2017

Accelerating The Discontinuous Galerkin Cell-Vertex Scheme (Dg-Cvs) Solver On Cpu-Gpu Heterogeneous Systems

Xiaoqi Hu
University of Mississippi

ACCELERATING THE DISCONTINUOUS GALERKIN CELL-VERTEXT SCHEME (DG-CVS) SOLVER ON CPU-GPU HETEROGENEOUS SYSTEMS

A Dissertation
presented in partial fulfillment of requirements
for the degree of Master of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Xiaoqi Hu

May 2017

ABSTRACT

DG-CVS (Discontinuous Galerkin Cell-Vertex Scheme) is an efficient, accurate and robust numerical solver for general hyperbolic conservation laws. It can solve a broad range of conservation laws such as the shallow water equation and magnetohydrodynamics equations. DG-CVS is a Riemann-solver-free high order space-time method for arbitrary space conservation laws. It fuses the Discontinuous Galerkin (DG) method and the Conservation Element/Solution Element (CE/SE) method to take advantage of the best features of both methods. Thanks to the CE/SE method, the time derivative of the solution is treated as an independent unknown, which is amendable to GPU's parallel execution.

In this thesis, we use a CPU-GPU heterogeneous processor to accelerate DG-CVS to demonstrate that complex scientific applications can benefit from a heterogeneous computing system. There are challenges that such scientific program poses on the GPU architecture such as thread divergence and low kernel occupancy. We developed optimizations to address these concerns. Our proposed optimizations include thread remapping to minimize thread divergence and register pressure reduction to increase kernel occupancy. Our experiment results show that DG-CVS on GPU outperforms CPU by up to 57% before optimization and 145% afterwards. We also use DG-CVS as a real world application to explore the possibility of using Shared Virtual Memory (SVM) for tighter collaboration between CPU and GPU. However, SVM did not help improve the performance due to the overhead of address translation and atomic operations. We developed a microbenchmark to better understand the performance impact of SVM.

DEDICATION

To my husband Justin and our daughter Chloe.

ACKNOWLEDGEMENTS

My first and foremost thanks goes to my family. I thank my husband Justin Trotter who respected and supported me for my decisions. I also thank my daughter Chloe, who brightens my day with her laugh and love, who is my motivation of hard work. I greatly appreciate my parents for their help taking care of Chloe so I could focus on completing this thesis without any worry at home.

I owe many thanks to Dr. Byunghyun Jang, who is a wise and supportive adviser. He provided valuable inputs to my academic life as well as the important event of my life, while leaving me the option to chose. His guidance on my academic progression is invaluable and his work ethic will continue to influence me in my professional careers.

I would also like to thank all my current lab members David Troendle, Esraa Gad, Mason Zhao and Hossein Pourmeidani, as well as previous lab members Tuan Ta, Kyoshin Choo and Ajay Sharma for the countless discussions which inspire solutions and findings, some of which are presented in this thesis.

My thanks also goes to all the faculty members in the department for teaching their knowledge to me. I still have a long way to go but I would not be here if it weren't for their carefully planned lectures and well-organized assignments. Special thanks to Dr. H. Conrad Cunningham who enrolled me to the graduate program and was my first advisor, and also thanks to Dr. Dawn E. Wilkins who is an inspirational women leader in the department. It is my honor to have them in my thesis defense committee.

Last but not least, I'd like to thank Dr. Shuangzhang Tu from Jackson State University. He is always prompt at responding to my questions regarding DG-CVS, which contributes to my deeper understanding of this solver. This work is in collaboration with Dr. Shuangzhang Tu and is funded by the US Army Corps of Engineer.

TABLE OF CONTENTS

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF FIGURES vi

INTRODUCTION 1

BACKGROUND 4

OPENCL IMPLEMENTATION AND OPTIMIZATION 13

EXPERIMENT RESULTS 21

CONCLUSION 32

BIBLIOGRAPHY 33

VITA 37

LIST OF FIGURES

2.1 Solution Element (SEs) and conservation elements (CEs). Left: solution elements. Right: conservation elements. 6
2.2 A simplified computation flow of DG-CVS. 7
2.3 Divergent kernel code. 10
2.4 Visualization of divergent kernel code. 10
3.1 Breakdown of DG-CVS execution time. 14
3.2 Work flow of OpenCL program. 14
3.3 Kernel code after thread remapping. 16
3.4 Thread remapping. 16
3.5 Separate Memory VS Shared MemoryIntel (2015). 19
4.1 Performance comparison between CPU version and unoptimized GPU version. 25
4.2 Performance comparison between vanilla GPU version and no divergence GPU version. 27
4.3 Performance comparison between vanilla GPU version and SVM version. . . . 29
4.4 Kernel code for SVM microbenchmark. 30

CHAPTER 1

INTRODUCTION

A Riemann-solver-free high order space-time method has recently been developed to solve arbitrary space conservation laws (Tu, 2015) (Tu, 2013) (Tu et al., 2012). It is referred to as Discontinuous Galerkin Cell Vertex Scheme (DG-CVS). The DG-CVS is a highly accurate and efficient computational tool based on an unconventional numerical algorithm to simulate shallow water flows. The term DG means Discontinuous Galerkin method. It is a class of numerical methods for solving differential equations, recently widely used to solve fluid dynamic or electromagnetic problems. The DG provides DG-CVS with the benefit of high order accuracy. CVS means Cell-Vertex Scheme, which is inspired by Conservation Element/Solution Element (CE/SE) method (Chang and To, 1991). The CE/SE is a numerical framework for solving conservation laws in continuum mechanics. It employs a staggered space-time mesh. The DG-CVS enjoys the feature of Riemann-solver free from CE/SE.

The details of DG-CVS are well summarized in (Tu, 2015): The core idea of the method (DG-CVS) is to construct a staggered space-time mesh through alternating cell-centered (CE) and vertex-centered CEs within each time step. Inside each SE, the solution is approximated using high-order spacetime DG basis polynomials. The spacetime flux conservation is enforced inside each CE using the DG discretization. The unknowns are stored at both vertices and cell centroids of the spatial mesh. The solutions at vertices and cell centroids are updated at different time levels within each time step in an alternate fashion. The DG-CVS method involves processing a large number of mesh nodes. It solves a small non-linear equation system at each mesh node. There are millions of mesh nodes to be solved at each time step and each small system is independent from each other. Such characteristics make DG-CVS amendable to GPU's computation model.

GPU has proven to be an ideal platform for accelerating data and compute intensive applications in many fields (Krakiwsky et al., 2004) (Rodrigues et al., 2008) (Ta et al., 2015) (Tubbs and Tsai, 2011). Its large number of cores provides unprecedented computing power. A well-tuned program can reach up to several orders of magnitude of speedup (Munshi, 2009). However, based on SIMT (Single Instruction Multiple Thread) execution model, GPU requires all threads within a wavefront to execute in lock step. If threads within a wavefront have different execution paths caused by control flow (e.g., conditional branch), it causes *thread divergence* which drops hardware utilization significantly.

Moreover, the per-thread memory latency of GPU is significantly higher than that of CPU. Therefore, GPU uses zero-cost thread switching to hide latency among threads rather than shortening it to achieve good performance speedup. In order to hide the long latency, it is best to launch a large number of threads to provide GPU with enough candidate wavefronts to switch to when active wavefront is waiting for data to arrive (AMD, 2015b). A metric to measure whether there are enough candidate wavefronts is called *occupancy*. Occupancy is the ratio of the number of active wavefronts to the maximum number of wavefronts allowed per Compute Unit (CU) (AMD). A higher occupancy rate usually means a larger number of candidate wavefronts to keep the Arithmetic Logic Unit (ALU) busy.

A recent trend in industry is to manufacture CPU and GPU on a single die, combining host and device memory space both physically and logically. This hardware architecture, together with software support (e.g., OpenCL 2.0 specifications), introduces a new memory model to achieve true heterogeneous computing. Shared Virtual Memory (SVM) allows CPU and GPU to access the same memory space at the same time, enabling a tighter collaborative relationship between CPU and GPU. Also, data structures that rely on pointers such as trees and graphs are accessible between CPU and GPU without flattening the whole structure. At the same time, SVM eliminates the data transfer overhead. The benefit that SVM brings is tremendous. However the address translation overhead can be a performance bottleneck for applications that uses SVM (Vesely et al., 2016).

In this thesis, we present the acceleration of DG-CVS on CPU-GPU heterogeneous processors. We address the challenges that DG-CVS poses on the GPU architecture such as thread divergence and low kernel occupancy. We also explore the impact of SVM on accelerating DG-CVS. The contributions of this thesis are summarized below:

- We analyze DG-CVS in detail and demonstrate that a complex scientific solver such as DG-CVS can benefit from CPU-GPU heterogeneous computing.
- We propose to minimize the impact of control flow within kernel by thread remapping. By pre-checking condition and grouping threads on the same execution path together, thread remapping can effectively eliminate or minimize thread divergence within wavefront.
- We propose to address low kernel occupancy caused by high register pressure with a number of techniques, including code change, Local Data Share (LDS), and *volatile* keyword.
- Lastly, we explore the effect of using SVM feature to enable CPU-GPU collaboration in DG-CVS. We also developed microbenchmark to understand the reason for performance degradation caused by SVM.

CHAPTER 2

BACKGROUND

In this chapter, we first introduce the background of DG-CVS solver, and then show the computation flow of DG-CVS, followed by the details of GPU architecture regarding thread divergence, kernel occupancy and shared virtual memory.

