4.9: 2D stencil: Performance with a varying number of HW threads on `supermicro`. Time in nanoseconds.

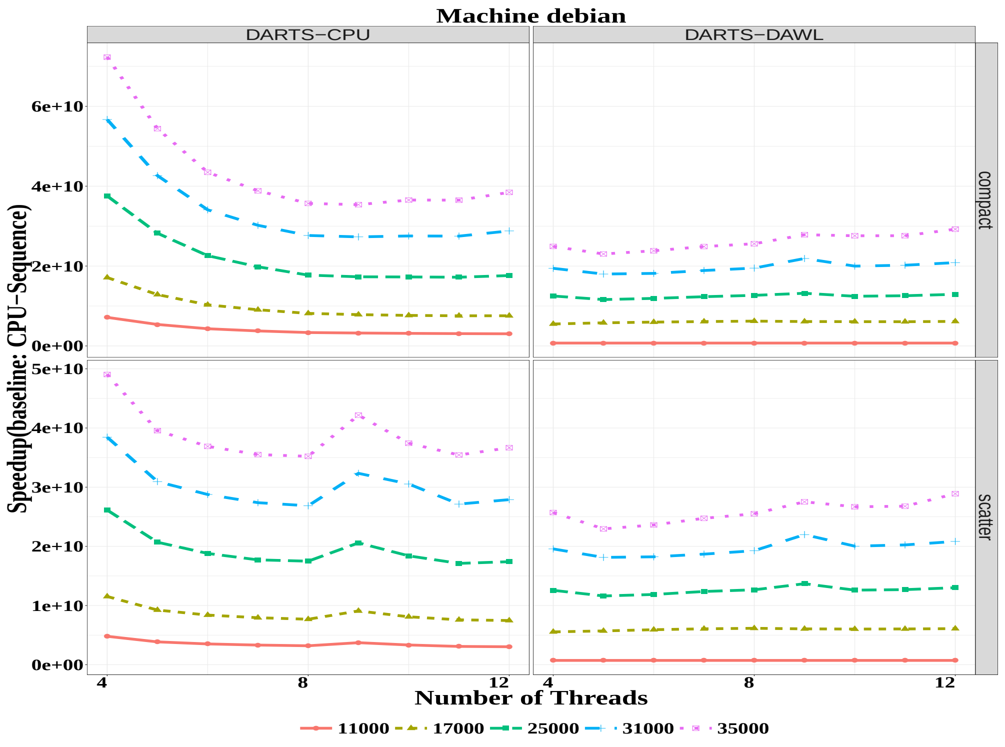

Figure 4.10: 2D stencil: Performance with a varying number of HW threads on `debian`. Time in nanoseconds.

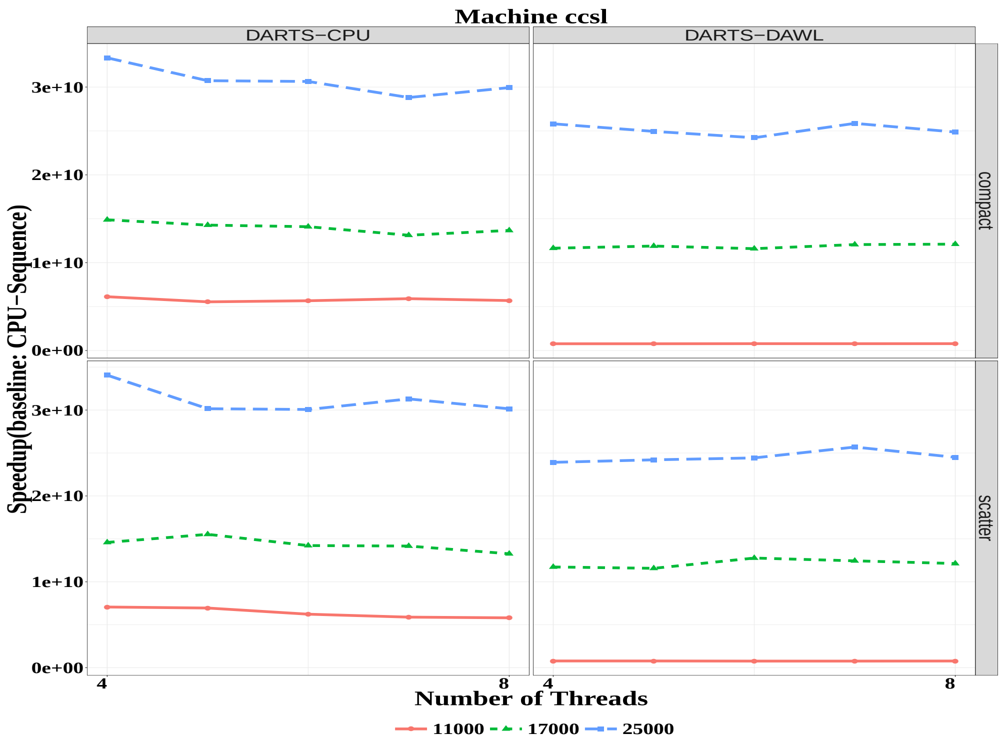

Figure 4.11: 2D stencil: Performance with a varying number of HW threads on `ccsl`. Time in nanoseconds.

sampling is a good way.

2D and 3D Stencil are using the same method to collect data. The Profile-based Estimation Model is designed for co-running applications, which means the target problem workload is larger than 17000 × 17000(2D stencil), 400 × 800 × 800 (3D stencil), with 2 GB of available GPU memory. Both, the data set size, and the number of time steps are leveraged to build estimation model.Considering the iterative nature of the application, 2D/3D stencil run with a small number of time steps (as training set) to predict (as test set) the execution time with a larger amount of time steps. The total sample set, including test and training data sets for 2D stencil, consists of several parameters: initial GPU workload slice (2000 × ∗, 4000 × ∗, 8000 × ∗); initial CPU workload: $GPU\_workload \times w$, $w \in \{0.5, 1.0, 1.5, 2.0\}$; time step (1, 4, 30); problem size (from 17000 × ∗ to 35000 × ∗ with steps of 2000); and number of CPU cores (4, 8, 16, . . .). Samples may vary between platforms, as hardware parameters differ a bit. The numbers of samples used used in the kernel ML profile were approximately 1000 (2D stencil) and 600 (3D stencil). These sample points were taken for leave-one-out cross-validation machine-learning profile model.

In the second step, the algorithm runs several matching models, as mentioned in section 4.3.3. It compares several strategies and picks the one that yields the best results, among linear regression, logistic regression (each using multiple meta-functions and polynomial functions), ensemble learning methods such as random forests, *etc.* In retrospect, it seems the model that finds the majority of the best matches is linear regression: its $R_{squared}$ are between 93 and 94%.

To measure the progress of the learning algorithm the Mean Absolute Percentage Error (MAPE) was used. Table 4.4 shows the MAPE of the linear model for each machine in the experiments.

Table 4.4: Mean Absolute Percentage Error

Machines	supermicro	fatnode	debian	ccsl
MAPE	7.41%	6.43%	1.68%	3.45%

The second goal in this statistical estimation model is to know which parameters had more impact in the construction of the model. Hence, the absolute value of the t-statistic is used for each model parameter and computed each parameter's importance in the model. Based on the experiment results that several parameters are enough to predict the performance of 2D stencils, table 4.5 list the most important features running on the 4 different machines, see table 4.1.

Feature selection was obtained using a black box approach. The decision is made for the ML algorithms based in the high correlation of all the parameters and ML modules. The selected features may totally different if the same applications run on different Hardware configuration platform, or the different applications run on the same or different Hardware

