Published in final edited form as:

Phys Med. 2019 August ; 64: 166–173. doi:10.1016/j.ejmp.2019.06.016.

Monte Carlo Processing on a Chip (MCoaC)-preliminary experiments toward the realization of optimal-hardware for TOPAS/Geant4 to drive discovery

Sachin Dev[2], **O.S Sarun**[1], **Amit Saxena**[1], **Yogindra S. Abhyankar**[1], **Rajendra Joshi**[1], **Hemant Darbari**[1], **Asheet K Nath**[1], **C Sajish**[1], **U. B. Sonavane**[1], **Vivek Gavane**[1], **Abhay Deshpande**[3], **Tanuja Dixit**[3], **Rajesh Harsh**[3], **Rajendra Badwe**[4], **Ashok Sharma**[5], **G. K. Rath**[5], **Siddhartha Laskar**[4], **Bruce Faddegon**[6], **Joseph Perl**[7], **Harald Paganetti**[8], **Jan Schuemann**[8], **Anil Srivastava**[9], **Ceferino Obcemea**[9], **Jeffrey Buchsbaum**[9,*]

[1]Centre for Development of Advanced Computing (C-DAC), Pune, India;

[2]Open Health Systems Laboratory (OHSL), USA;

[3]Society for Applied Microwave Electronics Engineering & Research (SAMEER), Mumbai, India;

[4]Tata Memorial Centre (TMC), Mumbai, India;

[5]All India Institute of Medical Sciences (AIIMS), Delhi, India;

[6]University of California San Francisco, USA;

[7]SLAC National Accelerator Laboratory, Menlo Park, USA;

[8]Massachusetts General Hospital and Harvard Medical School, Boston, USA;

[9]National Cancer Institute (NCI), Bethesda, USA.

Abstract

The TOPAS (Tool for Particle Simulation)[2] is amongst the scientific frameworks powered by the Monte Carlo (MC) toolkit Geant4[1]. TOPAS focuses on providing ease of use, and has significant implementation in the radiation oncology space at present. TOPAS functionality extends across the full capacity of Geant4, is freely available to non-profit users, and is being extended into radiobiology via the TOPAS-nBio project[3]. A current “grand problem” in cancer therapy is to convert the dose of treatment from physical dose to biological dose, optimized ultimately to the individual context of administration of treatment. Biology MC calculations are some of the most complex and require significant computational resources. In order to enhance TOPAS’s ability to become a critical tool to explore the definition and application of biological dose in radiation therapy, we chose to explore the use of Field Programmable Gate Array (FPGA) chips to speed up the Geant4 calculations at the heart of TOPAS, because this approach called “Reconfigurable Computing” (RC), has proven able to produce significant (around 90x) speed increases in

*To whom correspondence should be addressed: Jeffrey Buchsbaum, jeff.buchsbaum@nih.gov, Telephone: +1-240-276-5690, Fax No. : +1-240-276-5827.

Conflicts of Interest:
The authors have none to report.

scientific computing. Here, we describe initial steps to port Geant4 and TOPAS to be used on FPGA.

We provide performance analysis of the current TOPAS/Geant4 code from an RC implementation perspective. Baseline benchmarks are presented. Achievable performance figures of the subsections of the code on optimal hardware are presented. Aspects of practical implementation of "Monte Carlo on a chip" are also discussed.

Introduction

This technical report presents a novel, global project's initiation and initial steps, to use new computer hardware technology and novel thinking to increase the speed of computations critical for biological dose calculations in radiation oncology. If successful, the project goal is to enhance the field, by fusing three broad scientific thoughts to potentially create a new, critical capacity crossing into multiple emerging areas to ultimately help enable precision, adaptive, biologic treatment planning. The long term goal of this project is to create tools to help implement the capacity to biologically and contextually calculate the dose of all forms of radiation therapy including "blends" of radiation, the immunological status of the patient, the genetics and epigenetics of the patient's tumor and normal tissue, the biologic response to therapy by the patient to that point in treatment, and to be granular enough to address the complex spatial distribution of normal tissue and tumor tissue. Clearly, this is not the only tool that will be needed to enable this goal but the coalescence of expertise, need, and technology has made its deployment newly possible. Biologic treatment planning is a grand problem of radiation oncology and is driven by the need to properly employ radiation oncology to best treat our patients [4]. Tools to address this problem have broad, worldwide applicability.

The second critical component of this project is the hardware and expertise to implement what has been described as the "reconfigurable supercomputer" via the use of a class of processing unit called the Field Programmable Gate Array (FPGA)[5, 6]. The key concept in the use of the FPGA is to achieve speed via converting the series of general CPU instructions, each taking time, into an efficient equivalent-circuit in FPGA (Figure 1) performing required operations in parallel and thereby saving time. The strength of this approach is the generation of an optimized equivalent circuit; however, this approach lacks the inherent flexibility of the generic CPU to adapt to any code. At different times, multiple circuits can be put on FPGA making it reconfigurable, although at any point in time, it has essentially only one configuration. FPGAs are established as one of the best alternatives for solving complex scientific and engineering problems that are energy efficient compared to other available compute accelerators. This is mainly due to the fact that the FPGAs have inherently parallel hardware structure and are customizable as required. Use of this technique achieved a 90-fold improvement in search speed for the open source FASTA [7, 8, 9] search software. Multiple FPGAs, often on PCIe cards, can be employed at the same time within one system to further enhance this capacity. Beyond radiation oncology, the need for many areas of big science to achieve increased speed makes this approach of universal high impact. The focus on Geant4 [1] specifically will significantly impact the global physics community in this dimension, as many projects in addition to the medical physics community, for example in particle physics (such as the large High Energy Physics projects at CERN), particle-astrophysics (such as the Fermi Space Telescope) and materials scientists (such as studies of radiation effects in electronics), also use Geant4. Thus, the value of MC speedup being implemented in this project is significantly broader than that previously attempted [10].

The third aspect of this project convergence is the transformational ease of using TOPAS in the context of traditional MC programming and package use. It is not an understatement to say that to achieve expertise in Geant4 is something that can take a minimum of years. TOPAS provides the end user this power in only a few hours. This allows biologists and physicians to use MC, in addition to physicists. To achieve the goal of biological treatment planning, embracing more people with this tool to assist in research will be critical. In this paper we describe work that is focused on optimizing TOPAS[2] using new approaches for hardware and programming. The biological extension of TOPAS, the TOPAS-nBio [3], will also benefit directly from this project given the high calculation costs it demands and for the purpose of this paper should be considered part of TOPAS overall. Fast, easy to use, and biologically focused MC is the desired goal of this project.

Materials and Methods

Computer:

The computer environment utilized for this work was a multicore Xeon workstation, running the Centos version 7 LINUX operating system software with standard programming tools.

FPGA programming tools:

Xilinx Vivado Design Suite version 2017.3 was used for generating the FPGA design.

FPGA Card:

C-DAC developed accelerator card with Xilinx FPGA and Xilinx Alveo U200 was used.