2.1 Background of DG-CVS

Shallow water equation is widely used as a mathematical model to numerically simulate dam break, river inundation, long wave run-up and tide of ocean in coastal and civil engineering. There is close mathematical and physical analogy between the shallow water flows and compressible flows. The hydraulic jumps and bores are analogous to the stationary and moving shock waves in compressible gas flows. Therefore, the numerical methods used to solve the SWE often mimic those for solving the compressible Euler equations. Such methods usually employ some (approximate) Riemann solvers to provide numerical fluxes.

Riemann solvers are used to provide unique interface fluxes in solvers based on the finite volume method, discontinuous Galerkin method and some other methods. Numerical methods based on Riemann solvers have achieved tremendous success in solving hyperbolic systems (e.g., compressible Euler equations, where the eigen structure of the system is clearly known) in the past several decades. However, when physical process gets more complicated, for example, magnetohydrodynamics (MHD), the success of Riemann-solvers is far from satisfactory. For example, the Roe scheme has been modified to solve the MHD system. Roe's scheme requires eigen-decomposition and becomes very complicated in MHD equations. Moreover, due to the complexity and non-strict hyperbolicity of the MHD system, the validity of the eigen-system of the MHD system is not unanimously agreed upon among researchers.

Therefore, Riemann-solver free approaches are highly desirable to avoid the uncertainties caused by imperfect Riemann solvers.

DG-CVS is a novel high order Riemann-solver-free numerical method for general hyperbolic conservation laws. It provides several important features. It has arbitrary high-order accuracy in both space and time because space and time are handled in a unified way based on space-time flux conservation and high-order space-time discontinuous basis functions. It does not need a (approximate) Riemann solver to provide numerical fluxes as needed in finite volume or traditional Discontinuous Galerkin (DG) methods. The Riemann-solver-free feature offers two-fold advantages. First the Riemann-solver-free approach eliminates some pathological behaviors associated with some Riemann solvers. Second, it is suitable for any hyperbolic Partial Differential Equation (PDE) systems whose eigen structures are not explicitly known. The DG-CVS based solvers have been successfully applied to solve scalar advection-diffusion equations, compressible Euler equations, shallow water equations, the level set equation and magnetohydrodynamics equation. It is also reconstruction-free and suitable for arbitrary spatial meshes. DG-CVS only needs information at the immediate neighboring nodes to update the solutions at the new time level, which makes it easy to parallelize the solver. More details on the features of DG-CVS are found in (Tu, 2015).

DG-CVS is inspired by the spacetime Conservation Element/Solution Element (CE/SE) method and the Discontinuous Galerkin (DG) method. The solver integrates the best features of the two methods, i.e. the Riemann-solver-free feature of the CE/SE method and the high-order accuracy of the DG method. The solver constructs a staggered spacetime mesh through alternate cell-centered CEs and vertex-centered CEs (Figure 2.1 (right)) within each time step. Note that CEs at the vertex level are defined via the *dual mesh* obtained by connecting the surrounding cell centers and face centers. The difference between SEs and CEs is that the SE includes the volume-less vertical spike. Inside each SE (Figure 2.1 (left)), the solution is approximated using high-order spacetime DG basis polynomials. The spacetime flux conservation is enforced inside each CE using the DG discretization. The unknowns

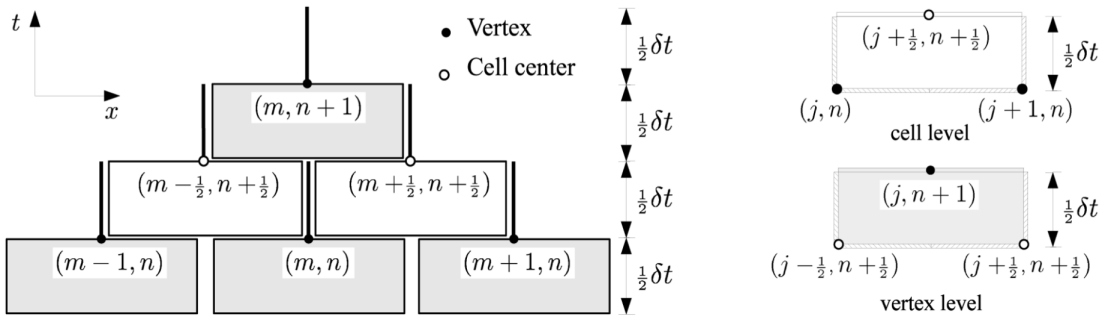


Figure 2.1. Solution Element (SEs) and conservation elements (CEs). Left: solution elements. Right: conservation elements.

are stored at both vertices and cell centroids of the spatial mesh. However, the solutions at vertices and cell centroids are updated at different time levels within each time step in an alternate fashion. Due to the alternate cell-vertex solution updating strategy and its DG ingredient, the method has been termed as the Discontinuous Galerkin Cell-Vertex Scheme (DG-CVS).

2.2 Computation Flow of DG-CVS

The computation flow of DG-CVS is shown in Figure 2.2. The solver starts off by reading in input, which contains spacetime mesh information. Such information is processed and then populates a list of cells, vertices, and faces with corresponding information such as their positions and components. When all information is stored in place, simulation is launched. This is the most time consuming part of DG-CVS solver. There are multiple time steps, during which the solution is updated with two steps: The first step updates the solution based on the known cell-centroid solution at the previous time level (a function named SolutionCell()). The second step updates the solution based on the known vertex solution at the previous time level (a function named SolutionVert()). Initial values for solutions are given at the first time step, and the rest of time step will use the previous time step's solutions.

There are three loops in SolutionCell(), each of which iterates through the list of faces,

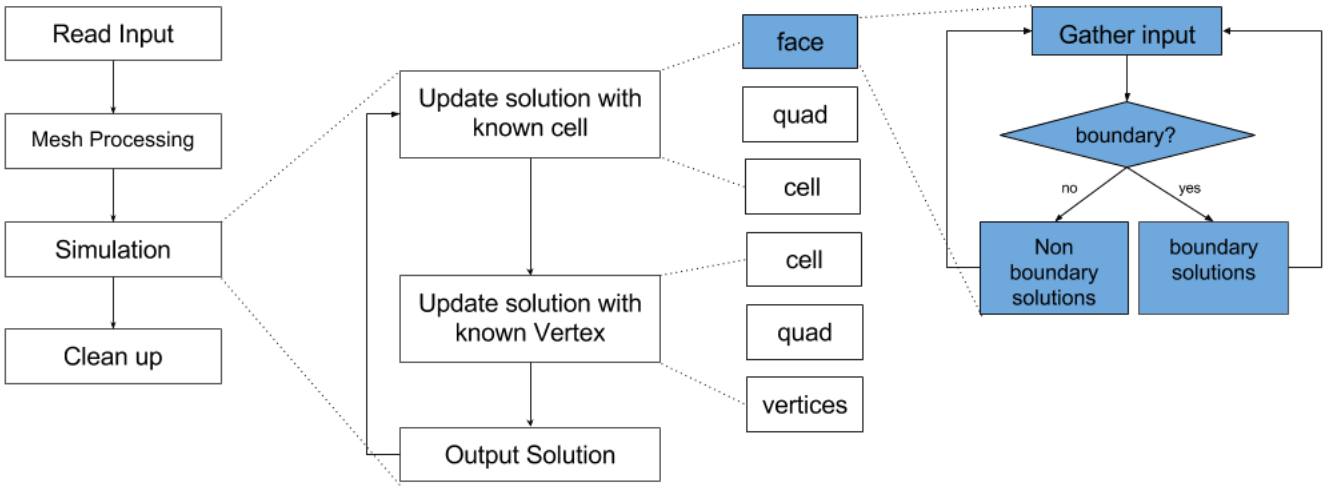


Figure 2.2. A simplified computation flow of DG-CVS.

quads, and cells of the mesh. Similarly, there are also three loops in SolutionVert() that go through the list of cell, quads and vertices. This thesis targets the face loop in SolutionCell(). In every iteration of the face loop, the first step is to gather input for computation for faces. Once input is ready, the solver will check whether this face is a boundary or non boundary face because boundary faces require a different set of functions to update solutions. Non boundary faces are the majority iterations of the for loop. Each face is independent therefore this loop is suitable for GPU acceleration. After face loop finishes, the solver has two more loops for quad and cell, which are the future targets of GPU acceleration.

When face, quad, cell loop finish, the solver continues to update solution with known vertex, which consists of three similar loops as the ones in cell solutions. Then output is written to a buffer and the solver finishes one time step. After the solver runs a set number of time steps, the simulation process is done and the rest of the program takes care of cleaning up.

2.3 CPU-GPU Heterogeneous Computing

GPU offers unprecedented level of computing power by launching a massive number of threads that run in parallel. More and more applications benefit from GPU's computing power as GPUs become more general-purpose. Well-tuned applications can receive several orders of magnitude of speedup.

However, due to its different architecture, programming model, and memory spaces, GPU faces some challenges that can seriously hurt the performance if not used correctly. DG-CVS also faces several performance challenges. This chapter focuses on the background information particularly associated with these challenges.