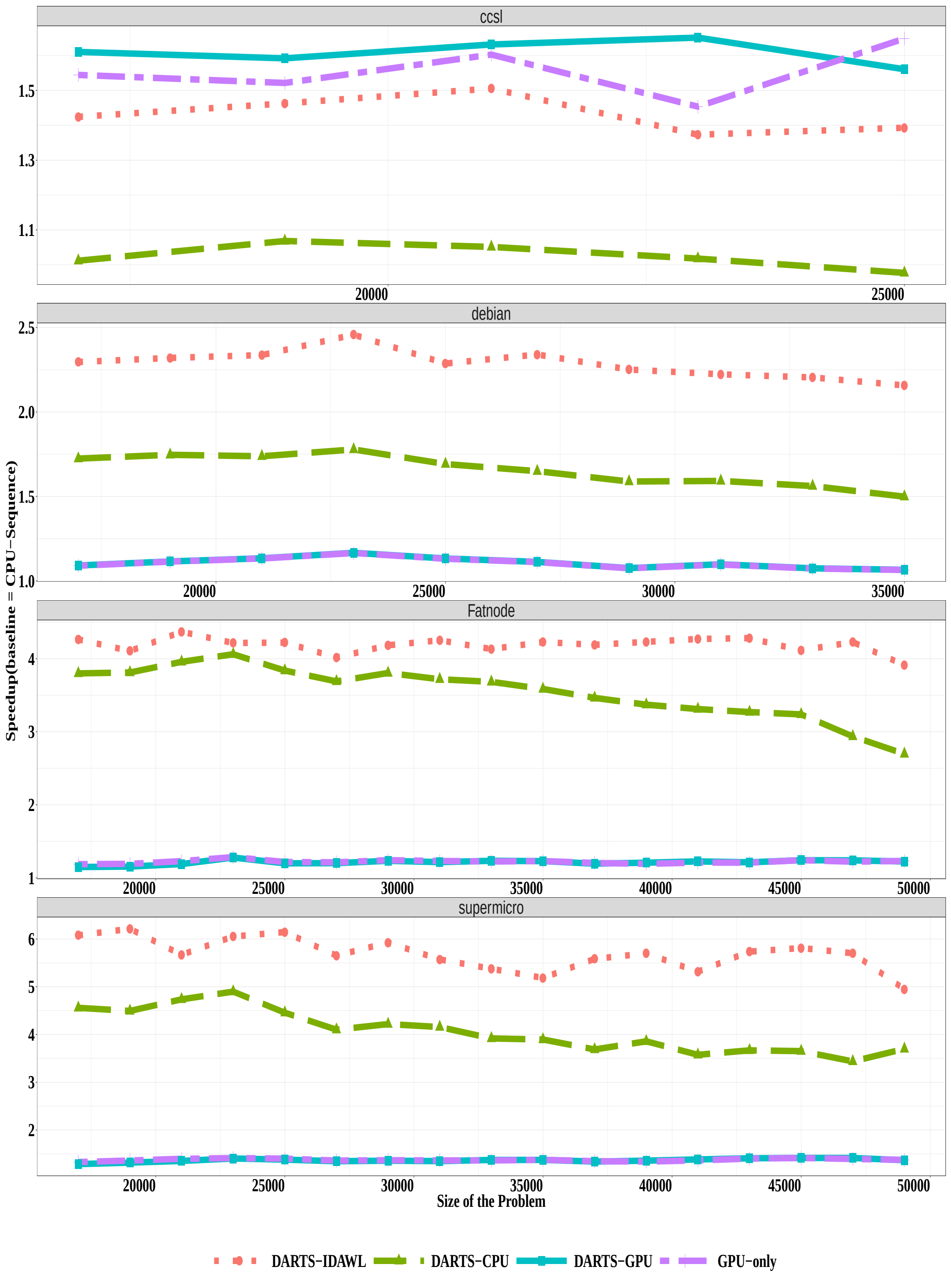

Figure 4.12: 2D stencil: Speedup when matrices are larger than 17K (IDAWL)

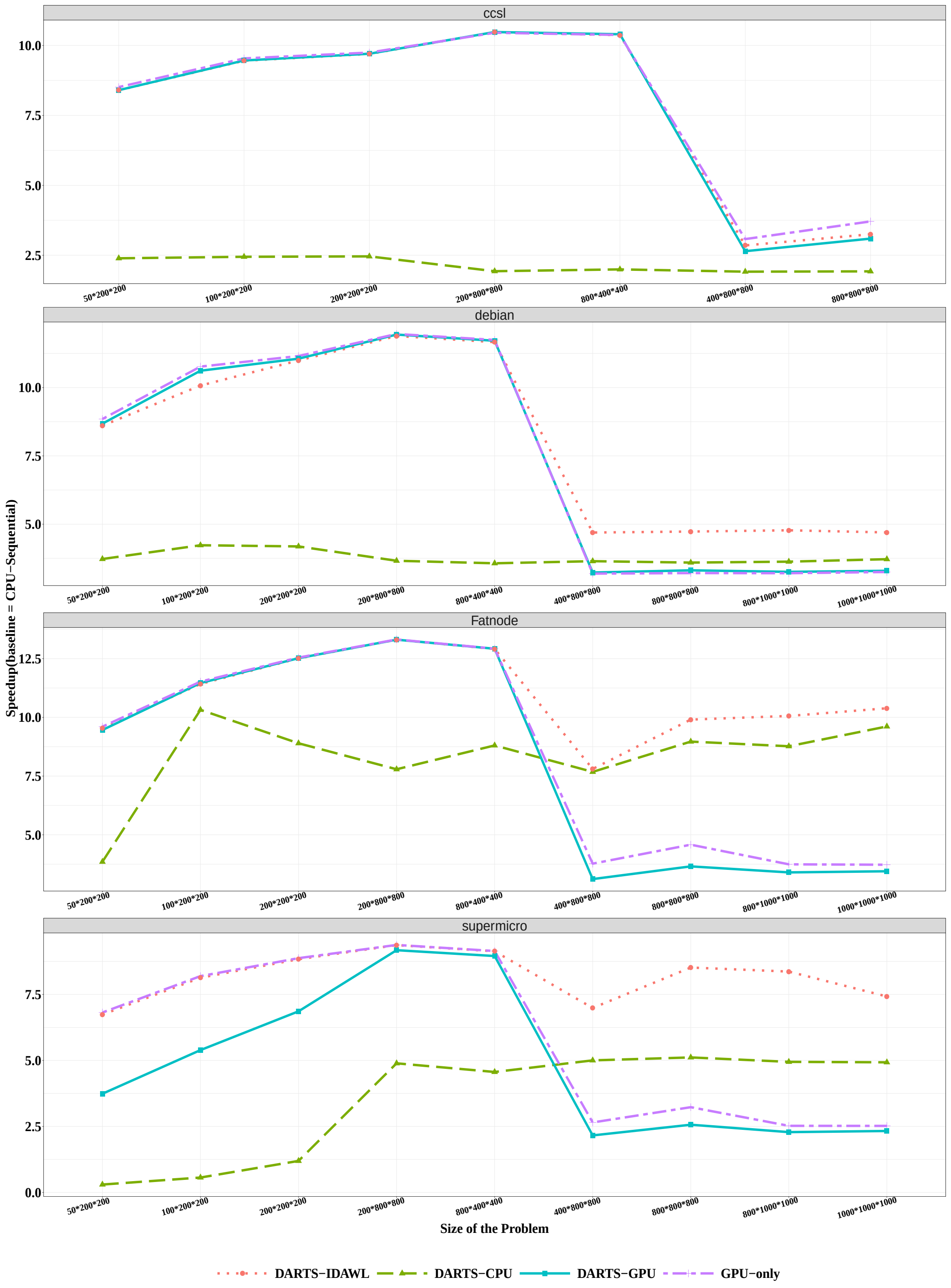

Figure 4.13: stencil3D: speedup (IDAWL)

Table 4.5: 2D Stencil: Important Features for ML Estimation Model

source	features
Hardware spec	The number of Sockets
	CPU clock frequency
	Total number of CPU threads
	L2 cache size
Operfile	L2 cache hit rate
	L2 cache miss rate
	IPC (Instruction Per Cycle)
Nvprof	GPU Grid/Block/Thread number
	GPU Thread-block occupancy
	Transfer bandwidth Host to Device
	Transfer bandwidth Device to Host
	The number of concurrent streams
	synchronization function

configuration platform.

Figures 4.12 and 4.13 show results for IDAWL. Compared to `DARTS-CPU`, which always uses all the CPUs, this implementation uses at most half of the CPUs (depending on the system). The new scheduler can reach up to 6× speedups compared to sequential runs, 1.6× speedup compared to the multiple core version, and 4.8× speedup compared to the pure GPU version in the 2D case. In the 3D case, `DARTS-DAWL` uses as many threads as `DARTS-CPU`, and reaches speedups up to 9× compared to the sequential version, 1.8× against multi-cores, and 3.6× against a pure GPU version. Comparing Figures 4.12 and Figure 4.6, and Figure 4.13 with Figure 4.7, it is clear speedups are not always obtained using profiling. This is especially true around *drop points*. Drop points are unstable points, and refer to multiple co-running hardware/software conflicts parameters, which this machine learning estimation model did not take into consideration. Moreover, the ML algorithm can be further improved by combining classifier algorithms and neural-network to this learning estimation model.

4.5 Related Work

Teodoro *et al.* [74] have proposed and implemented a performance variation aware scheduling technique along with an estimation optimization model to collaboratively use CPUs and GPUs on a parallel system. A number of scheduling methods or library [75–77] were combined with StarPU [78], a task programming library for hybrid architectures based on task-dependency graphs, to perform scheduling and handle task placement in heterogeneous systems. Panneerselvam and Swift proposed Rinnegan [75], a Linux kernel extension and run-time library, and implemented and validated that decisions of where to execute a task must consider not only execution time of the task, but also current heterogeneous system conditions. Sukkari *et al.* [76] proposed an asynchronous out-of-order task-based formulation of the Polar Decomposition method to improve hardware occupancy using fine-grained computations and look-ahead techniques. Gaspar *et al.* [79] have proposed a general framework for fine-grain applications-aware task management in heterogeneous embedded platforms. This framework was specifically developed for run time performance monitoring and self-reporting, and tackles OS task management and system resource utilization. In StarPU, the user provides different kernels and tasks for each target device and specifies inputs and outputs of each task. The run time ensures that data dependencies are transferred to the devices for each task. Furthermore, the user can specify explicit dependencies between tasks.

The main challenge of the load-balancing mechanism is to precisely divide workload on processing units. A simple heuristics devision approach may result in worse performance. Belviranli *et al.* proposed a dynamic load-balancing algorithm for heterogeneous GPU clusters named the Heterogeneous Dynamic Self-Scheduler (HDSS) [80]. Sant'Ana *et al.* described a novel profile-based load-balance algorithm [77] named PLB-Hec for data-parallel applications in heterogeneous CPU-GPU clusters. PLB-HeC algorithm performs an online customized estimation of performance curve models for each devices (CPU or GPU). Like a typical data-parallel application, data in PLB-HeC is divided in blocks, which can be concurrently processed by multiple threads. The granularity of the block size for each processing unit is crucial for performance: incorrect block sizes will produce idleness in some processing units and reduce performance. To get good block sizes, PLB-HeC solves the problem in three phrases: first, it dynamically computes performance profiles for different processing units at runtime; second, using a non-linear system model, they determine the best distribution of block size among different processing units; third, they re-balance the block size during execution. PLB-Hec obtained higher performance gains with more heterogeneous clusters and larger problems sizes.

