ENHANCEMENTS TO LOOPING CONSTRUCTS BY PREFETCHING AND EXPLOITING GPU

By

ANURAG PESHNE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2018

To my parents.

ACKNOWLEDGMENTS

At the outset, I would like to thank my advisor Dr. Beverly Sanders for providing me the opportunity to work on this project and all the guidance throughout the course of my thesis work. She provided new ideas whenever I got stuck and helped me with finer points. And most importantly she introduced me to a whole new area of computation which I wouldn't have known to exist if I hadn't got a chance to work on this project.

I would like to thank Dr. Ajith Perera for helping me in the matters of computational chemistry. As a domain scientist, he helped me in understanding domain problems and benchmarking my experiments on real life jobs.

I am deeply grateful to my family for the constant support and my brother for encouraging me to pursue higher education at a foreign university. And, I would like to thank my friend, Snehal, for keeping me motivated and helping me in writing my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ENHANCEMENTS TO LOOPING CONSTRUCTS BY PREFETCHING AND EXPLOITING GPU

By

Anurag Peshne

August 2018

Chair: Beverly A. Sanders
Major: Computer & Information Science & Engineering

The Super Instruction Architecture (SIA) is a parallel programming environment engineered to work on large blocks of floating point numbers. Looping constructs `do` and `pardo` are one of the most important constructs in Super Instruction Assembly Language (SIAL) since they allow the programmer to work with an individual block. To improve the runtime performance of the looping constructs, two techniques are used: improving GPU utilization and prefetching blocks to hide network latency.

In the previous versions, the domain programmer was needed to manually transfer blocks between main memory and GPU memory as well as to mark parts of SIAL code to be executed on GPU. Copying data between GPU memory and main memory is costly and have a high impact on overall performance of GPU execution. Automatic memory management is provided and various techniques are implemented to reduce and in some cases eliminate GPU memory and main memory transfer cost.

There is a common code pattern in SIAL to request a block of data followed by computation on that block. This pattern makes inefficient use of network and compute resources, since one remains underutilized while the other is being used. Deterministic request for blocks has been exploited to implement prefetching to optimally utilize the network resources during the time computational resources have blocked the control flow.

Various experiments have been conducted to evaluate the effect of change of various parameters on GPU utilization, prefetching, `wait_time` and overall computation time.

CHAPTER 1
INTRODUCTION

Super Instruction Architecture (SIA) is a parallel programming environment originally designed for problems in computational chemistry involving complicated expressions defined in terms of tensors. Tensors are represented by multidimensional arrays which tend to be very large in a typical computation chemistry calculation. SIA consists of a domain-specific programming language, Super Instruction Assembly Language (SIAL), and its runtime system, Super Instruction Processor (SIP). An important feature of SIAL is that algorithms are expressed in terms of blocks or multidimensional arrays rather than individual floating point numbers. This thesis presents two ideas to enhance the looping constructs in SIAL.

1.1 Issues of Working with GPUs

Advanced Concepts in Electronic Structure (ACES) is an implementation of SIA for chemical computation. In ACESIII, the previous version of ACES, the programmer had to deal with explicitly managing the memory transfer between GPU and main memory, and marking regions that are well suited for execution on a GPU. In ACES4, memory transfer is managed automatically by the runtime. This is implemented by tracking changes to blocks by every compute device.

Still, there is a need for the SIAL programmer to decide which portion of the SIAL code is suited for execution on GPU. Transferring data between GPU and main memory is expensive[20], and can dominate the overall time spent in computation. Hence, marking which block of code is suitable for execution on GPU is not a trivial decision. Time taken for transfering data depends on various factors such as the size of the blocks involved and the super instructions in the code-block, marked for execution on GPU. Larger blocks need more time to transfer between GPU and main memory, but at the same time, the speed improvement obtained by execution on GPU grows exponentially with the size of the block. Lastly, same SIAL programs are used to calculate different results by supplying different data files. It is possible that, for the same program, the overall time required for execution on GPU

be greater than CPU for some size of data and less than CPU for another size of data. This may happen if the speed gains by executing on GPU cannot compensate for the transfer time.

1.2 Issues in Transfer of Data from Servers

In SIAL, looping constructs are the only way to work with individual blocks. The typical workflow of working with large arrays in SIAL includes requesting a block of the larger array from a server, processing the block, computing resulting block and sending it back to a server. Though the operation of requesting block is nonblocking, subsequent operations which operate on the block, will wait until Message Passing Interface (MPI) transfer is completed. To reduce the overall cost of network transfer, prefetching of blocks has been implemented, which does the transfer concurrently with the block processing. By prefetching the blocks that are anticipated in the next iteration of the loop, the wait time for a block to be ready can be reduced and in some cases completely eliminated.