2.3.1 Terminology

Before we discuss the challenges that DG-CVS poses on GPU, some terminology used in this thesis needs to be cleared up. GPU takes advantage of SIMD execution model. It launches massive number of threads to execute functions in parallel to gain speedup. Such functions are referred to as *kernels*. Threads are called *work items* in OpenCL terminology. However, in this thesis, we use *threads* and *work items* interchangeably depending on context. *Work items* are grouped in *work groups* to execute in a *Compute Unit*. The size of *work groups* can be determined by programmers. However, *Compute Unit* divides *work groups* into equal-sized thread groups called *wavefronts* to run in parallel. The size of *wavefront* is hardware dependent and can not be changed by programmers.

There are several memory regions on the GPU. *Private memory* is the fastest and is only accessible to individual *work items*. However, if one allocates more private memory than hardware limitation, part of this allocation will be spilled to *global memory*, which is much slower than the on-chip *private memory*. *Local memory* is also on-chip memory, which can be equally fast as *private memory* if handled correctly. It is accessible to work items within the same work group. *Global memory* is visible to all work items but is very slow compared to on-chip memory.

2.3.2 Thread Divergence

GPU's massive parallel processing power takes advantage of the Single Instruction Multiple Data (SIMD) execution model. Such model simplifies the hardware, however, the complexity of control flow can seriously impact performance (AMD, 2015a). After kernel is launched on GPU, the hardware scheduler schedules work-groups to a Compute Unit (CU). Once scheduled, work-groups cannot migrate to other CUs. A workgroup is then divided into equal-sized thread groups called *wavefronts*. Our test hardwares, AMD GPUs, have a work-group size of 64 work-items. Work items inside a wavefront will execute the instruction in a SIMD fashion, meaning that all work items inside a wavefront will execute the same instruction at the same time. In some situations where some work items follow one direction and other work items follow another, the wavefront executes each direction serially. Figure 2.3 is an example of thread divergent code, and the visualization of its execution. Figure 2.4 visualizes the execution of two wavefronts for the code snippet in Figure 2.3. Threads that will execute *if* branch are marked in blue, and those executing *else* branch are marked in red. And it is clearly shown that thread divergence happens within wavefront.

In Figure 2.3, each work item is assigned an ID, namely the global ID, and it is put in local variable `tid`. The built-in function `get_global_id(0)` returns the global ID in the first dimension for each work item. All work items evaluate whether the condition `Array[gid]` is positive or not. Then all of them go through the *if* path. The work items that do not satisfy the condition are invalidated. The same thing happens for the *else* path. In the end, all work items converge and write the result back to `Array[gid]`. In this case, the hardware utilization becomes 50% on average.

Thread divergence can severely impact the performance of GPU, depending on degree of divergence. It is one of the most important issues that must be addressed if we want to achieve the maximum speedup for this application.

```
tid = get_global_id(0);
if(Array[tid] >= 0){
    Array[tid]++;
}else{
    Array[tid]--;
}
```

Figure 2.3. Divergent kernel code.

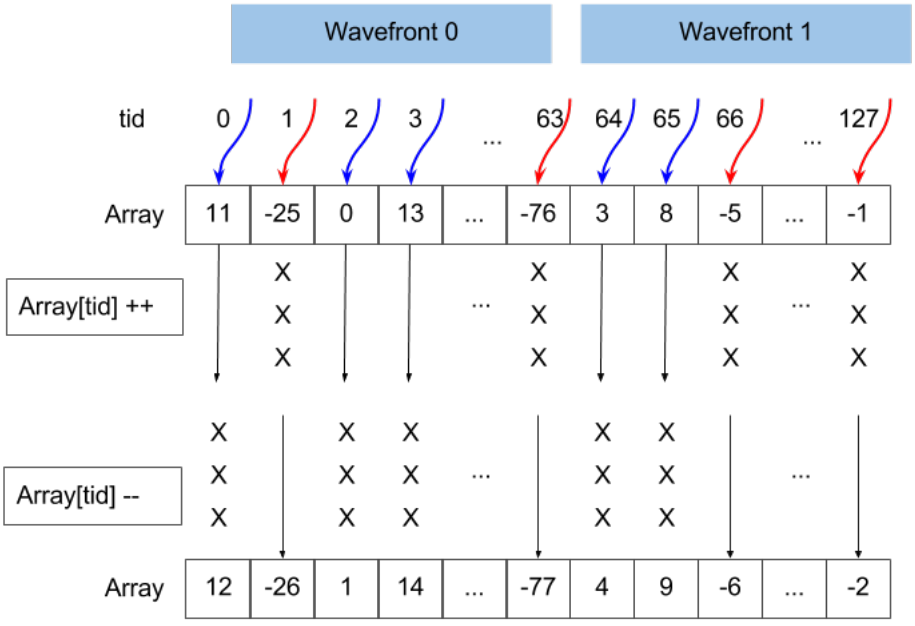


Figure 2.4. Visualization of divergent kernel code.

2.3.3 Kernel Occupancy

Kernel occupancy is the ratio of active wavefronts to the theoretical maximum concurrent wavefronts per CU. It is an easy way to measure the hardware resources utilization.

To understand kernel occupancy, we need to briefly explain the underlying hardware execution model. GPU hardware architecture is designed to hide memory latencies by dispatching a large number of threads and low cost thread switching. When memory access from the active wavefront stalls the kernel execution, CU switches to another ready wavefront while waiting for data. It is therefore important to schedule enough workgroups for each CU to keep it busy. GPU dispatches work items into workgroups to execute kernel code. The size of work group is configured from host code, which is recommended to be a multiple of wavefront size to best utilize the hardware. The size of wavefront is architecture specific. For our experiment machines from AMD, the wavefront size is 64 work items. The number of wavefronts can be assigned to one CU is constrained by three factors.

2.3.3.1 Work group size

The first factor is the granularity of workgroup size. It is important to properly configure the workgroup size because one workgroup cannot be separated into different CUs. For example, the theoretical maximum number of wavefronts per CU is 40 on the hardware we tested. If we configure workgroup size to be 6*64 work items, there are 6 wavefronts per work group. Then, each CU can contain maximum of 6 work groups, resulting 36 wavefronts in total. The leftover 4 wavefronts are wasted. However, if workgroup size is 4*64 work items, each CU can schedule a maximum of 10 work groups, then all 40 wavefronts are scheduled.

2.3.3.2 Register count

The latter two factors are hardware constraints. One is the number of registers needed by each work item. In our experiment setup, each SIMD in a CU has 16384 registers available. They are shared by work items in one wavefront. So every work item can use a maximum of 256 registers. One CU consists of 4 SIMDs, so when calculating occupancy

in terms of CU, we need to scale the number by 4. For example, if every work item in a wavefront uses 255 registers, we can schedule 1 wavefront (64 work items) per SIMD, so there are 4 wavefronts per CU. In this example, kernel occupancy is 10%. A "simple" kernel can reach up to 100% kernel occupancy since it may use the small number of registers, while a computation intensive kernel with large amount of work to be done by each work item would need lots of registers, and therefore the occupancy is usually low.

2.3.3.3 Local memory (LDS)

The other hardware constraint is local memory, also known as LDS (local data share) on AMD platforms. In GPU memory system, there are three memory spaces: global memory, local memory, and private memory. Local memory is memory space shared by work items within one workgroup, and cannot be accessed by work items outside of this work group. It can be just as fast as registers if managed properly. Therefore, it is also regarded as software managed cache. Besides fast access time, it also allows work items in the same work group to cooperate, or share data. However, if each work item tries to allocate more LDS, the amount of LDS needed for one work group increases, resulting less number of work groups per CU.

One last thing to note for kernel occupancy is that it increases or decreases in steps rather than linearly. Because even by tweaking work group size or modifying the kernel, extra resources in a CU are freed up. If such resources are not enough to schedule another work group, occupancy would remain the same. In the example case in subsection 2.3.3.2, the number of registers used by each work item needs to below 128 in order to get 20% occupancy. Fine tuning the configurations across work group size, LDS and registers is the key to increase kernel occupancy.

CHAPTER 3

OPENCL IMPLEMENTATION AND OPTIMIZATION

To identify candidate loops to offload to the GPU, we first profile the program and acquire the running time of each section. The profile information is shown in Figure 3.1. As we can see, the majority of the time in the main program is for simulation. Simulation process can be further divided into two parts: SolutionCell and SolutionVertex. There are three *for* loops each in SolutionCell and SolutionVertex. We accumulate the running time of each loop in each time step. This thesis targets the most time consuming one: the for loop that iterates through all faces (referred to as face loop) to update unknowns in SolutionCell() function, which is highlighted in red box in Figure 3.1.

In this section we first demonstrate the work flow of the implementation of DG-CVS on a GPU. Then we present our optimizations targeting thread divergence, low kernel occupancy, and analysis on SVM.