Software Tools:

TOPAS version 3.1.3 including Geant4 version 10.3.1 was used. Profiler tools Open| SpeedShop v2.3.1, Perf v3.10, Valgrind v3.13, kcachegrind v0.7.2 and IgProf v5.9.16 were used for profiling.

Design Flow:

FPGAs are semiconductor devices that can be reprogrammed to desired application or functionality requirements after manufacturing. They implement an actual circuit, corresponding to the desired functions. The design flow for porting applications on FPGA accelerators is different than that of a standard software-design-flow. The FPGA design flow consists of various steps such as application profiling, converting algorithm into Hardware Description Language (HDL) design or customization of algorithm to FPGA architecture, functional simulation of the obtained design and implementing the design into FPGA. To extract maximum performance, benchmarking and optimization of the design is performed. It is very difficult to put the whole application on the FPGA; only the compute intensive portion of the application is run on the FPGA and the rest of the application is run on the CPU. Application profiling is used to identify the most compute intense portions of the application. After the identification of the compute intense portions of the application, the same is converted to FPGA design. There are two approaches for doing this. The first one is traditional, manually converting the portion to HDL design. The second and more recent approach is to directly convert the high level C or OpenCL code to HDL using high level synthesis tools. With this approach, the user is not required to have hardware programming knowledge. After the conversion, functional simulation is performed to verify the functionality of the design. Further, the design is synthesized and converted to a programming file for the FPGA. After porting of the compute intense portion of the application onto the FPGA, optimization and benchmarking is done to extract further performance from the FPGA.

Use case:

SAMEER, India, has developed low energy oncology system, named Siddhartha, capable of delivering 6 MeV energy photons with flattened dose of 240 cGy/min at 1m distance [11]. The radiation field generated by this machine is square in shape due to symmetric movement of X-Y jaws and has a maximum field size of 35×35 cm^2. To test applicability of Geant4/TOPAS on the FPGA card, a Geant4 code which describes the basic geometry of SAMEER 6 MeV electron linear accelerator as shown in Figure 2 was developed. A pencil beam of 6 MeV energy was taken as input to study the bremsstrahlung pattern generated after impinging on a high Z target like tungsten or tantalum. The photon output is collimated in the forward direction using primary and secondary collimators. The dose profiles were obtained in a water phantom of dimensions 50 cm × 50 cm with a voxel size of 1 cm^3 for 20 million histories. The dose profile was also compared with the experimentally obtained data to verify the Geant4/TOPAS code. This code was profiled to identify the compute intense functions in Geant4.

Results

Although the main goal is to improve the computational efficiency of TOPAS, the work so far has focused on developments affecting the underlying Geant4 code. A Geant4 MC simulation toolkit-based application described as the use-case above was profiled using multiple open-source profilers such as Open|SpeedShop, pref, Valgrind and IgProf. Multiple profilers were used to validate the profiled output. For profiling the application, the application source code as well as the Geant4 toolkit were compiled using the –pg and –g options. This helps the profiler in creating a function call-list and file references by extracting information from the application compiled using the above options.

The function “G4PhysicsVector::Value” that takes most of the compute time as depicted by the profiled data (Figure 3), was analyzed and assessed for portability to FPGA. The same function surfaced as the most compute function when the application was profiled using all three profiler tools, confirming the selection of the function. The “G4PhysicsVector::Value” function as shown in Figure 4, calls an inline function “Interpolation”, this in-turn, calls another inline function “SplineInterpolation”. The Inline functions do not appear in our profile call-list due to the fact that for the inline functions the compiler at compile time places a copy of the code of that function at each point where the function is called. The call graph for the G4PhysicsVector (Figure 5) indicates the path that the application code follows, while calling the SplineInterpolation function.

The Spline-interpolation as shown in Figure 6, is based on a piecewise polynomial called cubic spline, known to reduce interpolation errors. This is a widely used numerical method for interpolation in the scientific domain, compared to other interpolation methods. In order to measure the software execution time of this function independently, we isolated the function code from the Geant4 toolkit. When executed independently, each call to the spline-interpolation function took around 23 ns. Since the cubic spline interpolation is a piecewise polynomial equation, we expect the number of calls to this particular function to be extremely high, when the full application is executed.

We implemented the spline interpolation function on the FPGA. A simplified block diagram of the spline interpolation function is shown in Figure 7.

The SplineInterpolation function was coded in System Verilog, a hardware description language to generate hardware. A snapshot of this code is shown in Figure 8. The System Verilog code was simulated (figure 9) using the Isim simulator that is part of the Xilinx Vivado design suite and the outputs verified against our SplineInterpolation software code. After the simulation, the function was synthesized and implemented into the FPGA using the Xilinx Vivado design suite (Figure 10). The resource utilization for a single instance of the hardware implementation is shown in table 1. This preliminary, non-pipelined implementation performs one spline interpolation in 280 ns when clocked at 125 MHz. By pipelining this design, we can effectively perform spline interpolation in 10 ns, clocked at 100 MHz. Based on this calculation, a pipelined design is around 2x faster than the spline interpolation software. Since we can fit multiple such spline interpolation hardware blocks onto the FPGA, each working in parallel, we expect further acceleration for spline interpolation calculation.

To validate the energy efficiency of the FPGAs, the power consumed by the spline interpolation hardware-block was measured using the Xilinx tool. A single block consumed around 2.7 Was shown in Table 1. If we populate the whole FPGA, it will consume around 30W as indicated by the Xilinx power estimator. This value is considerably less than the power consumed by general processors or other accelerators.

Discussion

The analysis of has shown Geant4 to be a highly optimized code, where no single piece of code occupies over 10% of the total time spent by the CPU. This was not surprising to our group considering the extensive history of the work done to Geant4 for optimizing it across traditional hardware. Despite this fact, our initial work of porting a single piece of Geant4 code on FPGA has shown encouraging results. Using the FPGAs full potential to handle parallel code we expect to see significant reduction in execution time. The results shown here are preliminary and do not reflect full FPGA utilization nor do they reflect possible code optimization of Geant4 for FPGA implementation. For example, taking the top 3 to 10 functions or models used for clinical radiation oncology and combining them into one FPGA "circuit" may allow enormous speed increases; however this would require some clever reorganization of the Geant4 code. It may be possible using the current 16 nm and upcoming 7 nm FPGA products to make very large portions of Geant4 fit in one FPGA. Finally, we show that power usage is relatively low for the FPGA solution employed. The high-level synthesis tools, that directly convert high level C or OpenCL code to HDL, look very promising. We plan to use these high-level tools for targeting more functions of Geant4, as this approach will reduce the development time, enabling evaluation of a number of functions in a short span of time.