All the works presented above rely on StarPU to implement their various strategies. Of all of them, Belviranli *et al.* and Sant'Ana *et al.*'s work are closest to IDAWL : they rely on online profiling, or resort to some ML techniques to perform load-balancing decisions. However, this work and most of the previous ones tend to focus on loosely synchronized parallel workloads, where specific tasks are often run only a specific type of processing element (*e.g.*, CPU or GPU). On the contrary, IDAWL focuses on workloads that are iterative in nature, feature heavy data dependences, and require regular and possibly frequent synchronization operations between the device and the host. The work itself is "homogeneous", but it can be run on either the host or the device, depending on their state of idleness, the remaining work size to perform, *etc.*

Werkhoven *et al.* proposed an analytical performance model that includes PCIe transfers and overlapping computation and communication [67]. A roofline model [81] was used to module the performance of GPU kernels execution time and PCIe transfer time. Their model's main features are the type of synchronization, stream number, and the number of copy engines for GPU and PCIe [82, 83].

Lutz *et al.* proposed PARTANS, an autotuning framework built for CPUs and GPUs [84]. They executed different shapes of stencil computations over two nodes with multiple GPUs. They analyzed the impact of different data transfer structures based in the stencil shapes across the PCIe bus. They designed a heuristic which determines the number of GPUs to use.

Luk *et al.* [85] proposed an adaptive mapping approach. It relies on a new API which maps work to either Intel Thread Building Blocks or CUDA. To handle the communication-synchronization problem between CPUs and GPUs, Lee *et al.* proposed SKMD (Single Kernel Multiple devices) [86]. It can transparently orchestrate a single kernel execution acros asymmetric heterogeneous devices regardless of memory access patterns. O'Boyle *et al.* proposed a machine-learning based approach to determine whether to run OpenCL code on GPU or OpenMP code on multi-core CPUs at run time [87] and presented a runtime framework [88] to decide whether to merge or separate multi-user OpenCL tasks running the most suitable devices in a CPU-GPU systems.

These works rely on offline training models. Kaleem *et al.* [89] presented a scheduling techniques for integrated CPU-GPU processors based on online profiling.

IDAWL dynamic scheduling approach differs in the following ways: First, it focus on the synchronization between CPUs and GPUs; second, the communication between CPUs and GPUs play a pivot role in IDAWL approach; third, this approach is neither purely offline nor online. it combines two models together where an offline ML model provides an initial workload allocation, and DAWL dynamically adjusts workload balancing to compensate offline-ML inaccuracies, resulting in real-time adaptation.

4.6 Observations

An iterative scheduling algorithm, IDAWL as described in section 4, were designed to better load balance tasks in a heterogeneous system. Further, it leverages a profile-based approach based on machine learning, which allows it to converge faster to a better load-balanced schedule. The ML's estimation model can obtain a customized initial workload on various heterogeneous architectures, as well as how many devices are necessary. As a case study, we used a 5-point 2D and a 7-point 3D stencils using an event-driven run-time system, *DARTS*.

This works also evaluates the limits of co-running on heterogeneous systems. Experimental results show our approach is at worst on par with a pure GPU approach (when data fits fully in the GPU), or yields speedups up to 1.6× against a multi-core baseline, and 4.8× against a pure GPU execution. In the 3D case, DAWL reaches 1.8× against multi-cores, and 3.6× against pure GPU.

The key contributions of section 4 are:

1. IDAWL, an Iterative Dynamic Adaptive Work-Load balancing algorithm for heterogeneous systems, which is combined with and provides data for an offline machine-learning based profiling system.IDAWL's efficiency with stencil-based kernels were evaluated: they feature a high-degree of data dependence, and require regular host-accelerators synchronizations.

2. On top of evaluating raw performance of a co-running solution, the limits of co-running are evaluated with respect to data input sizes fed to our kernels: (1) if resorting to using *all* available compute resources always yields better results were also evaluated; and (2) Which parameters matter when deciding where to schedule a task in a heterogeneous context were also provided.

Future work includes augment power-consumption parameters to enrich a IDAWL and determine good trade-offs between performance and power on heterogeneous architectures. We will also integrate more parameters in our machine learning algorithm to improve its real-time ability to allocate work to heterogeneous processing elements.

Chapter 5

Stream-based Event-Driven Heterogeneous Multithreading Model

5.1 Introduction and motivation

Streaming applications, where the computation can be naturally expressed as *streams*, are widely used in a lot of important areas, such as scientific computations, embedded applications, as well as the emerging field of social-media processing. Program execution models centered on streams have been studied by many researchers and have been an active field of research for the past 30 years [5–9, 90, 91]. The most relevant early work on streams is the data flow execution model pioneered by Dennis [10, 11], the Synchronous Data Flow(`SDF`) model [12, 13] and Program Dependence Graph(PDG) model [14]. Other work include data flow software pipeline [15–18]. However, these models do not address the parallelism, resources utilization and communication problems existing in highly heterogeneous and hierarchical system. Moreover, because of the physical limits that core count per chip continues to increase dramatically while the available on-chip memory per core is only getting marginally bigger. In this case, data locality, already a must-have in high-performance computing, will become an even critical point in streaming processing since smoothly data movement play a pivotal in streaming processing.

Heterogeneity has been studied by a number of researchers [92–97] but many of these efforts are only targeted at isolated dimensions of heterogeneity, either the cores, the memory, or the interconnects in isolation. While, heterogeneity can be cooperatively harnessed at cross-cutting scope, spanning heterogeneous cores, hybrid memory hierarchies and re-configurable and hybrid interconnect architecture and software. Especially for stream processing, it is paramount to keep the streaming data flow at high-speed and maintain performance efficiency, (*e.g.*, throughput, delay, *etc.*) and energy efficiency on heterogeneous system. To reach this goal, cross-layer cross-cutting design methodology, including algorithm design, programming models, architecture design, and system software design, are necessary to explore parallelism and deliver scalability, since parallelism is ubiquitous and found at many levels of the entire hardware-software stack.

The stream programming model offers a promising approach for exploiting parallelism for many-cores architecture. Firstly, it can explore a coarse-grain parallelism [98, 99]. Streaming parallelism is located at the data flow module-level. The multiple data flow modules can execute concurrently on many-cores architecture; Secondly, it also can explore the fine-grain parallelism within the body of an individual data flow module; Thirdly, both coarse and fine grain parallelisms can be explored at the temporal and spatial dimensions.

To efficiently exploit parallelism and delivering scalability in stream processing, a number of challenges must be overcome. To summarize,these challenges include: exploiting coarse- and fine-grain level exploitation of parallelism, designing and using heterogeneous computing environments, dealing with heterogeneous workloads, and developing efficient adaptive memory management mechanisms targeted at minimizing data movement (thus enhancing locality) for the sake of both performance and energy efficiency.

Section 5 proposes an approach, stream-based event-driven heterogeneous multithreading model, to solved parallelism, resource allocation and streaming data flow movement problems in High-Performance Computing. section 5.2 reviews the two level parallelism of stream processing, and proposes new streaming program execution model (`SPXM`); section 5.3 describes the details of design `SPXM`, including streaming *Codelet* Model, and streaming runtime system; section 5.4 describes the related work of stream processing.

5.2 Streaming Program Execution Model (SPXM)

The stream-based fine-grain program execution model defined in this section is based on the *Codelet* Model, see section 2 for more details, which provides a basic framework for an asynchronous, event-driven parallel program execution model.

5.2.1 Two Levels Parallelism and Data Locality

The stream programming model offers a promising approach for exploiting parallelism for many-core architecture. Firstly, it can explore a coarse-grain parallelism [98, 99]. Streaming parallelism is located at the data flow module level and the multiple data flow modules could execute concurrently on multicore architecture. Furthermore, non-strict [100, 101] aspect of data flow models(*i.e.*, I-Structures [102]) can be explored to facilitate out-of-order stream processing; Secondly, it also can explore the fine-grain parallelism within the body of an individual data flow module which often contains a collection of loops. Multiple parallelism approaches, *i.e.*, Data-level (Thread-level) parallelism [103], Task-level Parallelism [104], Instruction-Level Parallelism (data flow software pipeline [15–18]), can be leveraged to optimize performance; Thirdly, both coarse and fine grain parallelisms can be explored at the temporal and spatial dimensions.

In streaming processing, maximize locality and minimize data movement are effective optimization approach which can be used in both two parallel levels. Data can be classified into two categories a) Some data are continuously streaming through the data channels in the data flow graph. Conceptually, a (FIFO) buffer of a certain size should be allocated to such a channel to accommodate the throughput and delay requirement, and b) Some data, that are not streaming, should be placed in the shared memory hierarchy in order to exploit locality and minimize data movement. The new streaming *Codelet* model,describe in section 5.2.2, is developed to handle storage buffer allocation between stream producer-consumer channelsboth at the coarse-grain and the fine-grain levels.