3.1 Work Flow of OpenCL Program

In order to launch a kernel on the GPU, the CPU must set up the OpenCL platform, device configuration as well as kernel creation in the host code. This is a one time cost for the GPU program regardless of how many kernels are launched later in the program. The subsequent steps follow the work flow shown in Figure 3.2 which is executed repeatedly for *ts* times, where *ts* is the number of time steps. The first step is to copy data from original data structure to the memory buffer allocated on GPU. Then, `CL_MEM_USE_HOST_POINTER` flag is used when creating memory buffer to enable zero copy feature, which minimizes data transfer time. Next step is to set memory buffers as kernel arguments. After this step, host code is ready to launch kernel and let GPU execute. When kernel code finishes execution,

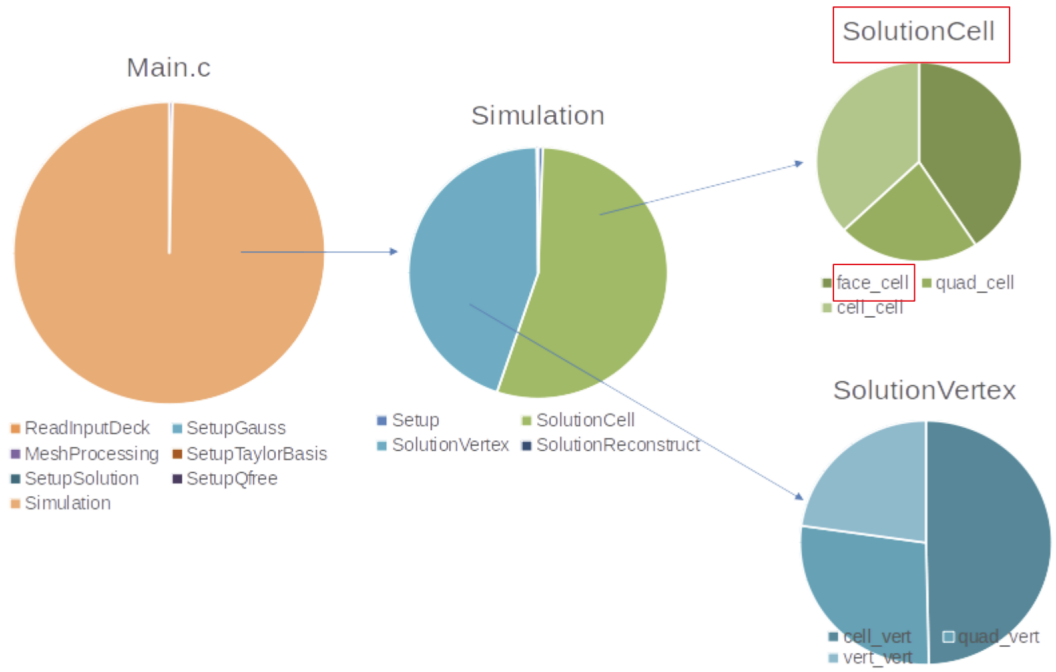


Figure 3.1. Breakdown of DG-CVS execution time.

Figure 3.2. Work flow of OpenCL program.

the host copies output back to original data structure, and releases memory buffer on GPU.

3.2 Optimization on Thread Divergence

Section 2.3.2 explains the underlying architecture of GPU, and the reason of thread divergence. In the past, researchers have tried to minimize thread divergence impact by software techniques (Han and Abdelrahman, 2011) (Zhang et al., 2010) or hardware modification (Brunie et al., 2012) (Fung et al., 2007) (Narasiman et al., 2011). In many cases,

algorithm redesign is often needed to achieve the most optimal performance (Chakroun et al., 2013). In this thesis, we approach divergence issue with slight modification of original algorithm.

Threads inside a wavefront execute the same instruction in lock step. Based on this architecture and execution model, we can remap thread ID to minimize number of divergent wavefronts down to only one wavefront or even completely remove divergence happening in this function. It is an effective way to minimize the divergence (Yoshitake and Keiji, 2015). Thread remapping requires the host to pre-check data for GPU kernels, then gather the index of threads taking *if* path and *else* path. After thread remapping, work items within one work group execute either *if* path or *else* path. If the number of *if* path is a multiple of 64, thread remapping completely removes divergence. Otherwise, this method only leaves one divergent wavefront, which is not a significant factor for performance.

To implement thread remapping, first, we created an array that records the index that is used to remap threads, referred to as `idx_arr`. The host checks the condition where the divergence happens. In an example case, it is to check if `Array[tid]` is greater than or equal to zero, and writes the index of elements that satisfies the condition check (taken) to `idx_arr`. Repeat the same steps for elements that fails the condition check (non-taken) and pass `idx_arr` to kernel.

After passing `idx_arr` to kernel as an argument, threads are mapped to this index array `idx_arr` instead of the index of `Array` by executing the code shown in Figure 3.3. Figure 3.4 shows the visualization of thread execution path after remapping.

3.3 Optimization on Low Kernel Occupancy

Kernel occupancy refers to the ratio of active wavefronts to the theoretical maximum concurrent wavefronts per compute unit (CU). In general, a high occupancy kernel can better hide the global memory access latency, therefore often leads to better performance. Optimizing kernel occupancy requires the programmer to find the balance between the three

```
tid = get_global_id(0);
idx = idx_arr[tid];
if(Array[idx] >= 0){
    Array[idx]++;
}else{
    Array[idx]--;
}
```

Figure 3.3. Kernel code after thread remapping.

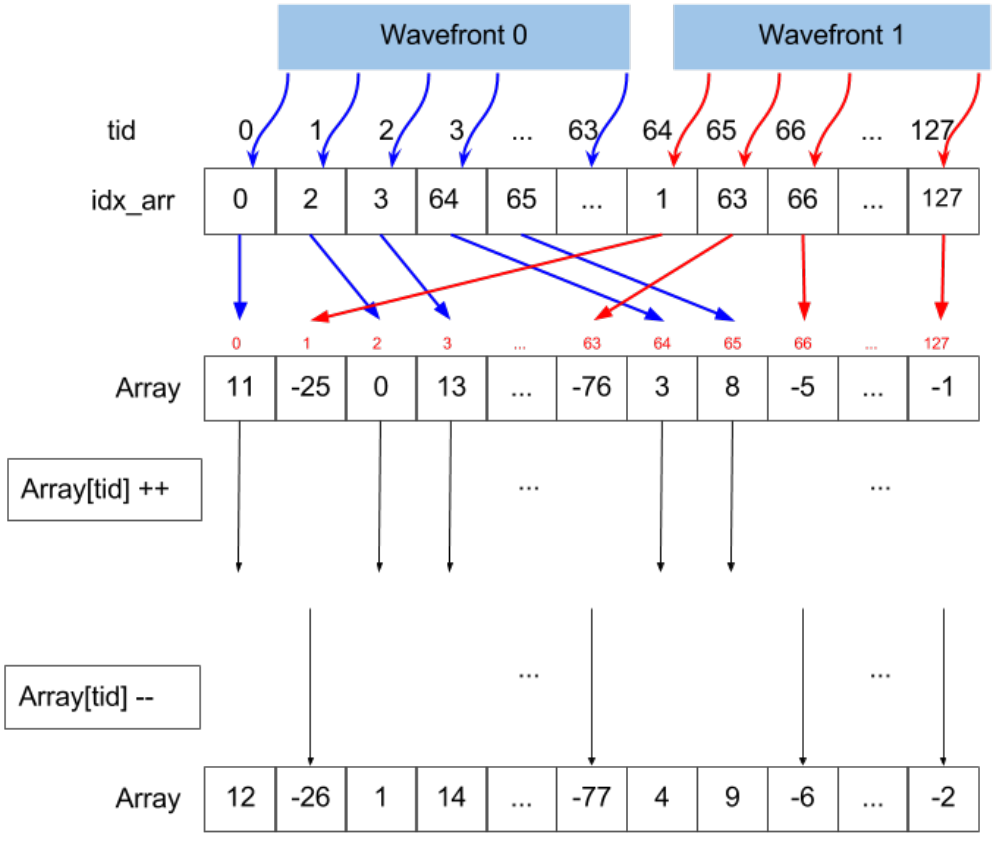


Figure 3.4. Thread remapping.

limiting factors discussed in Chapter 2.3: work group sizing, LDS, and registers.

To find out which factor is a constraint for kernel occupancy, we use a profiler tool such as CodeXL (AMD, 2012). According to the profiling result, VGPR (vector general purpose register) is a limiting factor for kernel occupancy. Each work group uses 169 VGPR, which limits the number of wavefronts to 4 out of 40 which is the device limit. Kernel occupancy is only 10% because of high register pressure. Based on the analysis of kernel code, we use three techniques to minimize the use of VGPR: code change, use of LDS, use of *volatile* keyword.

The original DG-CVS source code declares all local variables at the first line of function. Such programming style can lengthen register life, which in turn creates register pressure. Furthermore, it does not provide compiler proper scope information of some variables for optimizations. Code change is required to reduce the live ranges of register. We declare local variables only when initialized with value, and within proper scope. Some temporary variables are replaced with either recomputation, or completely eliminated in case of redundancy.

Some programmers also use LDS to alleviate register pressure (Gaster et al., 2012) (Pollefeys) . LDS is also referred to as software managed cache. It is slightly slower than register, but much faster than global memory access. Programmer can identify the highly reused data and manually "cache" them in LDS. However, just like other parameters that affect performance, LDS and register are antagonists: one is improved at the cost of the other. In unoptimized version of GPU kernel in DG-CVS, use of LDS is none. As we mentioned in Chapter 2.3, kernel occupancy is limited by register and LDS, also it changes in steps. As we increase the use of LDS, LDS quickly rises up to become a limiting factor. When register count decreases to allow next level of occupancy which is 20%, but due to the limitation of LDS, occupancy only increases to 12.5%.

Using *volatile* keyword is another way to decrease register use. If a variable is declared as volatile, the compiler assumes that this variable is likely to be modified by other threads.

Every reference to this variable would be loaded from its declared memory region, either in global memory or shared memory, instead of being kept in registers.