Conclusions

FPGAs represent a possible way to dramatically increase the speed of MC calculations. The power consumption advantage of using FPGA is clearly evident from our results. Our preliminary work shows that the MC code in Geant4 can be ported to an FPGA. To get considerable FPGA speed up, further work is needed to find Geant4 functions that take a large chunk of CPU time i.e. real computational bottlenecks. It may be required to look deeper into some of the Geant4 functions and may require formatting Geant4 in a novel fashion to suit FPGAs. Given the central importance of treatment planning in radiation oncology and the need to calculate biological dose, FPGA use for MC calculations will have an enormous impact on the field, if successful. As a bonus, FPGA acceleration offers new computational efficiency to all users of the Geant4 toolkit, from radiobiologists to particle physicists, particle-astrophysicists and materials scientists.

ACKNOWLEDGMENTS:

We wish to thank Xilinx (http://www.xilinx.com) for providing an FPGA card to the C-DAC (http://www.cdac.in) team in Pune, India for benchmarking and code analysis usage.

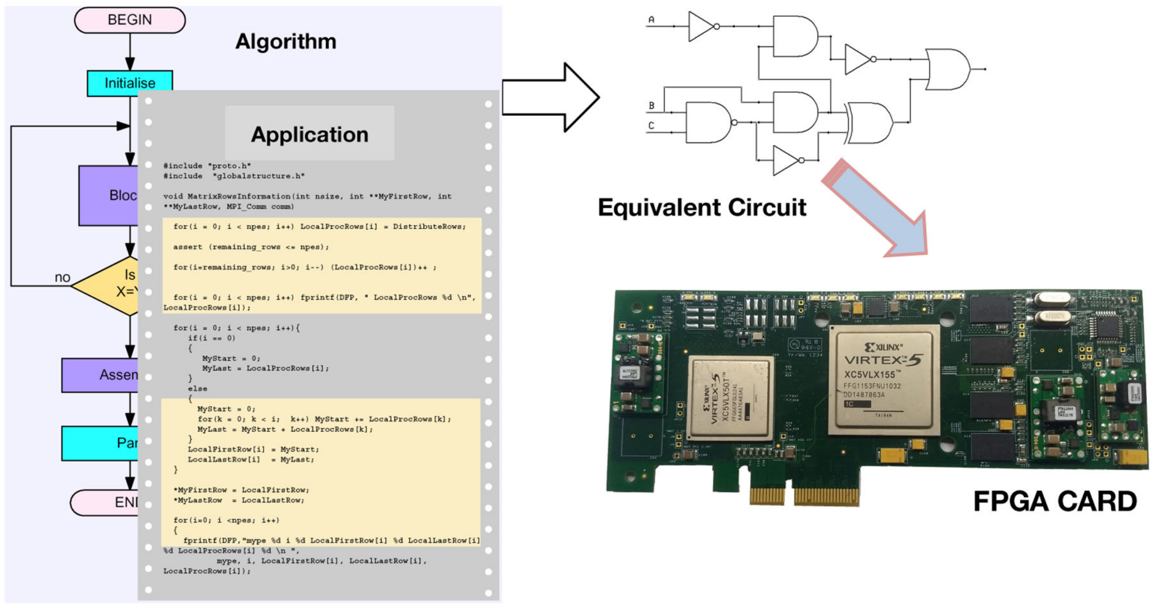
References

1. Agostinelli S, et al., Geant4-a simulation toolkit. Nuclear Instruments & Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment, 2003 506(3): p. 250–303.
2. Perl J, et al., TOPAS: an innovative proton Monte Carlo platform for research and clinical applications. Med Phys, 2012 39(11): p. 6818–37. [PubMed: 23127075]
3. Ramos-Mendez J, et al., Monte Carlo simulation of chemistry following radiolysis with TOPAS-nBio. Phys Med Biol, 2018 63(10): p. 105014. [PubMed: 29697057]
4. Ahmed MM, et al., Workshop Report for Cancer Research: Defining the Shades of Gy: Utilizing the Biological Consequences of Radiotherapy in the Development of New Treatment Approaches-Meeting Viewpoint. Cancer Res, 2018 78(9): p. 2166–2170. [PubMed: 29686020]
5. Abhyankar Y Reconfigurable Computing (RC). Available from: https://www.cdac.in/index.aspx?id=hpc_cc_recounfigurable_computing.`
6. Field-programmable gate array. Wikipedia 2018; 2018:[Available from: https://en.wikipedia.org/wiki/Field-programmable_gate_array#References.
7. Pearson WR, Rapid and sensitive sequence comparison with FASTP and FASTA. Methods Enzymol, 1990 183: p. 63–98. [PubMed: 2156132]
8. Lipman DJ and Pearson WR, Rapid and sensitive protein similarity searches. Science, 1985 227(4693): p. 1435–41. [PubMed: 2983426]

9. Sajish Chandrababu, Abhyankar Yogindra, and Joshi Rajendra, Sequence Similarity Search on Reconfigurable Computing System. International Journal of Computer and Electrical Engineering, 2012 4(5) : p. 771–774.

10. Fanti V, et al., Dose Calculation for Radiotherapy Treatment Planning Using Monte Carlo Methods on FPGA Based Hardware. 2009 16th Ieee-Npss Real Time Conference, 2009: p. 415–419.

11. Krishnan R, Deshpande AP, Dixit TS,et al., *"*S band linac tube developmental work in SAMEER*"*, Proceedings Particle Accelerator Conference, FR5REP083, PAC'09 Vancouver, BC, Canada, p. 4969–4971



```
#include "proto.h"
#include  "globalstructure.h"

void MatrixRowsInformation(int nsize, int **MyFirstRow, int
**MyLastRow, MPI_Comm comm)

  for(i = 0; i < npes; i++) LocalProcRows[i] = DistributeRows;

  assert (remaining_rows <= npes);

  for(i=remaining_rows; i>0; i--) (LocalProcRows[i])++ ;


  for(i = 0; i < npes; i++) fprintf(DFP, " LocalProcRows %d \n",
LocalProcRows[i]);

  for(i = 0; i < npes; i++){
     if(i == 0)
     {
        MyStart = 0;
        MyLast = LocalProcRows[i];
     }
     else
     {
       MyStart = 0;
       for(k = 0; k < i;  k++) MyStart += LocalProcRows[k];
       MyLast = MyStart + LocalProcRows[k];
     }
     LocalFirstRow[i] = MyStart;
     LocalLastRow[i]  = MyLast;
  }

  *MyFirstRow = LocalFirstRow;
  *MyLastRow  = LocalLastRow;

  for(i=0; i <npes; i++)
  {
    fprintf(DFP,"mype %d i %d LocalFirstRow[i] %d LocalLastRow[i]
%d LocalProcRows[i] %d \n ",
           mype, i, LocalFirstRow[i], LocalLastRow[i],
LocalProcRows[i]);
```

Figure 1. Algorithm to equivalent circuit.