5.2.2 Streaming ***Codelet*** Model

Stream data flow execution model pioneered by Dennis [11, 20], the Synchronous Data Flow(SDF) model [12, 13] and Program Dependence Graph(PDG) model [14], where each node represents a computation task (actor) and each arc represents the communication between tasks. During program execution, each actor, which has an independent instruction streams and address space, must fire repeatedly in a periodic schedule. However, these models, including *Codelet* model which is built up on SDF, do not address the parallelism and resources utilization problems existing in highly heterogeneous and hierarchical system.

In the new streaming *Codelet* model, a program(application) is partitioned into modules which are connected by communication channels. Here, module stands for streaming *Threaded Procedure*`(STP)`, see section 2 for more details about *threaded procedure*. A module can be seen as a group of streaming *Codelet*s connected by intra-module stream channels within a module. Modules are themselves connected through inter-module stream channels. Each module contains at least one streaming *Codelet*. A stream channel is modeled as an abstract FIFO queue where Direct Memory Access (DMA) can help speedup when the modules refer to big chunk of data movement. Each stream module or stream *Codelet* is an autonomous computation unit which consumes data at a given rate from the input channel and produces data at a given rate to its output channel, while this production-consumption rate can be static or dynamic and be determined at compile or run time. At execution time, each stream computation module is ready to run only if there are enough data items in the input channels and enough buffer space in the output channels.

Furthermore, to explore scalability, a module (`STP`) itself also can be as a component of other module called upper level module. This feature makess streaming *Codelet* model smoothly applicable to the hierarchical heterogeneous many-core system. A module can either be mapped onto one cluster made up of computing engines, onto one chip made up of clusters, onto one computing node made up of chips or even onto a internet. Streaming *Codelet*s belonging to the same module can be mapped to different computing engines within a given cluster (*e.g.*, a data-parallel portion can be mapped to a vector-friendly computing engine, while a more control-irregular part of the graph may be better suited to a general-purpose computing engine). The need to pass data between *Codelet*s may induce unreasonably long latencies, thus requiring the use of intra-module stream channels to specify buffer sizes, buffer address and , *etc.*. On the other hand, within a given portion of the machine, latencies will be essentially non-existent, thus only requiring that streaming *Codelet*s signal the availability of new data, much like the original *Codelet* model proposes.

Streaming *Codelet*s are *Codelet*s with some key additional properties: for example, an interface must describe buffer sizes and latencies to ensure steady state scheduling preferences in terms of resources. This is required to implement software pipelining. Streaming *Codelet*s are also by nature most likely going to be persistent. In addition, and interface of streaming *Codelet* should express interconnections between *Codelet*s, since some may be mapped to different clusters. Furthermore, a device attribute should clearly indicate that which type of computing engine is recommended for this streaming *Codelet*. Based on these `requirements`, the more implement details can be found in section 5.3.

Both the modules (STP) and the streaming *Codelet*s they contain are event-driven, with the arrival of data as the primary event to satisfy, thereby potentially exploiting both coarse-grained and fine-grained parallelism. At the module level, pipelining and task parallelism can be exploited between stream computation modules. Further, each module may also contain any degree of parallelism. This fact should be fully exploited by applying the fine-grained streaming *Codelet* model to address performance and scalability needs.This is an important point, considering the rapid increase in heterogeneity and hardware chip-level parallelism.

5.2.3 Streaming *Codelet Abstract Machine* Model

The original *Codelet* Model relies on an *Codelet Abstract Machine* Model (CAM), see section 2, which is hierarchical and distributed, and provides two types of engines: the computing engines called CUs, which perform the actual work, and the scheduling engines called SUs, which ensure the correct scheduling and resource allocation across the machine. Computing engines are grouped into clusters along with at least one scheduling engine.

Targeting at stream processing on heterogeneous and hierarchical system, Streaming *Codelet Abstract Machine Model* (SCAM) extends original one level CAM Model to two levels to better fit future architectures which exhibit a high diversity of computing capabilities. Hence, the high-level layer will feature clusters of computing engines, where streaming *Codelet* modules will be mapped. However, as opposed to the original CAM, SCAM's clusters are expected to widely differing capabilities and levels of parallelism. While the *nominal* capabilities and/or degree of parallelism of a given cluster are known statically when starting a given application, there are various reasons to force it to expose a different set of capabilities over time: faulty components, high-contention of part or all of the cluster, elevated power consumption, *etc.* Thus, the assignment of modules across a machine will partly rely on information only available at run-time, while additional properties must be defined and added to the basic `CAM`. This high level layer, which exposes clusters of computing engines without unveiling the engines themselves, will be visible to the high-level programmer. Specific scheduling engines are dedicated to the distribution of work among the clusters.

The second layer of the `SCAM` is a low-level abstract machine model, visible to both the compiler and the runtime system. It must identify the type of capabilities embedded in clusters–although it is sufficient for the compiler to know what kind of support it can expect from the low-level `CAM`. This allows it to generate the adequate code variants. The computing engines contained in a cluster range from specific functions provided by a low-level component of the cluster to fully general purpose computing units. Here again, a scheduling unit is in charge of mapping the portions of a given module to the available computing engines.

How to put together the computing engines (cpu cores, GPUs, accelerators, FPGAs, *etc.*) to operate in the high power/energy-efficient domains and high performance domains while providing full support for the SPXM is a key question in design `SCAM`. To achieve this goal, on-chip and off-chip memory systems should be optimized to feed the computing engines and the communication structure that transports the streams. In some cases. These two elements (memory and communication, especially communication between different types of resources) introduce overhead orders of magnitude higher than the overhead of the computing engines. So, in stream processing, minimize latencies and global data movements play a pivotal role.

The Runtime system of `SCAM`, called *streaming DARTS*, will manage parallelism, memory management, communication traffic, *etc.* It will allocate stream modules to the appropriate computing engines (*e.g.*, a streaming *Codelet* containing a vector operation should go to the cluster which has GPUs on it) and perform dynamic resource management to ensure that processing and memory resources are not left idle when there are tasks/streaming *Codelet* available for execution. It will creates stream channels for the communications between the modules/streaming *Codelet*s. Finally, it needs to schedule the streaming *Codelet*s to the computing engines in order to exploit the fine-grained parallelism. Dynamically adjust resources assignment besed on the run-time situation is also necessary function for the runtime system scheduler to fully utilize the computing resources.

5.3 SPXM Design

5.3.1 Streaming *Codelet*

Streaming *Codelet* locates in the fine-grain level of `SPXM` and stands for the fine-grain task. It will be mapped to and run on computing engine when all the synchronization requirements are satisfied. As described in section 5.2.2, it is *Codelet* with some key additional properties(streaming properties):

1. ID. Instead of option in *Codelet*, it is necessary in streaming *Codelet*. It will be used to construct streaming *Codelet* graph (`SCG`), to build up the stream data and message channel. Streaming *Codelet*'s ID is unique in current streaming module. ID also can be used by scheduler of streaming runtime system of `SCAM`. Scheduler can assign and pin the streaming *Codelet* to specific computing engine since it will be fired (run) repeatedly in a periodic schedule during stream processing. The pinning operation can help streaming *Codelet* utilizes the data locality and reduce unnecessary data movement.

2. Parent streaming module (`STP`) ID. it can help streaming *Codelet* locates itself in the whole system since the `SPXM` is a logical hierarchical Model.

3. Data connection slot. It helps to construct `SCG`. Some streaming *Codelet*s may be mapped to different clusters of computing engines. Buffer size, buffer address and production-consumption rate also locate in this slot. Each arc of the `SCG` represents

one data communication channel between tasks. The arc can connect either streaming *Codelet*s or streaming modules(`STP`s). To help huge bulk data movement between streaming modules/*Codelet*s, `DMA` *etc.* components can be added to the slot. One streaming *Codelet*/module can contain one or multiple data connection slots.

4. Synchronization slot. It is a basic component of event-driven model, will be equipped with more functions, such as control production-consumption rate in further to control the fire rule of streaming *Codelet.* Same with data interconnection slot, One streaming *Codelet*/module can contain one or multiple synchronization slots. The consumer can be fired only if all its synchronization slots' requirements are satisfied. Different with data connection slot, synchronization slot only exis in consumer side.

5. Message connection slot. Same with StreamIt [9] language, `SACM` also provides a dynamic messaging system for passing irregular and low-volume control information between Streaming *Codelets*s and streaming modules. Messages are sent from one streaming *Codelet* to other streaming *Codelet*s located in the same streaming module or to current streaming module which can broadcast the messages to all its group members (streaming *Codelet*s). Messages also can be transferred between streaming modules. There are several types of messages: a) change the parameters,*e.g.* production-consumption rate; b) change running status,*e.g.*, from `RUNNING` to `STOP/SLEEP` *etc.*. For example, if the consumer encounters some issues (the usable resources are sharply reduced because of power limitation) which cause its data processing speed unstable and dropping quickly, a `STOP` or `SLEEP` message will be sent to its producers once the accumulated data over the buffer limitation. Then the producer will adjust its status based on the received message; c) change allocated device,*e.g.*, change low computing capability of current computing engine to high computing capability computing engine if current one can't satisfy the performance requirement. For example, if consumer is always in `WAITING` status, it can send `CHANGING` message to scheduler in module to

request reassign its producer's computing engine.

6. Device attribute. It indicates that the current streaming *Codelet* will run on which types of computing resources. Up to now, it is static assign to the streaming *Codelet*, this work should be done by compiler in the future.