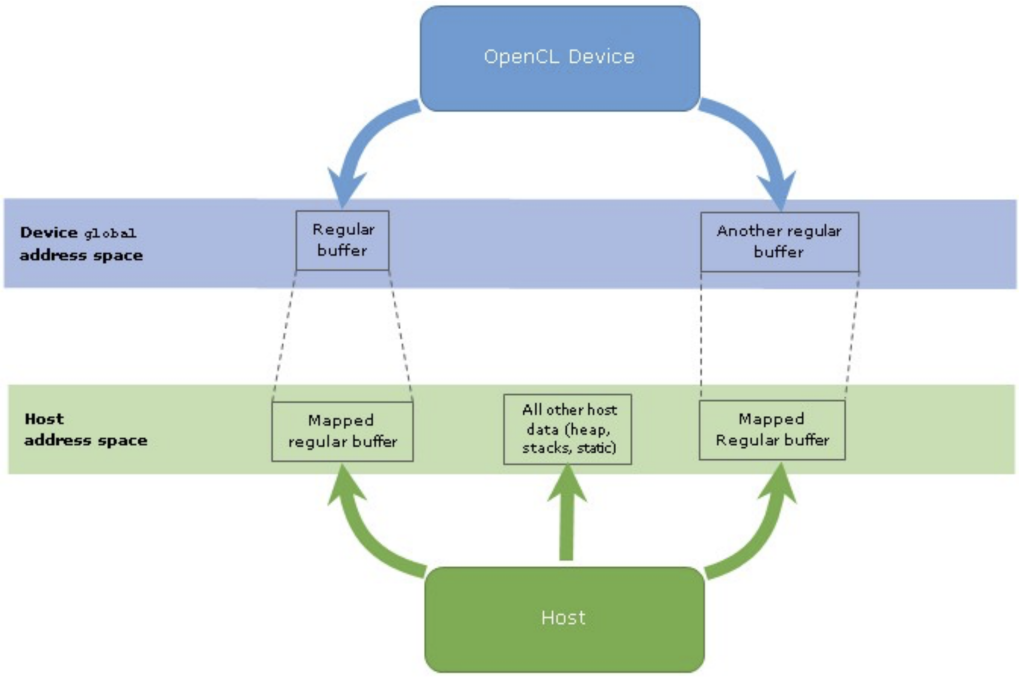
3.4 Shared Virtual Memory

A recent trend in GPU computing is transitioning from discrete GPU that are physically separated from CPU to tightly integrated CPU and GPU on a single die. Physically connecting CPU and GPU memory opens up an opportunity to share memory address between CPU and GPU. SVM is the landmark feature of OpenCL 2.0 (Howes and Munshi, 2015). The benefits of SVM are the following: It removes the overhead of data copy. GPU can read pointer type data structures such as graph and trees, that are declared in CPU and directly work on such data structures without flattening them. Lastly, atomic operation is available on SVM which enables synchronization between CPU and GPU to allow tighter collaboration between the two. If used correctly, SVM can boost performance by 30% comparing to using regular memory buffer in OpenCL 1.2 (Robert Ioffe) (Mukherjee et al., 2016).

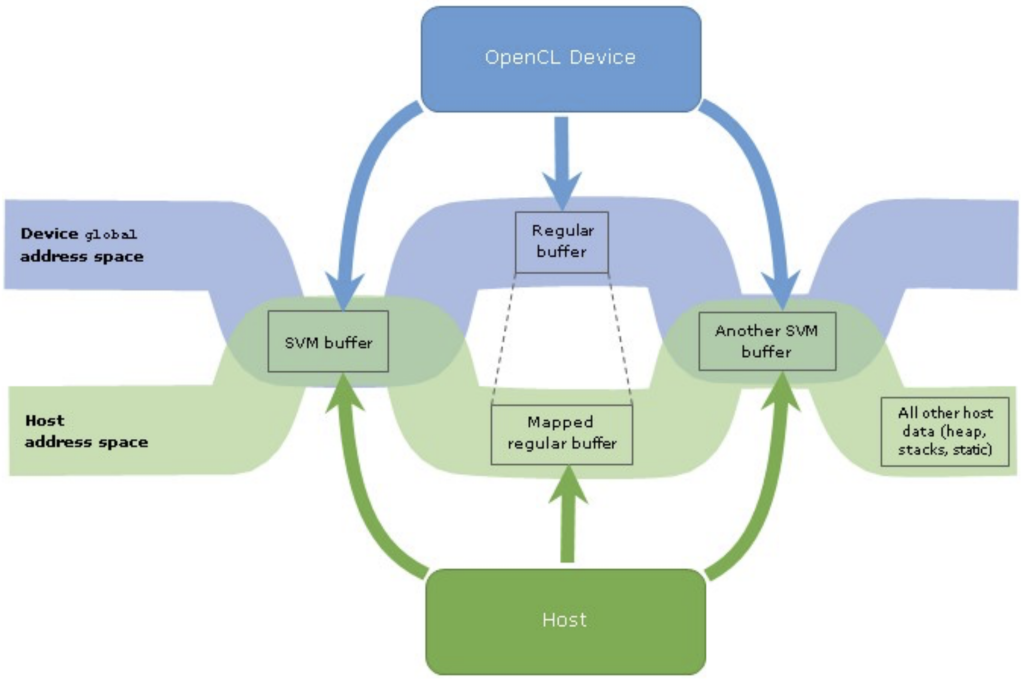
The difference between regular memory buffer and SVM buffer is shown in Figure 3.5. Figure 3.5a shows traditional memory address and data transfer in OpenCL 1.x style code. In 1.x OpenCL, memory buffer is first created on host side and passed to device side. If host wishes to read a memory buffer, a read request must be enqueued on the command queue. This extra step is an overhead at host code. SVM allows CPU and GPU to seamlessly access the same data structure without copying back and forth.

The overlapping area in Figure 3.5b shows SVM memory region. A SVM buffer is no longer owned just by device. Both host and device can read or write to this memory region at any given point. The programmer does not need to worry about coherence between the two because underlying hardware takes care of it.

SVM also enables GPU to directly work on pointer type data structures such as trees or linked lists that are created in SVM region, by simply passing the root pointer to the



(a) Separate Memory of CPU and GPU.



(b) Shared Virtual Memory.

Figure 3.5. Separate Memory VS Shared MemoryIntel (2015).

device. Before this feature was introduced, the programmer had to flatten the tree to one dimension and calculate indexes to access nodes.

Combined with atomic operations, SVM allows fine-grained lightweight synchronization between host and device. Concurrent modification on the same address is visible to both host and device, and memory consistency is guaranteed without issuing new commands to the command queue. SVM brings out new collaboration relationships between CPU and GPU other than traditional master and slave relationship (Ta et al., 2017).

SVM is a landmark feature from OpenCL 2.0 that would bring performance and programmability impact to many applications. It has success in accelerating shallow water equations (Mukherjee et al., 2015) and in image processing (Junkins, 2015). However DG-CVS experiences slow down after incorporating SVM. We developed micro benchmark for deeper understanding on this negative impact. Micro benchmark and analysis are shown in Chapter 4.6.

CHAPTER 4

EXPERIMENT RESULTS

Our performance evaluation considers both CPU implementation and an unoptimized vanilla version of GPU implementation as base cases. In this chapter, we first explain our test environment, and then show the impact of applying optimization techniques mentioned in Chapter 3 along with analysis of the impact of each proposed method. Our test results are summarized as follow:

- Our unoptimized GPU version achieves performance gain up to 57% speedup compared to CPU implementation.
- By minimizing thread divergence, we increase kernel speedup significantly across all test cases by up to 56% when compared to unoptimized GPU kernel, and 145% compare to CPU version.
- Kernel occupancy increases from 10% to 12.5%.
- SVM (Shared Virtual Memory) enables CPU and GPU collaboration at a finer granularity. However, the address translation overhead of SVM and cost of atomic operations on this memory region outweighs the benefit it brings, causing the kernel to run up to significantly longer than on regular buffers.

4.1 Description of Test Cases

We included 24 test cases from DG-CVS. Each test case provides a different set of mesh information, which affects the performance. Table 4.1 lists all test cases. There are two main categories of test cases: one that solves the advection equation (denoted as *adv-sin*),

and the other solves the burgers equation (denoted as *burgers*). Each category can be further divided into 2 groups, p1 and p3. P1 stands for second order accuracy, and P3 for fourth order accuracy. p1 has four unknowns at each space time node, while p3 has 15 unknowns at each space time node. Therefore, p3 produces more accurate results than p1 with the price of being more expensive. Next, two shapes of mesh cell are considered: quadrilateral and triangle. Next level of division is the granularity of mesh cell size, which increases from 10 to 40. Generally, for the same domain, finer meshes have larger number of mesh cells. However, the given test cases are of different domains. The number of cells increases as the mesh cell size increases. The number of GPU threads launched is mapped to the number of mesh cells, because the solutions stored in each cell are independent, which can be done in parallel.

4.2 Experiment Setup

We consider serial CPU version and an unoptimized GPU version as our base cases. To measure the performance, wall clock timer and OpenCL event are used to get the overall time and kernel time respectively. CodeXL is used to get hardware counter information. We check the correctness by comparing the output of serial version and every GPU version.

We use AMD A10-7850K APU (code name "Kavari") to test all versions of implementation. This tightly coupled heterogeneous processor combines 4 CPU cores running at 3.7 GHz and 8 Radeon R7 GPU compute units running at 720 MHz. The machine is running Ubuntu 14.04 64-bit Operating System with 12 GB main memory. We run the CPU version base case on the same quad-core CPU on the APU. OpenCL 1.2 is used for both unoptimized and optimized GPU versions, while OpenCL 2.0 is used for SVM GPU version. SVM is a feature in OpenCL 2.0. Therefore, OpenCL 2.0 flag is required in order to use SVM. The compiler will produce different assembly code using different OpenCL version.