The algorithm is manually converted to equivalent circuit to extract performance

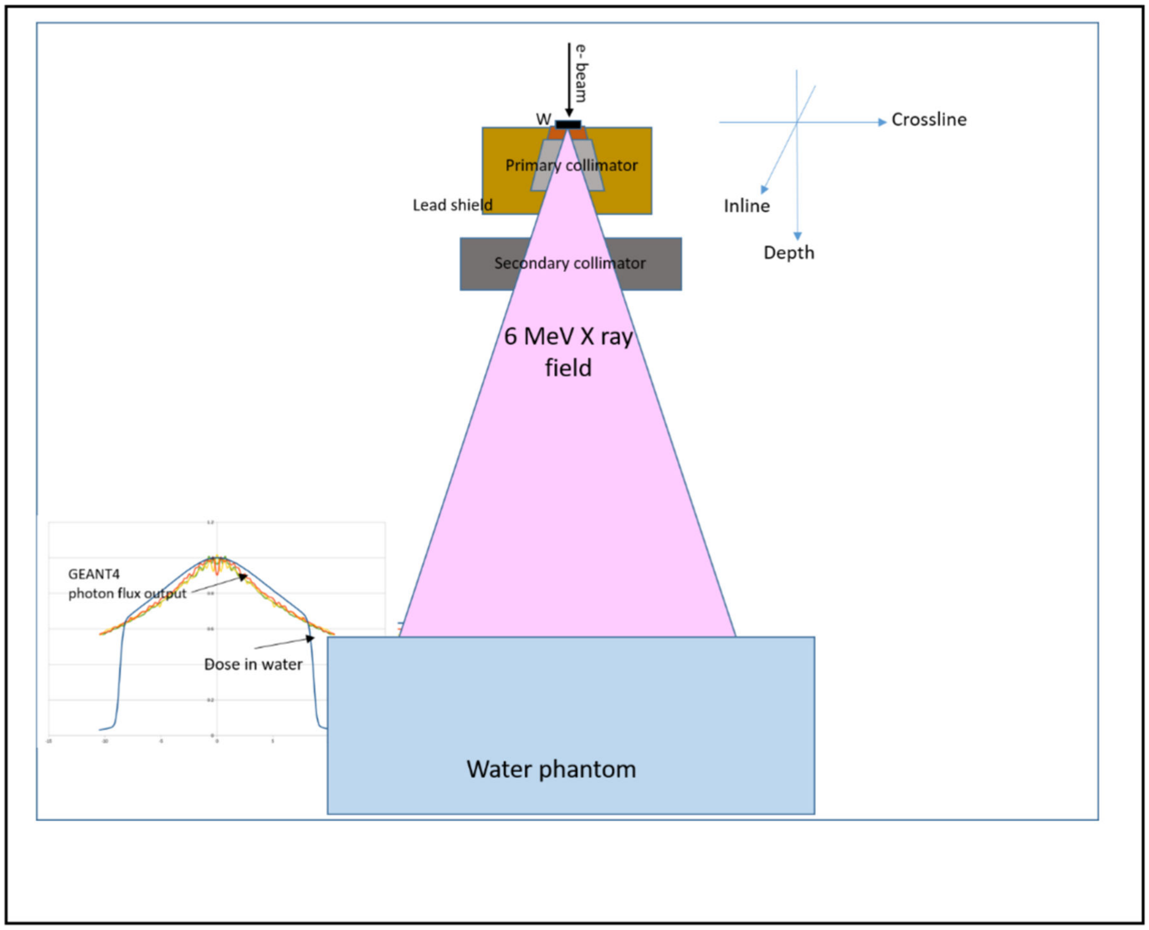
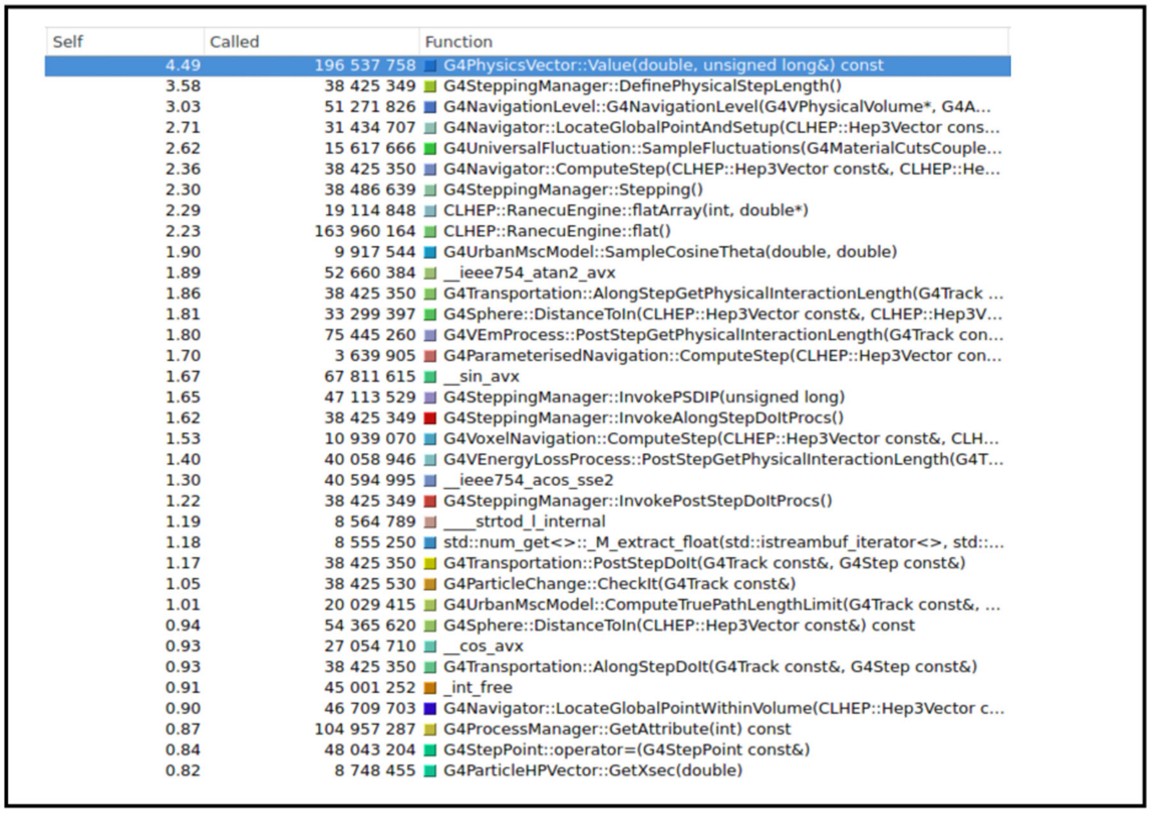