7. Fire rules. One streaming *Codelet* can be fired need to satisfy two requirements: first, the synchronization requirements should be satisfied, which means the current streaming *Codelet* have enough data in the input channels and enough buffer space in the output channels; second, there is no `STOP` or `SLEEP` messages were sent to current streaming *Codelet*.

8. History record. It records the past execution time, it can be used to build estimation model in the future.

9. Reset status. It will record all the configuration/status current streaming *Codelet*s and can be used when the the streaming *Codelet* wake up from `STOP` and `SLEEP` status.

10. Latency. It will be used in pipeline stage.

5.3.2 Streaming Module (*Threaded Procedure*)

Streaming module locates in the coarse-grain level of `SPXM` and stands for the coarse-grain tasks. One stream stage can contain one or more streaming modules depending on `SCG` and available computing resources. It contains all the streaming properties of streaming *Codelet*, but with small differences. Except these streaming properties, it is based on *Threaded Procedure*:

1. console component. Streaming module, containing a collection of streaming *Codelet*, will run on the cluster. One of main functions of streaming module is synchronizing all

its group members (streaming *Codelet*s or low level streaming modules, as described in section 5.2.2). The console component records all the information of its group members, it also provides search function.

2. data connection slot. it sets up a data bridge between streaming modules, and between streaming module with its group members. Special communication components,such as `DMA`, can be added to help speed up the bulk data movement.

3. Message connection slot. it can broadcast/receive messages to/from all its group members, send/receive messages from its peers(other streaming modules), and transfer cross-layer message.

5.3.3 Runtime Stream Scheduler

The high level runtime stream scheduler will access which clusters/computing resources are available and decide on which would be the best cluster to run the various streaming modules. Once the clusters are selected, the runtime system then determines which of its computing resources are current usable and map the `SCG` to the available hardware resources. The runtime scheduler handles the case where a specific accelerator is not available to the computation. There are various reasons for the unavailability of a specific processing unit: the accelerator is already busy, or, for execution-time reasons, the scheduler did not assign the *Codelet* to a tile that featured such an accelerator. In this case, the local scheduler will select the code variant of the best-suited resource (e.g., an FPGA version if such a device is ready to be used), and insert the output stream address to feed data to the next streaming *Codelet*s which will then apply their data transformation process in the pipeline. During run time, the local scheduler will change the binding (between resources and *Codelet*) if `CHANGING` message is obtained, see example in section 5.3.4.

The scheduler supports both balanced and unbalanced `SDFG`. If converter from unbalanced

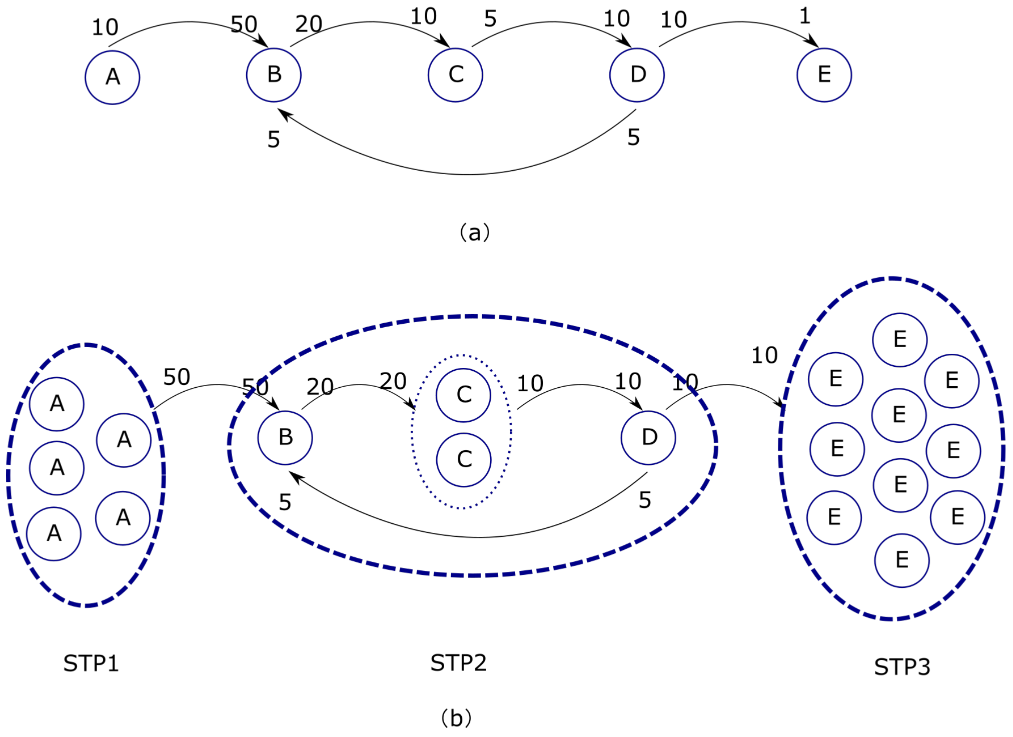

Figure 5.1: sheduler: unbalanced `SDFG` to balanced `SDFG`

`SDFG` to balanced `SDFG` option is activated, it can automatic convert unbalanced `SDFG` to balanced `SDFG` and then assign and schedule tasks to corresponding computing resources. In Figure 5.1, (a) stands for the original *SDFG* (b) stands for the converted `SDFG`, while the tasks are grouped into three groups (3 *STP*s). How to map task (circle in the Figure) depends on the available resources. Unbalanced `SDFG` is also supported by the scheduler. A `STOP` or `SLEEP` message will be sent to producer when too many data are accumulated into the consumer's buffer. `WAKEUP` message will be sent from consumer to producer when the accumulated data in consumer's buffer reach to a suitable level. Consumer will automatically enter into `WAITING` status when there are no enough data available based on the features of event-driven tasks.

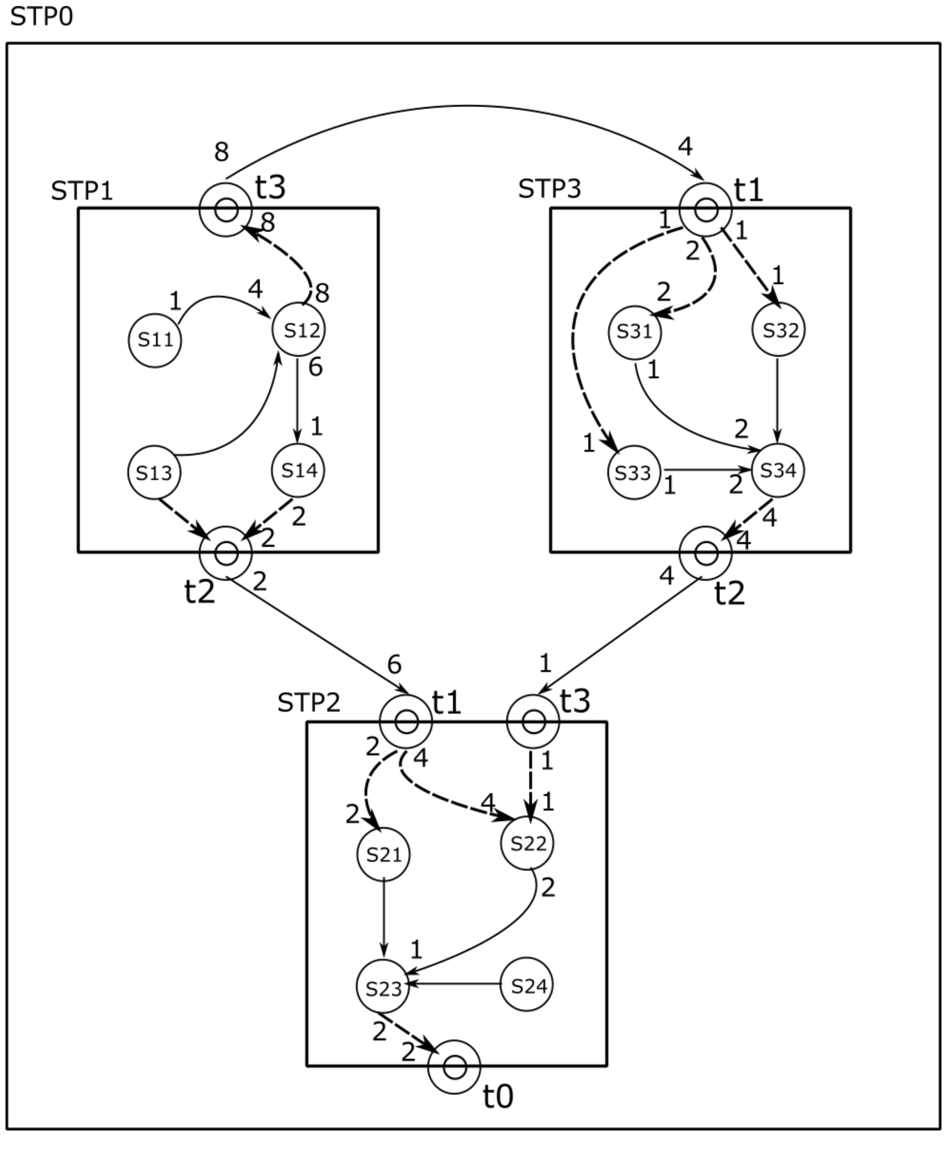

Figure 5.2: Example: streaming *Codelet* graph (SCG)

5.3.4 Detailed Example

Figure 5.2 shows an example of `SCG`. As described in section 5.2.2, every component,including streaming module(*Threaded Procedure*) and streaming *Codelet*, has an unique ID. In this Figure, rectangle stands for streaming module(`STP` in Figure 5.2); single circle stands for streaming *Codelet*(`S` in Figure 5.2); double circle stands for the transfer gate between *STP*s (*i.e.*,`t1` in `STP3` stands for `STP1`'s transfer gate in `STP3`.); the solid line stands for the same layer connection, such as the line connecting `S11` to `S12` in `STP1` group, and the line connecting `STP1` to `STP3` in `STP0` group; dotted line stands for the across layers connection (*i.e.*, the line connecting `S12` to `t3` means `S12` produce data to `STP3`); the number on the line stands for the production-consumption token(data); no number on line stands for producing or consuming one token. As shown in Figure 5.2, multiple streaming *Codelets* can connect to the same transfer port, and one *STP* can own multiple transfer port.