Equation	Order of Accuracy	Cell Shape	Cell Size	Number of Faces
adv-sin	p1	quadrilateral	10x10	220
			20x20	840
			40x40	3280
		triangle	tri-10	404
			tri-20	1576
			tri-40	6224
	p3	quadrilateral	10x10	220
			20x20	840
			40x40	3280
		triangle	tri-10	404
			tri-20	1576
			tri-40	6224
burgers	p1	quadrilateral	10x10	220
			20x20	840
			40x40	3280
		triangle	tri-10	380
			tri-20	1480
			tri-40	5840
	p3	quadrilateral	10x10	220
			20x20	840
			40x40	3280
		triangle	tri-10	380
			tri-20	1480
			tri-40	5840

Table 4.1. A list of test cases.

4.3 Impact of Using GPU for Acceleration

In this section we show the performance improvement of using GPU for acceleration. In short, after porting and optimizing GPU kernel, DG-CVS achieves performance gain up to 147% compared to CPU version, and 57% speedup if overhead is included. Figure 4.1 shows all performance comparison between CPU version and unoptimized GPU version (denoted as *vanilla*). The X-axis is execution time in micro seconds, while the Y-axis is test cases.

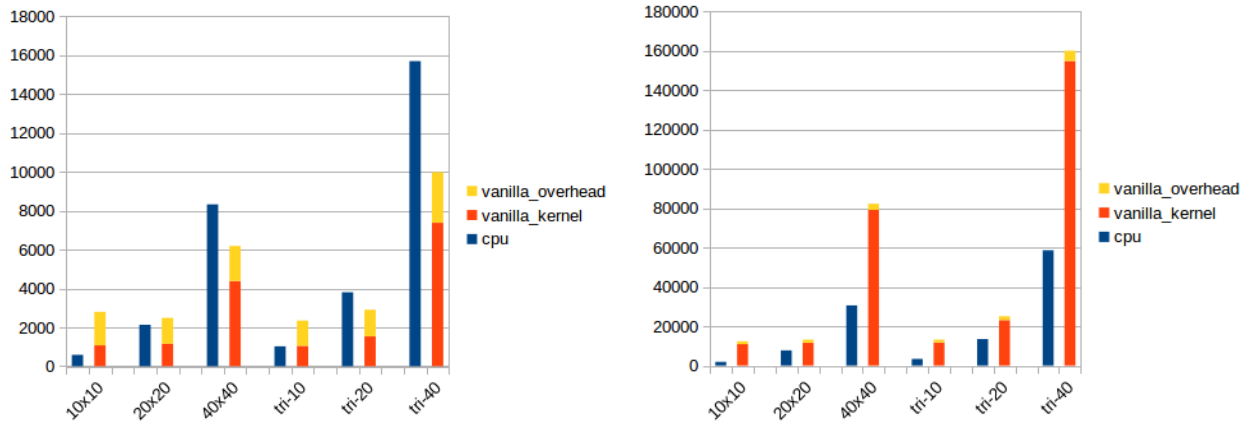
All improved cases fall into p1 cases. Test cases with a smaller mesh size (i.e., less number of faces) did not show speed up because the overhead of data copy and launching GPU kernel can not be justified with small input size. As the mesh size gets larger, the benefit of using GPU becomes more obvious. All accelerated cases are of large mesh size cases. This shows that given large enough data set and parallelism, the overhead of using GPU pays off.

P3 cases experience worse performance on GPU. As mentioned in Chapter 4.1, p3 cases produce more accurate results at the cost of being more expensive. The kernel code remains the same for p1 and p3 cases. However, there are several loops inside kernel code, and the control variable for those loops are 4 times bigger in p3 cases than that of p1 cases. Within these loops, global memory access quadruples for p3 cases. Moreover, such global memory access has a larger stride for p3 cases which result in poor cache hit. Lastly, these global memory accesses happen inside a tight nested loop, leaving GPU little choice to switch to other wavefront to hide the latency since they are in similar code region. These are reasons why p3 cases did not get acceleration on GPU.

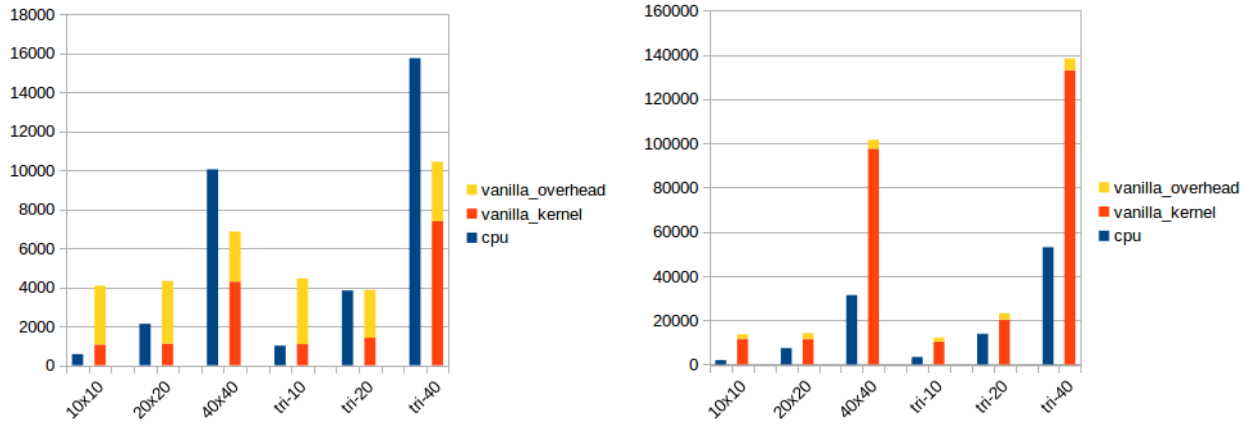
4.4 Impact of Minimizing Thread Divergence

In this section we show the improvement of kernel performance after thread remapping.

In DG-CVS face kernel, thread divergence happens because the threads that works on both boundary faces and non boundary faces are interleaved. We move condition checking



(a) Test cases adv-sin. Left graph is p1 cases, right graph is p3 cases.



(b) Test cases burgers. Left graph is p1 cases, right graph is p3 cases.

Figure 4.1. Performance comparison between CPU version and unoptimized GPU version.

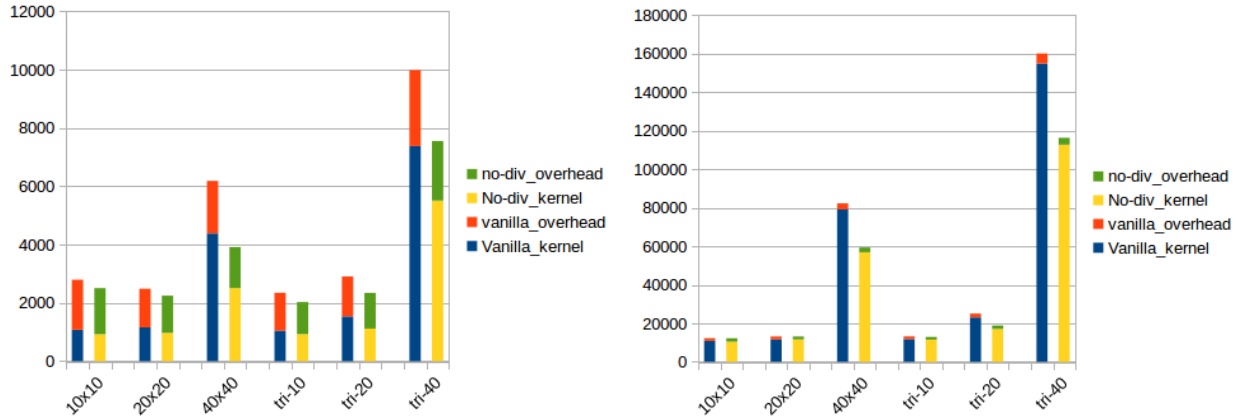
out side of kernel body and record the indexes of non boundary faces to the first part of an index array and the indexes of boundary faces to the second part of array. This step groups threads that takes non-boundary path together, and threads that takes boundary path together, so that threads within the within the same wavefront takes the same path.

Thread remapping significantly boosts kernel performance and the overhead from condition checking and passing an additional memory buffer is negligible. The degree of improvement depends on the severeness of divergence (i.e., the number of divergent paths and whether divergence occurs within a wavefront). In DG-CVS, there are two divergent paths in face kernel: boundary and non boundary, and they are interleaved which causes divergence to happen within wavefronts. Figure 4.2 shows performance comparison of kernel before thread remapping (denoted as *vanilla*) and after thread remapping (denoted as *no-div*). The X-axis is execution time in micro seconds, while the Y-axis is test cases. Comparing to vanilla GPU version, there is a 56% increase in best case, and 22% on average for all 24 test cases. Comparing to CPU version, there is up to 145% speed up. We also see an interesting trend which is that as the number of concurrent threads increases, the impact of thread remapping rises.