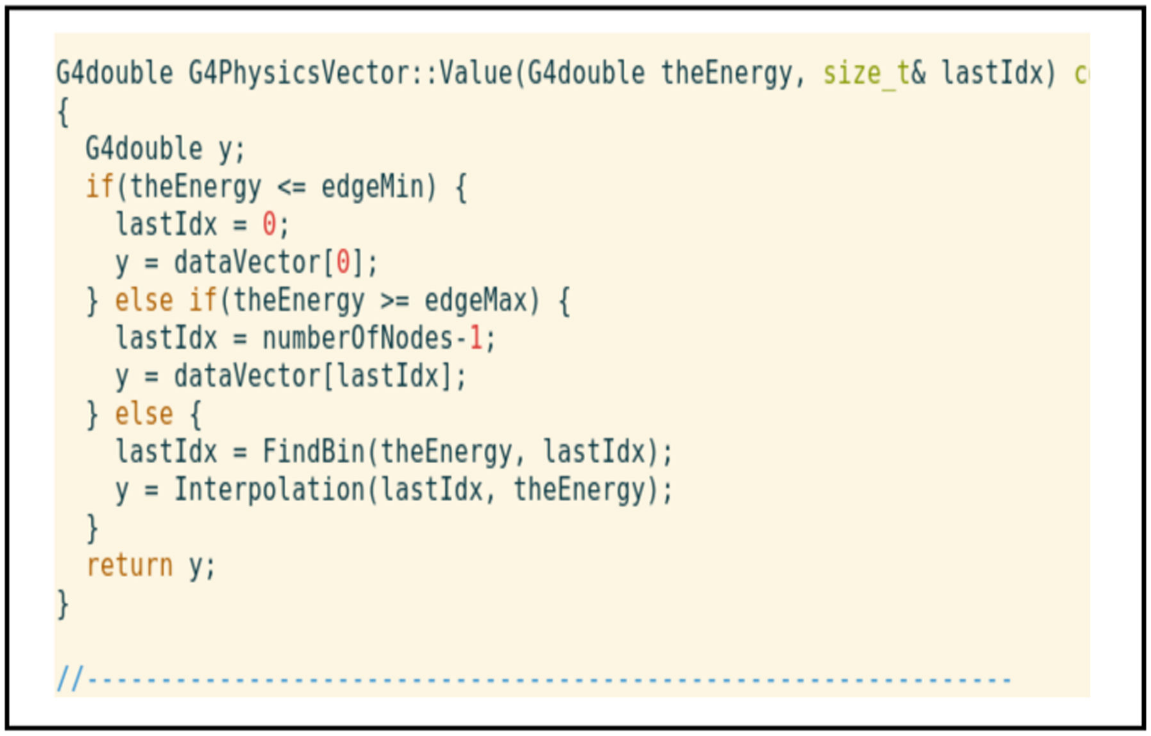
Figure 2. Geant4 Use case.
Shows setup of the use case developed using Geant4 toolkit



Self	Called	Function
4.49	196 537 758	G4PhysicsVector::Value(double, unsigned long&) const
3.58	38 425 349	G4SteppingManager::DefinePhysicalStepLength()
3.03	51 271 826	G4NavigationLevel::G4NavigationLevel(G4VPhysicalVolume*, G4A...
2.71	31 434 707	G4Navigator::LocateGlobalPointAndSetup(CLHEP::Hep3Vector cons...
2.62	15 617 666	G4UniversalFluctuation::SampleFluctuations(G4MaterialCutsCouple...
2.36	38 425 350	G4Navigator::ComputeStep(CLHEP::Hep3Vector const&, CLHEP::He...
2.30	38 486 639	G4SteppingManager::Stepping()
2.29	19 114 848	CLHEP::RanecuEngine::flatArray(int, double*)
2.23	163 960 164	CLHEP::RanecuEngine::flat()
1.90	9 917 544	G4UrbanMscModel::SampleCosineTheta(double, double)
1.89	52 660 384	__ieee754_atan2_avx
1.86	38 425 350	G4Transportation::AlongStepGetPhysicalInteractionLength(G4Track ...
1.81	33 299 397	G4Sphere::DistanceToIn(CLHEP::Hep3Vector const&, CLHEP::Hep3V...
1.80	75 445 260	G4VEmProcess::PostStepGetPhysicalInteractionLength(G4Track con...
1.70	3 639 905	G4ParameterisedNavigation::ComputeStep(CLHEP::Hep3Vector con...
1.67	67 811 615	__sin_avx
1.65	47 113 529	G4SteppingManager::InvokePSDIP(unsigned long)
1.62	38 425 349	G4SteppingManager::InvokeAlongStepDoItProcs()
1.53	10 939 070	G4VoxelNavigation::ComputeStep(CLHEP::Hep3Vector const&, CLH...
1.40	40 058 946	G4VEnergyLossProcess::PostStepGetPhysicalInteractionLength(G4T...
1.30	40 594 995	__ieee754_acos_sse2
1.22	38 425 349	G4SteppingManager::InvokePostStepDoItProcs()
1.19	8 564 789	____strtod_l_internal
1.18	8 555 250	std::num_get<>::_M_extract_float(std::istreambuf_iterator<>, std::...
1.17	38 425 350	G4Transportation::PostStepDoIt(G4Track const&, G4Step const&)
1.05	38 425 530	G4ParticleChange::CheckIt(G4Track const&)
1.01	20 029 415	G4UrbanMscModel::ComputeTruePathLengthLimit(G4Track const&, ...
0.94	54 365 620	G4Sphere::DistanceToIn(CLHEP::Hep3Vector const&) const
0.93	27 054 710	__cos_avx
0.93	38 425 350	G4Transportation::AlongStepDoIt(G4Track const&, G4Step const&)
0.91	45 001 252	_int_free
0.90	46 709 703	G4Navigator::LocateGlobalPointWithinVolume(CLHEP::Hep3Vector c...
0.87	104 957 287	G4ProcessManager::GetAttribute(int) const
0.84	48 043 204	G4StepPoint::operator=(G4StepPoint const&)
0.82	8 748 455	G4ParticleHPVector::GetXsec(double)

Figure 3. Valgrind profile.

Shows the profiles of Geant4 based code generated by Valgrind profiler. The function “G4PhysicsVector::Value” appears at the top of the profile list.



```
G4double G4PhysicsVector::Value(G4double theEnergy, size_t& lastIdx) c
{
  G4double y;
  if(theEnergy <= edgeMin) {
    lastIdx = 0;
    y = dataVector[0];
  } else if(theEnergy >= edgeMax) {
    lastIdx = numberOfNodes-1;
    y = dataVector[lastIdx];
  } else {
    lastIdx = FindBin(theEnergy, lastIdx);
    y = Interpolation(lastIdx, theEnergy);
  }
  return y;
}

//-----------------------------------------------------------------
```

Figure 4. G4PhysicsVector.cc source snippet.

G4PhysicsVector::Value is defined here and the function in-turn call a function interpolation.