Table 5.1: streaming *Threaded Procedure* `STP0` attributes based on the Figure 5.2

attributes	**content**
ID	0
parent ID	–
group members	STP1
	STP2
	STP3
shared	share0
receivers	–
Final *Codelet*	default:`SLEEP`
history	execution time
latency	number0
device	cluster/node/... (ID)

Only data connection graph is shown in Figure 5.2. The message connection graph will be set up based on the runtime situation. For example, in the Figure 5.2 (when unbalanced scheduler is used), Streaming *Codelet*(`S12`) produce 6 token while Streaming *Codelet*(`S14`) only consume 1 token. Figure 5.3 shows the mapping graph of `S12` and `S14`. There are three

```
inst STP0{

    /*
     * setshare: set up shared variables, data space
     *    shared variable/data can be accessed by all the group
        member
     */
    setshare share0;

    /*
     * add: add components
     */
    add STP1 to STP0;
    add STP2 to STP0;
    add STP3 to STP0;

    /*
    * setcxn(producer, ptoken, consumer,ctoken,address)
    *   ptoken: producer produced token
    *   ctoken: consumer consumed token
    *   address: option
    */
    setcxn(STP1,8,STP3,4,addr1);
    setcxn(STP1,2,STP2,6,addr2);
    setcxn(STP3,4,STP2,1,addr3);

    /*
     * setlatency: set the largest latency (for pipeline stage)
     *    default latency: Infinity
     */
    setlatency( number0 );

}
```

Listing 5.1: pseudocode of Figure 5.2 `STP0`

```
    inst STP3{

        setshare share3;

        add S31 to STP3;
        add S32 to STP3;
        add S33 to STP3;
        add S34 to STP3;

        setcxn(S31,1,S34,2);
        setcxn(S32,1,S34,1);
        setcxn(S33,1,S34,2);

        /*
        * setcrosscxn: producer and consumer are in differ layers
        * level = producer layer - consumer layer
        * STP1~3: layer = 1; STP0: layer=2
        * S: layer = 0
        */
        setcrosscxn(t1,1,S33,1,level=1);
        setcrosscxn(t1,2,S31,2,level=1);
        setcrosscxn(t1,1,S32,1,level=1);
        setcrosscxn(s34,4,t2,4,level=-1);

        /*
        * setsync(producer, ctoken, max)
        *   set synchronization slot(only consumer)
        *   max: optional, the max number of token can keep in the
            synchronization slot
        */
        setsync(STP1,4);
        setsync(S34,4);

        setlatency( number3 );
    }
```

Listing 5.2: pseudocode of Figure 5.2 STP3

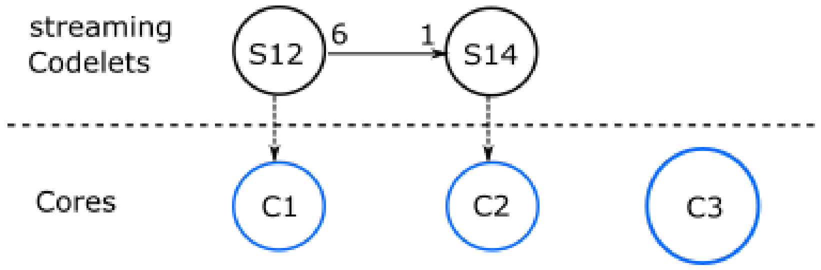

Figure 5.3: mapping streaming *Codelets* to cores example

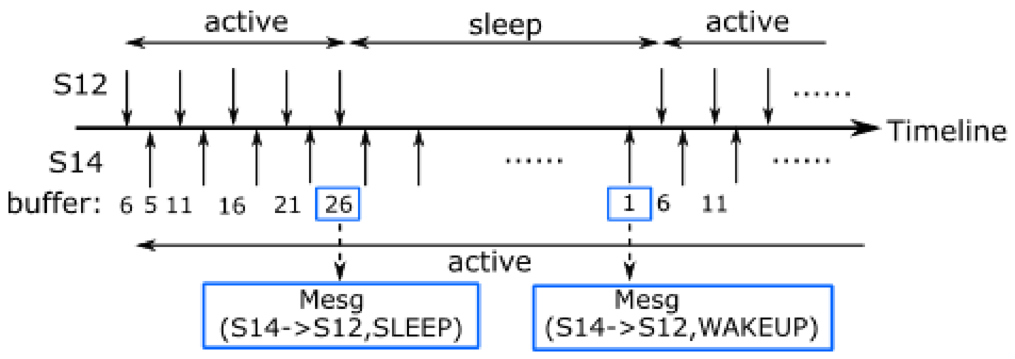

Figure 5.4: message example1

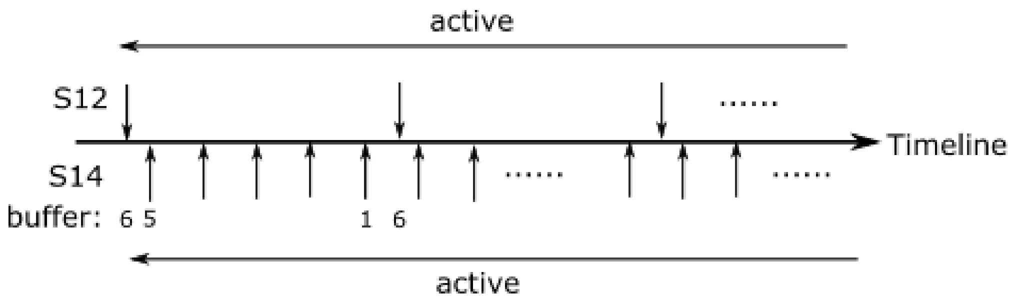

Figure 5.5: message example2

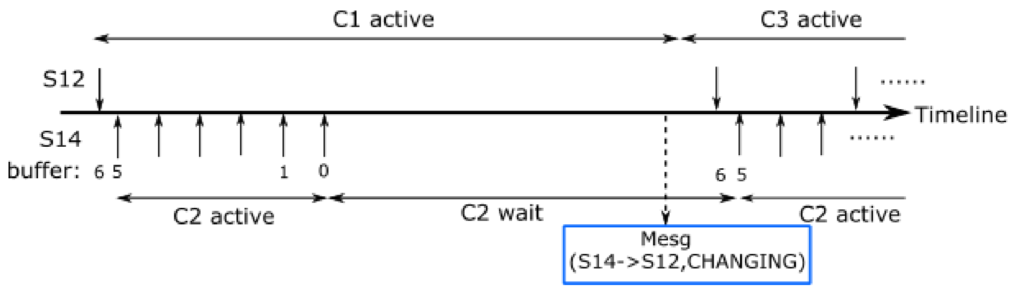

Figure 5.6: message example3

Table 5.2: streaming *Threaded Procedure* `STP3` attributes based on the Figure 5.2

attributes	**content**
ID	3
parent ID	STP0
group members	S31,S23,S33,S34
data connection slot	(STP1, 8, STP3, 4, addr= addr1)
	(STP3, 4, STP2, 1, addr= addr3)
	(t1, 1, S33, 1, addr=, level=1)
	(t1, 2, S31, 2, addr=, level=1)
	(t1, 1, S32, 1, addr=, level=1)
	(S34,4, t2, 4, addr=, level= -1)
synchronization slot	(STP1,ctoken = 4,max=INF)
	(S34,ctoken = 4,max=INF)
mesg connection slot	default: running
shared data/variables	share0,share3
receivers	STP2
history	execution time
reset status	–
latency	number3
device	cluster/node/... (ID)

Table 5.3: streaming *Codelet* `S12` attributes based on the Figure 5.2

attributes	**content**
ID	12
parent ID	STP1
data connection slot	(S11, 1, S12, 4, addr =)
	(S13, 1, S12, 1, addr =)
	(S12, 6, S14, 1, addr =)
	(S12, 8, t3, 8, addr =, level=-1)
synchronization slot	(S11,ctoken = 4,max=INF)
	(S13,ctoken = 1,max=INF)
mesg connection slot	default: running
receivers	STP3
history	execution time
reset status	–
latency	number
device	CPU/GPU/FPGA/... (ID)

cores, `C1`,`C2` and `C3`, exist in the system. `C3`'s computing capability is larger than `C1` and `C2`; the computing capability of `C1` and `C2` are equal. If the execution time of `S12` and `S14` are equal, then the extra token will be accumulated at `S14`'s buffer,as shown in Figure 5.4, once the `S14` buffer size reach/close to its maximum limitation(28 token), a `SLEEP` message will be sent from `S14` to `S12`. while `S14` keep running and consuming the stored token, a `WAKEUP` message will be sent to `S12` when `S14`'s buffer size reaches to its minimum limitation (1 token); In another case, if the execution time of `S14` are faster than `S12`, (which will be either the computing ability of computing engine mapped by `S14` is far more better than the computing engine mapped by `S12` or the computing in `S12` is more complicate than `S14` and it will take a longer time to finish, as shown in Figure 5.5), no `STOP` or `SLEEP` message will be sent if the `S14`'s buffer size is less than its maximum limitation. But the if `S14`'s buffer size is always less than the minimum limitation, as shown in Figure 5.6, a `CHANGING` message will be sent to the `STP1`'s scheduler to request change computing engine of `S12` to one with higher computing ability, which will be core `C3` in current system.