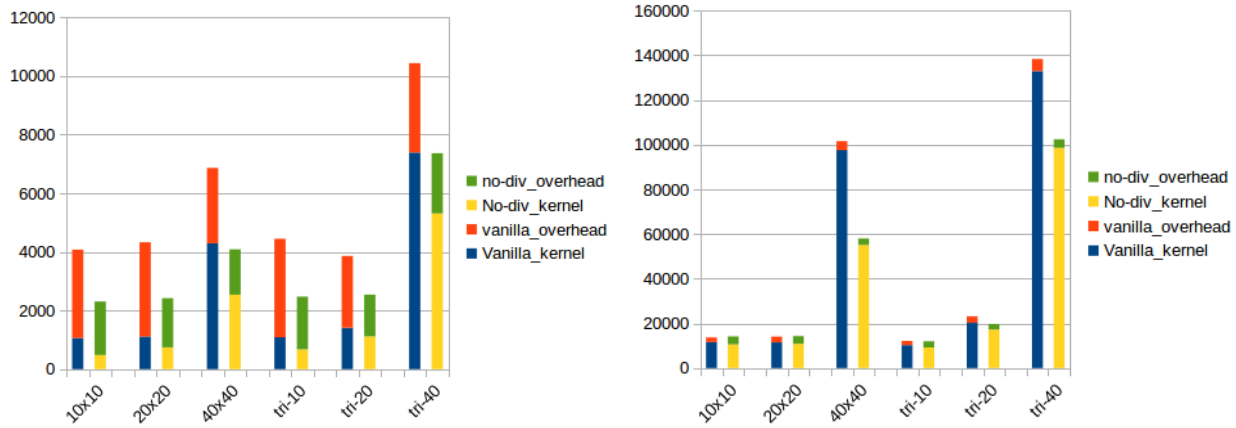
4.5 Impact of Higher Kernel Occupancy

Multiple techniques have been used to lower register pressure in order to increase the kernel occupancy. The vanilla version of GPU kernel uses 169 registers, limiting the kernel occupancy to 10%. Kernel occupancy changes in steps rather than continuously, and is affected by multiple factors such as group size, register, and local data share (LDS). To reach the next level of occupancy, which is 20%, register count must be lower than 129.

We did the following changes to reduce the use of registers. The first is to change the location of local variable declaration. Instead of declaring all local variables at the top of function, we declare them when they are used, and place them in appropriate scope. These changes can shorten register life, giving compiler more information to produce machine code



(a) Test cases adv-sin. Left graph is p1 cases, right graph is p3 cases.



(b) Test cases burgers. Left graph is p1 cases, right graph is p3 cases.

Figure 4.2. Performance comparison between vanilla GPU version and no divergence GPU version.

with fewer registers. This code change lowers the VGPR count from 169 down to 149. Secondly, using *volatile* keyword drops register count from 149 to 142. Lastly we use LDS to alleviate register pressure. LDS is on chip memory region shared by threads within a work group. It is visible to all threads within a work group and accessing speed is much shorter than global memory but slightly longer than registers. Note that LDS is also one of the limiting factor for occupancy. After moving some local data to LDS, we successfully lowered register count down to 126. However, the increased use of LDS quickly rises up to be the limiting factor for kernel occupancy. The best occupancy rate we can get by balancing LDS and register is 12.5%.

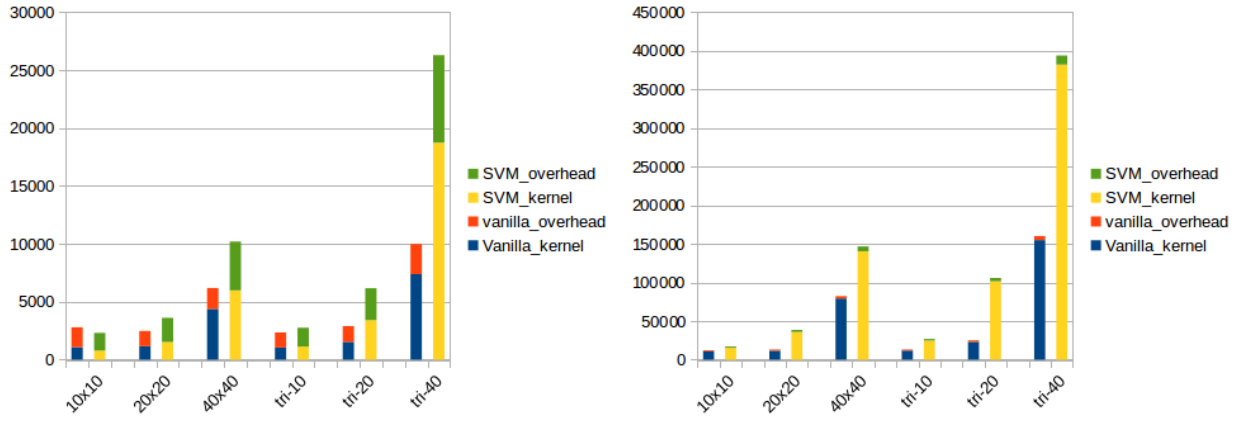
Optimizing kernel occupancy is an art of balancing the use of limited resources. With slight increase in occupancy, we did not observe remarkable change in kernel running time.

4.6 Analysis on SVM (Shared Virtual Memory)

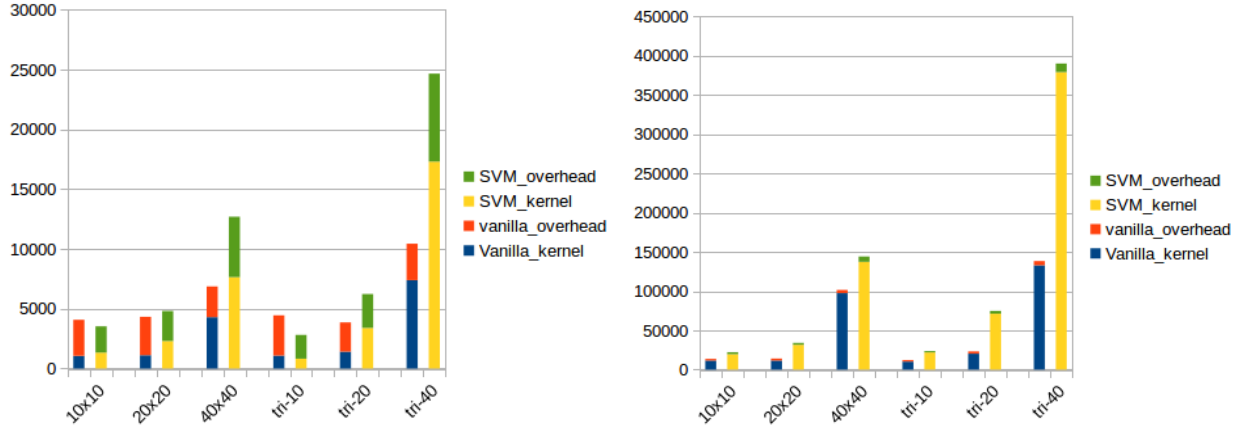
Shared Virtual Memory (SVM) is a landmark feature of OpenCL 2.0. Multiple works reported performance increase when using SVM. On the contrary, the kernel for DG-CVS runs longer when using SVM. In this section we first show the performance of DG-CVS face kernel after using SVM. Then we describe our micro benchmark to understand the reason of this performance hit.

Figure 4.3 shows performance comparison between vanilla GPU version and SVM version. The X-axis is execution time in micro seconds, while the Y-axis is test cases. After switching regular buffer to SVM on DG-CVS solver, total execution time increased significantly. There are two reasons. One is due to AMD's compiler redesign. The compiler for OpenCL 2.0 produces quite different assembly than for OpenCL 1.x. We compared the assembly code and execution time using different compiler flags to verify the result. The same kernel code runs 30% slower using *-cl-std=CL2.0* flag than using *-cl-std=CL1.2* flag. Hardware vendor needs to address this issue in future compiler design.

Another reason for the performance hit is due to using atomic on SVM. We developed



(a) SVM adv sin p1 and p3 cases.



(b) SVM burgers p1 and p3 cases.

Figure 4.3. Performance comparison between vanilla GPU version and SVM version.

microbenchmark to analyze the cost of atomic on SVM.

4.6.1 Microbenchmarking

In order to understand the reason why DG-CVS slows down after using SVM to replace OpenCL regular memory buffers, we developed a micro benchmark to test the speed of atomics on SVM.

In this test, we created an array of size 6144. We launch 6144 threads from GPU, with each thread performing an atomic operations on its corresponding array element. Each work group contains 64 threads. We test different operations on both regular memory buffer and SVM buffer, with the flags specified as below:

```
0. __kernel void atomic_test ( __global int * Array){
1.     int tid = get_global_id(0); // no op
2.     Array[tid] ++;              // add
3.     atomic_add(Array + tid, 1); //atomic_add
4.  }
```

Figure 4.4. Kernel code for SVM microbenchmark.

- Regular memory buffer is created with clCreateBuffer() command, using the flag CL_MEM_READ_WRITE and CL_MEM_USE_HOST_POINTER. First flag sets this buffer to grant threads read and write access. Second flag utilizes zero copy for this memory buffer. This configuration is denoted *REG_BUF*.

- SVM buffer is created with clSVMAlloc() command, with two options:

  - using the flag CL_MEM_READ_WRITE, CL_MEM_SVM_FINE_GRAIN_BUFFER. Second flag allocates fine grain memory buffer region, which is shared by both CPU and GPU. we denote this configuration as *SVM (FINE_GRAIN)*.

  - using the flag CL_MEM_READ_WRITE, CL_MEM_SVM_FINE_GRAIN_BUFFER, and CL_MEM_SVM_ATOMICS. Second flag and third flag are both needed for atomic operation to work on a SVM region because atomic is a feature added on fine grained sharing. This configuration is referred to as *SVM (ATOMIC)*.