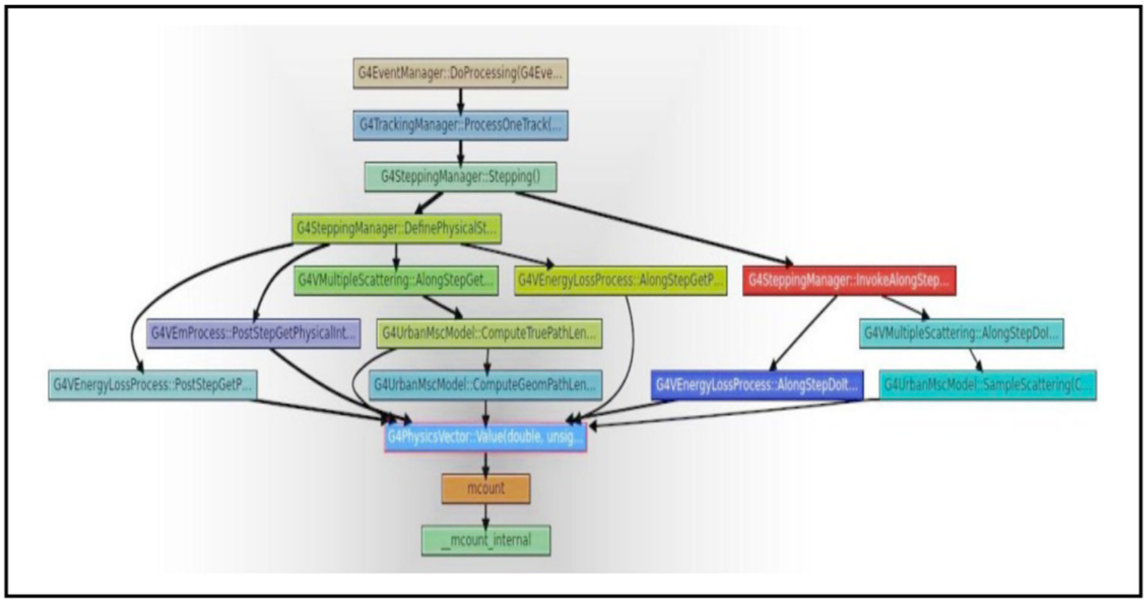
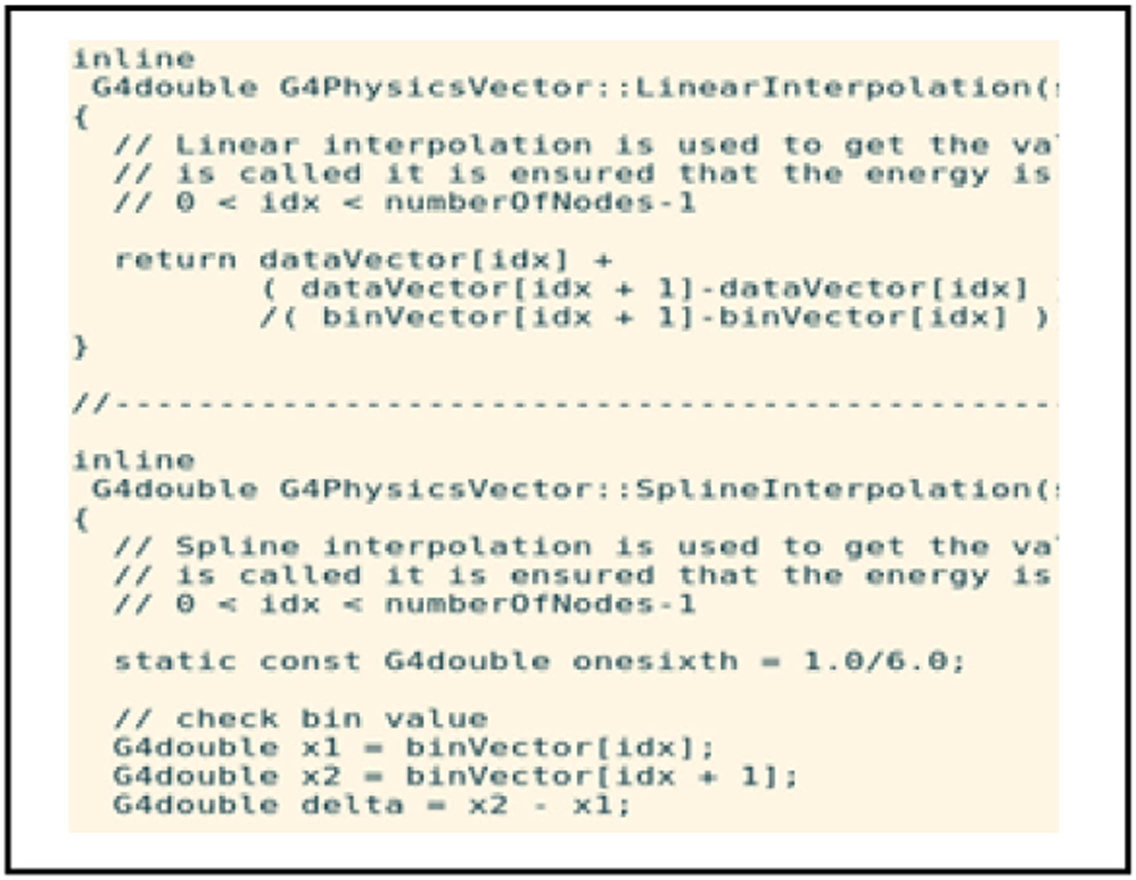

G4EventManager::DoProcessing(G4Eve...
G4TrackingManager::ProcessOneTrack(...
G4SteppingManager::Stepping()
G4SteppingManager::DefinePhysicalSt...
G4VMultipleScattering::AlongStepGet...
G4VEnergyLossProcess::AlongStepGetP...
G4SteppingManager::InvokeAlongStep...
G4VEmProcess::PostStepGetPhysicalInt...
G4UrbanMscModel::ComputeTruePathLen...
G4VMultipleScattering::AlongStepDoI...
G4VEnergyLossProcess::PostStepGetP...
G4UrbanMscModel::ComputeGeomPathLen...
G4VEnergyLossProcess::AlongStepDoIt...
G4UrbanMscModel::SampleScattering(C...
G4PhysicsVector::Value(double, unsig...
mcount
__mcount_internal

Figure 5. Call graph for G4PhysicsVector.

The call graph for the G4PhysicsVector indicates the path that the code follows.



```
inline
 G4double G4PhysicsVector::LinearInterpolation(
{
  // Linear interpolation is used to get the va
  // is called it is ensured that the energy is
  // 0 < idx < numberOfNodes-1

  return dataVector[idx] +
         ( dataVector[idx + 1]-dataVector[idx]
         /( binVector[idx + 1]-binVector[idx] )
}

//--------------------------------------------

inline
 G4double G4PhysicsVector::SplineInterpolation(
{
  // Spline interpolation is used to get the va
  // is called it is ensured that the energy is
  // 0 < idx < numberOfNodes-1

  static const G4double onesixth = 1.0/6.0;

  // check bin value
  G4double x1 = binVector[idx];
  G4double x2 = binVector[idx + 1];
  G4double delta = x2 - x1;
```

Figure 6. Spline Interpolation code snippet.

The spline interpolation is an inline function called by G4PhysicsVector::value and this doesn’t show up in our profile list.