Listing 5.1 and 5.2 are the pseudocode of `STP0` and `STP3` in Figure 5.2. `STP0` is the upper most layer of the module.The scheduler run in this layer. As described in section 5.3.3, this scheduler will map its group members (modules) to clusters based on the hardware resources. `STP0` has three module components, `STP1`,`STP2`,`STP3`. Every module component in `STP0` has a subscheduler which schedule, bind its group members, *Codelet*s, to computing engines, and synchronize all its group members.

As shown in Listing 5.1 and 5.2, every module need to set up shared spaces or variables, using function `setshare`, which can be accessed by all its group members. Fully utilizing the data locality to minimize the data movement is one of important factors to make sure the stream flow smoothly running. Both functions `setcxn` and `setcrosscxn` will set up or change the data connection slot for the streaming module/*Codelet*. Function `setcxn` will be used to connect components in same layer. The connection is directed, starting from producer and end at consumer. The number on connection (production-consumption token) can be used by synchronization slot. If the produced token number less than consumed token number, the consumer has to wait. Function `setcrosscxn` in Listing 5.2 plays the similar roles, but it is used to connect two components from different layers, *i.e.*, connecting `S31` to `t1` which is the transfer port of `STP1` in `STP3`. The `level` attribute in `data synchronization slot` indicates that whether the producer and consumer related the current line/connection are located in same level or not. `level` equals to the producer layer minus to consumer layer. By default, the *level* is set to zero. Function `setsync` in List 5.2 is to set up a `synchronization slot` for the streaming module/*Codelet*. As described in section 5.3.1, `synchronization slot` only exists in consumer side. It records its producer of current connection, the number of token is needed to `fire`, the maximum and minimum number of token that buffer can hold. Function `setlatency` will set the latency of current component to make sure the pipeline can run smoothly. Table 5.1 and 5.2 show the attributes of streaming module of `STP0` and `SPT3`. Table 5.3 shows the attributes of streaming *Codelet* (`S12`).

5.4 Related Work

Lee [12, 13] proposed the concept of Synchronous data flow (SDF), where each node represents a computation task (actor) and each arc represents the communication between tasks. In a `SDF` graph, the token of every data flow node will be consumed or produced maybe specified a priory. The scheduler can be done at compile time (statically). During program execution, each actor, which has an independent instruction streams and address space, must fire repeatedly in a periodic schedule.

In stream processing, multiple optimization approaches about pipeline parallelism technique have been studies by many researches in scheduling and compiler area. Hwang*et al.* [15] proposed Pipeline net/chain which can be viewed as a two-level pipelined and dynamically reconfigurable systolic array and is constructed from interconnecting multiple Functional Pipelines (FP) through a buffered crossbar network. Software pipelining [98, 99, 105] is an efficient method to exploit the coarse-grained and fine-grained parallelism in stream programs, and it has been one of the successful instruction level parallelism (ILP) techniques.It considers the whole program as a loop and takes a periodic schedule as iteration of the loop, and successive iterations can be overlapped at run-time [106]. Single-dimension software pipelining(SSP) [17] is a resource-constrained software pipelining method for both perfect (data independent), and imperfect (data dependence) loop nests on single-core architectures. Multi-threaded software pipelining (MT-SSP) [16] based on SSP could automatically extract threads from loop nest written in a sequential language and parallel schedule these multi-threads on homogeneous multi-core with resource constraints.Decoupled software pipelining (DSWP) [103] exploits the fine-grained pipeline parallelism lurking in most applications to extract long-running, currently executing threads.

Task level parallelism can be used in the every stage of pipeline. Allocating task into apposite computing unit play a pivot role. Sridharan*et al.* [60] proposed an integer linear programming (ILP)-based parallelization approach which can automatically extract task-level parallelism and balance the extracted tasks for processing units which have different performance characteristics. Furthermore, the resource allocation problem can be formulated as a constraint satisfaction problem. To balance the load of the multiple applications equally over all heterogeneous multi-processor system-on-chip(tiles), Stuijk *et al.* [107] proposed three steps resource allocation strategy: bind the nodes of SDFG to tiles to estimate the critically execution cycles of SDFG; construct static-order schedules to fire all of the nodes bound to each tile and the communication cost (delay when tokens were sent between tiles) is taken into account; use binary search algorithms to allocate time slices for all tiles to satisfy throughput constraint. Zhu *et al.* [108] proposed an implicit retiming and unfloding approach for binding and scheduling static rate-optimal scheduling of Synchronous dataflow graphs(SDFGs) on a multiprocessor platform.

To deliver the optimal long-term throughput by exploring inter-tasks parallelism on Multiprocessor Systems-on-Chips (MPSoC), Tang *et al.* [109] use Parallelism Graph(PG) to quantify and model the task-level parallelism of the `SDFG`, and transform the mapping problem to graph partition problem. 0-1 integer linear programming(ILP) were utilized to solve small-scale graph partition problem and for the large-scale problems, two-step heuristic called greedy partition and refinement algorithm(GPRA) is proposed. However both ILP and GPRA are incapable at producing the global optimal solution, Hybrid Genetic Algorithm (HGA) which combine genetic algorithms(GAs) with parallelism enhancement were proposed. Tang *et al.* focused on the task-to-processor mapping problem which no more than one processor is allocated to each `SDFG` task in the mapping.

Thies *et al.* [110] proposed StreamIt language and corresponding compiler for streaming applications. It includes four main language features: a structured model of streams, a messaging system for control, a re-initialization mechanism, and a natural textual syntax. StreamIt assumes that independent processors communicate in regular pattern and overlooks the granularity, memory layout, network interconnect, *etc.*.

All the works mentioned above do not address or overlook data movement (communication) problem existing in current highly heterogeneous and hierarchical system. Indeed, on-chip and off-chip communication is projected to become a major bottleneck in terms of performance, energy consumption, and reliability when hundreds of heterogeneous compute engines are integrated. Moreover, because of the physical limits that core count per chip continues to increase dramatically while the available on-chip memory per core is only getting marginally bigger. In this case, the stream-based event-driven model, described in section 5, focuses on the fully utilization of data locality in fine-grain parallelism level and minimization bulk data movement cost in coarse-grain parallelism level.

Chapter 6

Conclusions and Future Work

A fine-grain event-driven execution model has been presented here with the goal of solving many of the various challenges that exist in current/future high-performance hierarchical homogeneous/heterogeneous many-core systems: exploitation of parallelism, efficient utilization of resources and system scalability to satisfy the continued growing pressure for increased processing performance requirements from industry (applications) and scientific circles (scientific computing).

Fine-grain synchronization with event-driven multithreading model, based on the *Codelet* Model, has given us large-scale parallelism exploitation of dependence-heavy applications as opposed to the coarse-grain synchronization in current high-performance general purpose many-core shared-memory compute nodes. The advantages of using finer-grained synchronization come from the fact that, even with initially " almost embarrassingly parallel" workloads such as stencil-based iterative solvers, performance can be significantly improved using regular work distribution among processing elements. However, the fine-grain synchronization work demonstrated here relies on a hand-coded approach. A compiler equipped with an *OpenMP-to-Codelet* fine-grain function will be developed in the future.

Our heterogeneity-aware iterative scheduling algorithm, `IDAWL`, has been designed to leverage load-balancing techniques to obtain the best workload partition between CPUs and general-purpose accelerators—*e.g.*, GPUs. However, naïve heuristics may result in worsened performance and power consumption, especially for the applications which feature a high-degree of data dependence and need to regularly perform host-accelerator synchronizations. `IDAWL` leverages a profile-based approach based on machine learning and online scheduling to offer a general approach to efficiently utilize, in a dynamic fashion, available heterogeneous resources.

Energy efficiency of our schemes will be the topic of future work. It will guarantee that our `IDAWL` can reach a good trade-off point between performance and power on heterogeneous architectures. Stream-based event-driven heterogeneous multithreading model has a huge potential in the streaming application domain. The features of exploiting two levels parallelism (coarse- and fine-grain level) to construct streaming pipeline stage, fully utilizing the data locality to minimize the data movement and easily adding new heterogeneous components help this model overcome the majority of issues encountering by streaming applications on heterogeneous system. Future work will focus on how to dynamic schedule workloads and overlap the computation and communication of different heterogeneous computing resources to improvement the performance. Energy efficient computing, which seeks to utilize specialized cores, accelerators (FPGAs), and graphical processing units (GPUs) to eliminate the energy overheads of general-purpose homogeneous cores, is another important topic for future work.