Kernel code is shown in Figure 4.4. In a real world application, such access does not need atomic operation because we know for a fact that there is no concurrent access here since each thread operates on their own array element. But in order to test for the true cost for atomic operation on SVM, we eliminate the cost of contention. Contention is when multiple threads concurrently modify a single variable, they will compete for the ownership for this variable.

No op means the kernel only gets the global ID of each thread and returns. It only executes line 1 in Figure 4.4. *Add* is a regular increment on one element on Array. It executes

Operations	REG_BUF	SVM (FINE_GRAIN)	SVM (ATOMIC)
No op	3 us	3 us	2 us
add	11 us	15 us	22 us
atomic_add	14 us	x	180 us

Table 4.2. Time comparison of different operation on SVM and regular buffer.

line 1 and 2 in Figure 4.4. *Atomic_add* does line 1 and 3 in Figure 4.4. We test all operations on above mentioned memory spaces. Test result is shown in Table 4.2.

4.6.2 Experiment result and analysis

Kernel time for *no op* remains the same for all configurations. It does not access any global memory. For *add*, we observe a slight increase from REG_BUF to SVM (FINE_GRAIN), then to SVM (ATOMIC), because an *add* operation requires read and write to global memory space. The increase is due to the address translation overhead that occurs on SVM region. Maintaining coherence between CPU and GPU consumes memory bandwidth as well.

There is a significant jump for *atomic_add* from REG_BUF to SVM (ATOMIC). Atomic_add operation on SVM takes 12 times longer than in regular buffer. Finding out the reasons for this performance hit is a future research topic. We speculate that when atomic operation is done on REG_BUF, GPU can finish the atomic operation then write it to L2 cache, which is shared across all GPU cores. On the other hand, SVM region indicates that variables declared in this memory space could be modified by both CPU and GPU. An atomic modification will be cached to Last Lower Cache (LLC) which is visible to the whole platform. The overhead of going lower to the cache hierarchy contributes to the lengthened kernel time.

Our benchmark shows that performing atomic operation on SVM incurs significant overhead. It can become a serious bottleneck of an application if no attention is paid.

CHAPTER 5

CONCLUSION

In this thesis, we present the acceleration of DG-CVS using CPU-GPU heterogeneous processors. We first show the benefit of offloading computation intensive parts of DG-CVS to many core GPU processors by comparing performance of the program on serial processors and GPUs. Moreover, we developed further optimizations for GPU kernel after understanding the underlying hardware execution model. Our result shows that an improved thread mapping that reduces thread divergence further improves performance. We also address low kernel occupancy caused by register pressure and used multiple techniques to reduce register pressure. However, due to hardware resource limitations, kernel occupancy slightly increases and does not show impact on execution time. Lastly, we observe performance decrease after using SVM to enable collaboration between CPU and GPU. We develop micro benchmark to analyze the reason for such negative results.

This thesis demonstrates the benefit of using heterogeneous processor to accelerate computation intensive applications such as DG-CVS using OpenCL implementation. However, future research is needed to minimize the overhead of using SVM for the community to truly take advantage of the benefits that SVM brings to heterogeneous computing.

BIBLIOGRAPHY

BIBLIOGRAPHY

AMD (), App profiler kernel occupancy.

AMD (2012), Get started with codexl.

AMD (2015a), Amd opencl programming user guide.

AMD (2015b), Opencl optimization guide.

Brunie, N., S. Collange, and G. Diamos (2012), Simultaneous branch and warp interweaving for sustained gpu performance, in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 49–60, IEEE.

Chakroun, I., M. Mezmaz, N. Melab, and A. Bendjoudi (2013), Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm, *Concurrency and Computation: Practice and Experience*, 25(8), 1121–1136.

Chang, S.-C., and W.-M. To (1991), A new numerical framework for solving conservation laws: The method of space-time conservation element and solution element.

Fung, W. W., I. Sham, G. Yuan, and T. M. Aamodt (2007), Dynamic warp formation and scheduling for efficient gpu control flow, in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 407–420, IEEE Computer Society.

Gaster, B., L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa (2012), *Heterogeneous Computing with OpenCL: Revised OpenCL 1.*, Newnes.

Han, T. D., and T. S. Abdelrahman (2011), Reducing branch divergence in gpu programs, in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, p. 3, ACM.

Howes, L., and A. Munshi (2015), The opencl specifications, version 2.0.

Intel (2015), Opencl 2.0 shared virtual memory overview.

Junkins, S. (2015), Memory sharing and the compute architecture of intel processor graphics gen8.

Krakiwsky, S. E., L. E. Turner, and M. M. Okoniewski (2004), Acceleration of finite-difference time-domain (fdtd) using graphics processor units (gpu), in *Microwave Symposium Digest, 2004 IEEE MTT-S International*, vol. 2, pp. 1033–1036, IEEE.

Mukherjee, S., X. Gong, L. Yu, C. McCardwell, Y. Ukidave, T. Dao, F. N. Paravecino, and D. Kaeli (2015), Exploring the features of opencl 2.0, in *Proceedings of the 3rd International Workshop on OpenCL*, p. 5, ACM.

Mukherjee, S., Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli (2016), A comprehensive performance analysis of hsa and opencl 2.0, in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pp. 183–193, IEEE.

Munshi, A. (2009), The opencl specification, version 1, in *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pp. 1–314, IEEE.

Narasiman, V., M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt (2011), Improving gpu performance via large warps and two-level warp scheduling, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308–317, ACM.

Pollefeys, M. (), Gpu optimization.

Robert Ioffe, M. S., Sonal Sharma (), Achieving performance with opencl 2.0 on intel processor graphics.

Rodrigues, C. I., D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu (2008), Gpu acceleration of cutoff pair potentials for molecular modeling applications, in *Proceedings of the 5th conference on Computing frontiers*, pp. 273–282, ACM.

Ta, T., K. Choo, E. Tan, B. Jang, and E. Choi (2015), Accelerating dynearthsol3d on tightly coupled cpu–gpu heterogeneous processors, *Computers & Geosciences*, 79, 27–37.

Ta, T., D. Troendle, X. Hu, and B. Jang (2017), Understanding the impact of fine-grained data sharing and thread communication on heterogeneous workload development, in *The 16th International Symposium on Parallel and Distributed Computing*.

Tu, S. (2013), Riemann-solver free space-time discontinuous galerkin method for magnetohydrodynamics, in *44th AIAA Plasmadynamics and Lasers Conference*, p. 2755.

Tu, S., G. W. Skelton, and Q. Pang (2012), Extension of the high-order space-time discontinuous galerkin cell vertex scheme to solve time dependent diffusion equations, *Communications in Computational Physics*, 11(05), 1503–1524.

Tu, S. Z. (2015), A riemann-solver free spacetime discontinuous galerkin method for general conservation laws, *American Journal of Computational Mathematics*, 5(02), 55.

Tubbs, K. R., and F. T.-C. Tsai (2011), Gpu accelerated lattice boltzmann model for shallow water flow and mass transport, *International Journal for Numerical Methods in Engineering*, 86(3), 316–334.

Vesely, J., A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee (2016), Observations and opportunities in architecting shared virtual memory for heterogeneous systems, in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pp. 161–171, IEEE.

Yoshitake, O., and O. Keiji (2015), Reducing thread divergence in gpu applications through memory partitioned streams, *77* , 3, 02.

Zhang, E. Z., Y. Jiang, Z. Guo, and X. Shen (2010), Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping, in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 115–126, ACM.

VITA

Xiaoqi Hu

https://www.linkedin.com/in/chelseahu

- **Key Strengths**

  Key experience in profiling and benchmarking program to find out bottleneck and fine tune performance accordingly. Knowledge on CPU GPU architecture. Able to conduct research and software development on heterogenous system. Develop and maintain software using version control tools. Talent for quickly learning. Exceptionally organized and strong team work skills.

- **Experience**

  * **Graduate Student Assistant** 05/2015 Present

  Profile and analyze program to find out computational intensive parts and offload them to GPU to accelerate program, Use profiler to find out bottleneck and apply optimization technique to fine tune the program. Implement multi-threaded lock-free algorithms. Research on memory system on heterogeneous processor.

- **Projects**

  * **Accelerating Shallow Water Flow Simulator** 03/2016 - Present

  The simulator runs 145% faster after porting and optimization on GPU. Performed detailed analysis on serial code to identify time consuming part and offload it to GPU. Come up with solutions to accommodate limitations of GPU programming. Thoroughly tested to guarantee correctness. Used open source profiler CodeXL to find out bottleneck of the GPU kernel. Applied architecture-aware optimization to further

speed up the simulator. Develop and maintain Drafted reports and presentations to update progress with collaborators. Documented development steps in detail.

* **Lock-Free Data Structure on APU** 10/2015 - 06/2016

Implemented lock-free pointer type data structures such as linked-list for experimenting workload sharing on heterogeneous processor. Experimented different configurations for best speed up option.

* **Hand Detection with Web-Cam** 05/2015 - 10/2015

Developed GPU-friendly hand detection algorithms using distance transform image. Hand detection success rate is 80.36% and gain 17.5 times speed up with GPU.

- **Skills**
  * Languages: C/C++, OpenCL 2.0, Java, Shell
  * Op. Systems: Linux, Mac
  * Tools: CodeXL, Git, gcc/gdb, Eclipse, Vim