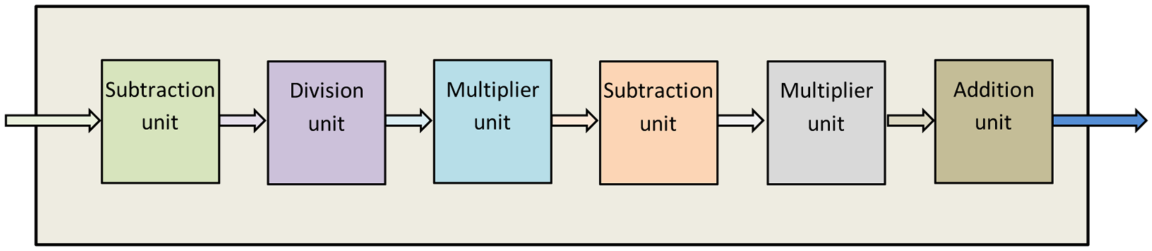
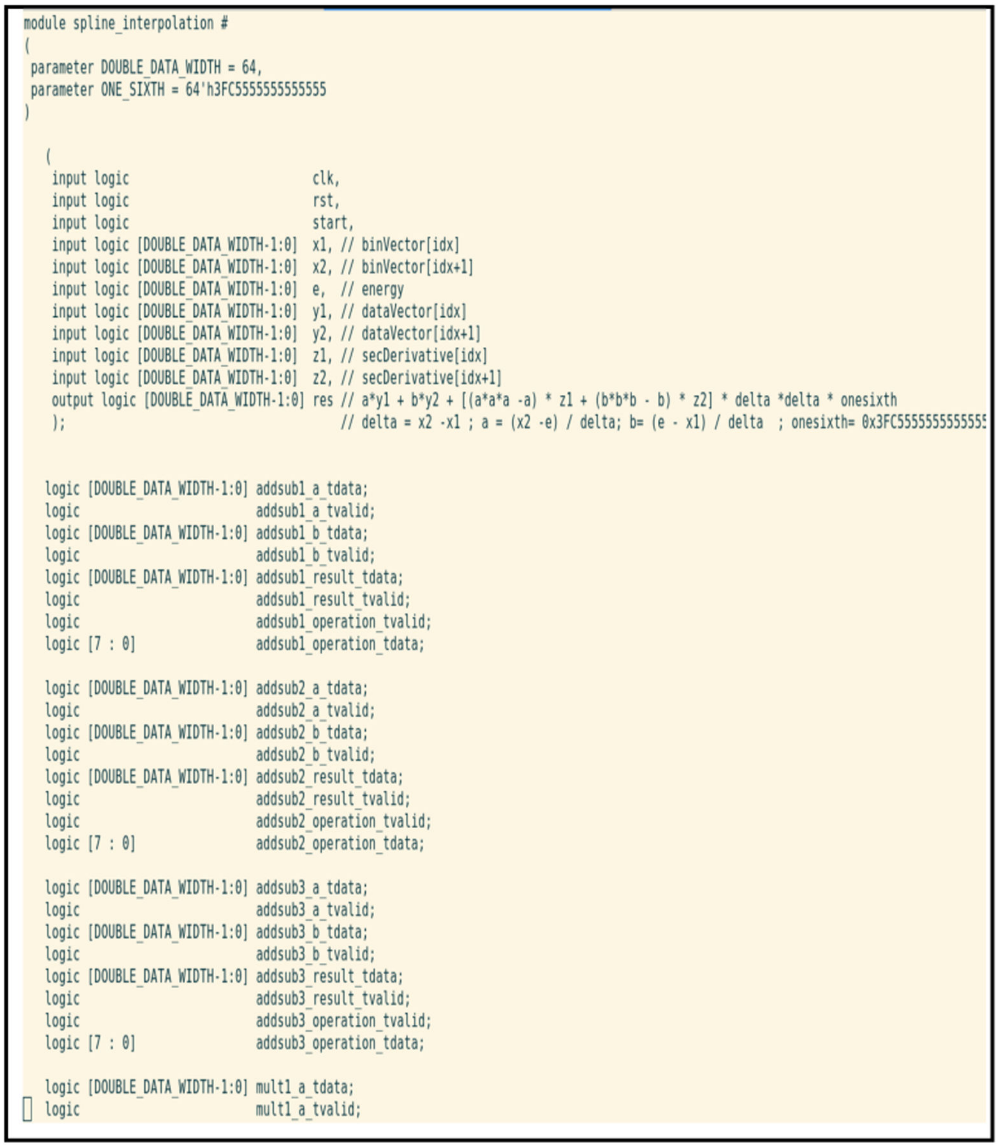


Figure 7. Simplified block diagram of spline interpolation.