Bibliography

[1] TOP500 Supercomputer List. `http://www.top500.org` (visited on Nov. 2017).

[2] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[3] OpenMP Architecture Review Board. Openmp application program interface version 4, July 2013.

[4] OpenMP Architecture Review Board. Openmp application program interface version 4.5, November 2015.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[6] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 291–303. ACM, 2002.

[7] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[8] Ujval J Kapasi, Scott Rixner, William J Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[9] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[10] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, pages 362–376, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[11] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 368–373, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[12] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[13] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[15] K. Hwang and Z. Xu. Multipipeline networking for compound vector processing. *IEEE Transactions on Computers*, 37(1):33–47, Jan 1988.

[16] Alban Douillet and Guang R. Gao. Software-pipelining on multi-core architectures. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Trans. Archit. Code Optim.*, 4(1), March 2007.

[18] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA, 2014. ACM.

[19] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, New York, NY, USA, 2011. ACM.

[20] Jack B. Dennis. First version of a data flow procedure language. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*. Springer, 1974.

[21] Arvind and Kim P. Gostelow. The u-interpreter. *IEEE Computer*, 15(2), 1982.

[22] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. An implementation of the codelet model. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 633–644, Berlin, Heidelberg, 2013. Springer-Verlag.

[23] J. Arteaga, S. Zuckerman, and G. R. Gao. Multigrain parallelism: Bridging coarse-grain parallel programs and fine-grain event-driven multithreading. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 799–808, May 2017.

[24] Jaime Arteaga, Stéphane Zuckerman, and Guang R. Gao. Generating fine-grain multithreaded applications using a multigrain approach. *ACM Trans. Archit. Code Optim.*, 14(4):47:1–47:26, December 2017.

[25] OpenMP Architecture Review Board. Openmp application program interface version 3.0, May 2008.

[26] Intel. Intel 64 and ia-32 architectures software developers manual. *Volumes 1–3*, 1, December 2015.

[27] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory, July 2011.

[28] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[29] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[30] Kathleen Knobe. Ease of use with concurrent collections (cnc). *Hot Topics in Parallelism*, 2009.

[31] Zoran Budimlic, Michael G. Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David M. Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. Concurrent collections. *Scientific Programming*, 18(3-4), 2010.

[32] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard W. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010.

[33] Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. Concurrent collections programming model. In David A. Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011.

[34] Chenyang Liu and Milind Kulkarni. Optimizing the lulesh stencil code using concurrent collections. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, New York, NY, USA, 2015. ACM.

[35] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013.

[36] T. Gautier, F. Lementec, V. Faucher, and B. Raffin. X-kaapi: A multi paradigm runtime for multicore architectures. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013.

[37] Tim Mattson, R Cledat, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, B Seshasayee, R van der Wijngaart, and Vivek Sarkar. Ocr: The open community runtime interface. Technical report, Tech. Rep., June 2015.[Online]. Available: https://xstack. exascale-tech. com/git/public, 2015.

[38] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, New York, NY, USA, 2012. ACM.

[39] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, New York, NY, USA, 2006. ACM.

[40] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1), 1996.

[41] Rajkishore Barik, Zoran Budimli, Vincent Cav, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Tarlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.

[42] Zoran Budimli, Vincent Cav, Raghavan Raman, Jun Shirako, Sağnak Tarlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the habanero-java parallel programming language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, New York, NY, USA, 2011. ACM.

[43] Vincent Cav, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, New York, NY, USA, 2011. ACM.

[44] Judit Planas, Rosa M. Badia, Eduard Ayguad, and Jesus Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3), 2009.

[45] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29*

- *September 2, 2011, Proceedings, Part I*, chapter Productive Cluster Programming with OmpSs. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[46] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, New York, NY, USA, 1988. ACM.

[47] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. *SIGPLAN Not.*, 26(7), April 1991.

[48] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, 2007. ACM.

[49] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[50] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, New York, NY, USA, 2015. ACM.

[51] Sunil Shrestha, Joseph Manzano, Andres Marquez, John Feo, and Guang R. Gao. *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers*, chapter Jagged Tiling for Intra-tile Parallelism and Fine-Grain Multithreading. Springer International Publishing, Cham, 2015.

[52] Sunil Shrestha, Guang R. Gao, Joseph Manzano, Andres Marquez, and John Feo. Locality aware concurrent start for stencil applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, Washington, DC, USA, 2015. IEEE Computer Society.

[53] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, New York, NY, USA, 2011. ACM.

[54] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010.

[55] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In

Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, May 2011.

[56] Takayuki Muranushi and Junichiro Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51, 2015. International Conference On Computational Science, {ICCS} 2015Computational Science at the Gates of Nature.

[57] Michael Lesniak. Pastha: Parallelizing stencil calculations in haskell. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, New York, NY, USA, 2010. ACM.

[58] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Sdslc: A multi-target domain-specific compiler for stencil computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, New York, NY, USA, 2015. ACM.

[59] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 134–144, Washington, DC, USA, 2011. IEEE CS.

[60] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA, 2014. ACM.

[61] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. Understanding co-running behaviors on integrated cpu/gpu architectures. *IEEE TPDS*, 28(3):905–918, March 2017.

[62] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *39th Symposium on Computer Architecture*, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE CS.

[63] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *46th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 457–467, Dec 2013.

[64] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena. Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sept 2016.

[65] Q. Chen and M. Guo. Contention and locality-aware work-stealing for iterative applications in multi-socket computers. *IEEE Transactions on Computers*, PP(99):1–1, 2017.

[66] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In *IEEE International Conference on Cluster Computing*, pages 19–28, Sept 2010.

[67] B. v. Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal. Performance models for cpu-gpu data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 11–20, May 2014.

[68] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[69] NVIDIA. *CUDA C: Programming Guide, Version 10.0.*, Oct 2018.

[70] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R. Alam, Thomas C. Schulthess, and Torsten Hoefler. A pcie congestion-aware performance model for densely populated accelerator servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 63:1–63:11, Piscataway, NJ, USA, 2016. IEEE Press.

[71] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications. In *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*, 2012.

[72] Benedict R. Gaster and Lee Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 45:42–52, 2012.

[73] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[74] G. Teodoro, T. M. Kurc, T. Pan, L. A. D. Cooper, J. Kong, P. Widener, and J. H. Saltz. Accelerating large scale image analyses on parallel, cpu-gpu equipped systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1093–1104, May 2012.

[75] Sankaralingam Panneerselvam and Michael Swift. Rinnegan: Efficient resource use in heterogeneous architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 373–386, New York, NY, USA, 2016. ACM.

[76] D. Sukkari, H. Ltaief, M. Faverge, and D. Keyes. Asynchronous task-based polar decomposition on single node manycore architectures. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2017.

[77] L. Sant'Ana, D. Cordeiro, and R. Camargo. PLB-HeC: A profile-based load-balancing algorithm for heterogeneous CPU-GPU clusters. In *2015 IEEE International Conference on Cluster Computing*, pages 96–105, Sept 2015.

[78] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[79] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa. A framework for application-guided task management on heterogeneous embedded systems. *ACM Trans. Archit. Code Optim.*, 12(4):42:1–42:25, December 2015.

[80] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.

[81] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[82] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. Logp: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, November 1996.

[83] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71 – 79, 1997.

[84] Thibaut Lutz, Christian Fensch, and Murray Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59, 2013.

[85] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[86] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[87] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[88] Yuan Wen and Michael F.P. O'Boyle. Merge or separate?: Multi-job scheduling for opencl kernels on cpu/gpu platforms. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 22–31, New York, NY, USA, 2017. ACM.

[89] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis, and K. Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 151–162, Aug 2014.

[90] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[91] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The stream virtual machine. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 267–277, Washington, DC, USA, 2004. IEEE Computer Society.

[92] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating amdahl's law through epi throttling. *SIGARCH Comput. Archit. News*, 33(2):298–309, May 2005.

[93] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, May 2005.

[94] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[95] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007.

[96] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[97] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 23–32, Sept 2006.

[98] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGPLAN Not.*, 41(11):151–162, October 2006.

[99] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008.

[100] G. Tremblay. Lenient evaluation is neither strict nor lazy. *Computer Languages*, 26(1):43 – 66, 2000.

[101] Guy Tremblay. *Parallel implementation of lazy functional languages using abstract demand propagation.* PhD thesis, McGill University, 1994.

[102] Alfredo Cristobal, Andrei Tchernykh, and Wen Yen. I- structure software cache, an approach to exploit data locality in cluster computing. 2000.

[103] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12 pp.–118, Nov 2005.

[104] D. Cordes, O. Neugebauer, M. Engel, and P. Marwedel. Automatic extraction of task-level parallelism for heterogeneous mpsocs. In *2013 42nd International Conference on Parallel Processing*, pages 950–959, Oct 2013.

[105] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[106] H. Wei, J. Yu, H. Yu, M. Qin, and G. R. Gao. Software pipelining for stream programs on resource constrained multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2338–2350, Dec 2012.

[107] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 777–782, New York, NY, USA, 2007. ACM.

[108] X. Y. Zhu, M. Geilen, T. Basten, and S. Stuijk. Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):905–918, June 2016.

[109] Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk, and Ji-Bo Wei. Mapping of synchronous dataflow graphs on mpsocs based on parallelism enhancement. *Journal of Parallel and Distributed Computing*, 101:79 – 91, 2017.

[110] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.