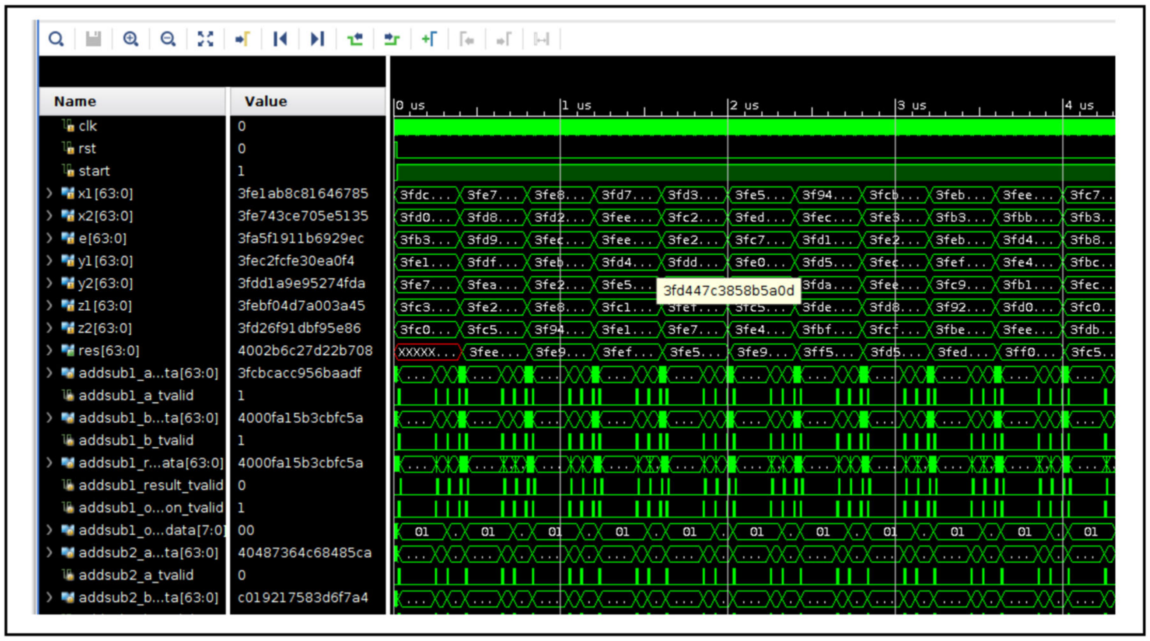
The simplified block diagram of spline interpolation calculation



```
module spline_interpolation #
(
 parameter DOUBLE_DATA_WIDTH = 64,
 parameter ONE_SIXTH = 64'h3FC5555555555555
)

   (
    input logic                         clk,
    input logic                         rst,
    input logic                         start,
    input logic [DOUBLE_DATA_WIDTH-1:0]  x1, // binVector[idx]
    input logic [DOUBLE_DATA_WIDTH-1:0]  x2, // binVector[idx+1]
    input logic [DOUBLE_DATA_WIDTH-1:0]  e,  // energy
    input logic [DOUBLE_DATA_WIDTH-1:0]  y1, // dataVector[idx]
    input logic [DOUBLE_DATA_WIDTH-1:0]  y2, // dataVector[idx+1]
    input logic [DOUBLE_DATA_WIDTH-1:0]  z1, // secDerivative[idx]
    input logic [DOUBLE_DATA_WIDTH-1:0]  z2, // secDerivative[idx+1]
    output logic [DOUBLE_DATA_WIDTH-1:0] res // a*y1 + b*y2 + [(a*a*a -a) * z1 + (b*b*b - b) * z2] * delta *delta * onesixth
    );                                       // delta = x2 -x1 ; a = (x2 -e) / delta; b= (e - x1) / delta  ; onesixth= 0x3FC555555555555


   logic [DOUBLE_DATA_WIDTH-1:0] addsub1_a_tdata;
   logic                         addsub1_a_tvalid;
   logic [DOUBLE_DATA_WIDTH-1:0] addsub1_b_tdata;
   logic                         addsub1_b_tvalid;
   logic [DOUBLE_DATA_WIDTH-1:0] addsub1_result_tdata;
   logic                         addsub1_result_tvalid;
   logic                         addsub1_operation_tvalid;
   logic [7 : 0]                 addsub1_operation_tdata;

   logic [DOUBLE_DATA_WIDTH-1:0] addsub2_a_tdata;
   logic                         addsub2_a_tvalid;
   logic [DOUBLE_DATA_WIDTH-1:0] addsub2_b_tdata;
   logic                         addsub2_b_tvalid;
   logic [DOUBLE_DATA_WIDTH-1:0] addsub2_result_tdata;
   logic                         addsub2_result_tvalid;
   logic                         addsub2_operation_tvalid;
   logic [7 : 0]                 addsub2_operation_tdata;

   logic [DOUBLE_DATA_WIDTH-1:0] addsub3_a_tdata;
   logic                         addsub3_a_tvalid;
   logic [DOUBLE_DATA_WIDTH-1:0] addsub3_b_tdata;
   logic                         addsub3_b_tvalid;
   logic [DOUBLE_DATA_WIDTH-1:0] addsub3_result_tdata;
   logic                         addsub3_result_tvalid;
   logic                         addsub3_operation_tvalid;
   logic [7 : 0]                 addsub3_operation_tdata;

   logic [DOUBLE_DATA_WIDTH-1:0] mult1_a_tdata;
   logic                         mult1_a_tvalid;
```

Figure 8. System Verilog code.

System Verilog code for spline interpolation to generate the necessary hardware.



Name	Value
clk	0
rst	0
start	1
x1[63:0]	3fe1ab8c81646785
x2[63:0]	3fe743ce705e5135
e[63:0]	3fa5f1911b6929ec
y1[63:0]	3fec2fcfe30ea0f4
y2[63:0]	3fdd1a9e95274fda
z1[63:0]	3febf04d7a003a45
z2[63:0]	3fd26f91dbf95e86
res[63:0]	4002b6c27d22b708
addsub1_a...ta[63:0]	3fcbcacc956baadf
addsub1_a_tvalid	1
addsub1_b...ta[63:0]	4000fa15b3cbfc5a
addsub1_b_tvalid	1
addsub1_r...ata[63:0]	4000fa15b3cbfc5a
addsub1_result_tvalid	0
addsub1_o...on_tvalid	1
addsub1_o...data[7:0]	00
addsub2_a...ta[63:0]	40487364c68485ca
addsub2_a_tvalid	0
addsub2_b...ta[63:0]	c019217583d6f7a4

Figure 9. Simulation output:

Simulation waveform of spline interpolation system verilog code

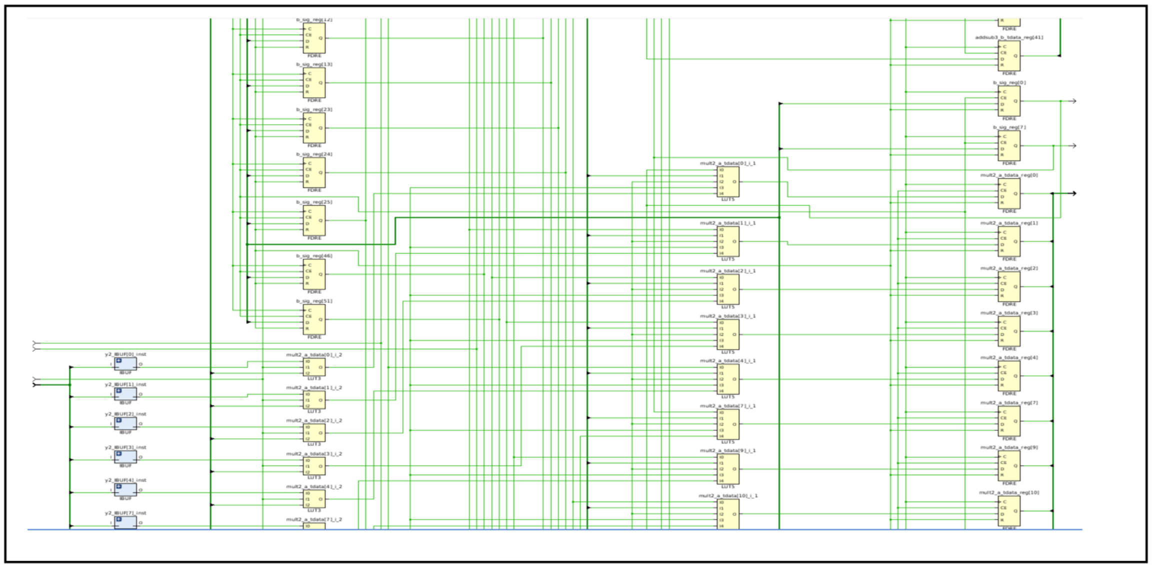


Figure 10. Synthesized netlist:

Synthesis output of spline interpolation system verilog code.

Table 1.

Resource and Power utilization of single instance of spline interpolation function.

Resource Utilization (Xilinx UltraScale+)	
CLB LUT	0.82%
CLB Registers	0.16%
DSP’s	0.37%
Power Utilization	
Total Power	2.76 W
Dynamic Power	0.29 W
Static Power	2.47 W