1.3 Organization of the Thesis

Chapter 2 presents literature study of the work done in use of GPU for general purpose computing, efficient GPU memory transfer, and the technique of prefetching to hide latency in accessing the data. Then a background of this thesis, including architecture and implementation of SIA and ACES4, is presented in chapter 3. Chapter 4 describes the technique of block prefetching implemented in SIA. Chapter 5 presents the optimizations done for efficient GPU memory transfer. Chapter 6 states the results of the benchmarks and experiments. And finally, chapter 7 concludes this thesis by presenting the conclusion and scope of future works.

CHAPTER 2
RELATED WORK

We discuss work done by others on General Purpose computing on Graphics Processing Units (GPGPU) as well as using GPU specifically for evaluating scientific calculations and using prefetching to hide latency in accessing data.

2.1 GPGPU

This section discusses previous work done on techniques such as porting existing code to GPU using directives, using CUDA directly to speed up scientific calculations and techniques to efficiently transfer data to GPU memory.

2.1.1 Directive Based GPU programming

There are several attempts made in the area of directive-based programming models. These attempts include OpenACC [27], OpenMP for accelerator [3], and less updated attempts including OpenMPC [17] and hiCuda [13]. The approach taken by these models is to make an implementation GPU-ready by placing directives which make execution of loops parallel.

This approach, however, is not suitable for SIAL, which works with blocks rather than individual floating point number. SIAL has its own parallel looping construct which distributes the loop iterations among participating nodes. Each participating node, having acess to GPU, can execute super instructions using GPU. These super instructions, can make use of above models to make an existing CPU implementation GPU-ready by inserting suitable directives or by having a completely rewritten low-level CUDA implementation.

2.1.2 GPU in Computational Chemistry

There are implementations of coupled cluster methods in computational chemistry on GPUs reported [4][11][18]. In these implementations, a specific algorithm was selected and then implemented on a GPU in a highly optimized form. These implementations have hardware specific optimizations, and are not generic enough to remain effective as new hardware development takes place.

One of the implementations of contraction operators on a GPU by Ma et al. [19] generates CUDA code to directly implement the contractions by optimizing for the particular order of indices in the contractions. This, as compared to using permutations and then two-dimensional matrix-matrix multiplication as implemented in SIA, should achieve better performance for a contraction. These specialized CUDA contraction implementations were added to NWChem, a computational chemistry package, and considerable speedups are reported for CCSD(T) calculation on CPU/GPU hybrid systems. Such optimized operators can be incorporated into SIA and be provided as built-in super instruction for contraction.

2.1.3 GPU Programming in Other High-Level Programming Languages

Several attempts for support for GPU in general purpose high-level programming languages have been made. A few examples include PyCUDA [16], jCUDA [39] and Chapel [34]. These toolkits give an interface to GPU from high-level programming languages such as Python and Java but programmer still needs to deal with low-level GPU bookkeeping such as memory management and defining kernels. As described in section 2.1.4, this is similar to ACESIII, wherein the programmer had to manually transfer memory between GPU and CPU. However, SIAL is a special purpose language and SIA runtime has information about block sizes and block operations. GPU implementation for some of the common block operations can be defined into the language itself. With these implementations and the information about block, the runtime can jump between GPU and CPU implementation transparently to the domain programmer along with taking care of memory management.

2.1.4 GPU in Previous Versions of ACES

There has been work done previously to exploit GPU in SIA by Jindal et al [14] in ACESIII. This implementation required the programmer to mark a block of code to be executed on GPU using directives `gpu_on` and `gpu_off` and manually manage memory transfer to and from GPU. This puts a burden of managing memory and recognizing suitable suitable block of code for execution on GPU on the domain programmer. If a block of SIAL code that is not suitable for execution on GPU is marked, it can result in suboptimal use of resources.

Executing operations on small blocks on GPU can be slower than executing on CPU due to time spent in memory transfer and insignificant gain in speed by execution. It can be a difficult decision for the SIAL programmer to decide if a block is large enough so that benefit of executing it on GPU becomes significant.

2.1.5 Optimizing GPU Memory Transfer

2.1.5.1 Host Memory Transfer

GPU cannot work directly with pageable host memory. One of the key condition for effectively exploiting GPU is to optimally manage data transfer between host memory and GPU memory. This is true for any computation running on GPU and not just High Performance Computing computations. Work done by Fatica [12], in accelerating Linpack describes various ways for optimizing memory transfers. This implementation intercepts calls to `DGEMM` and `DTRSM` and executes them simultaneously on both GPU and CPU cores. Memory allocated on host is pageable by default and GPU cannot access it. To achieve faster memory transfer, the implementation uses fast *PCI-E* transfers by page locking or *pinning* the host memory. Using PCI-E transfer the speed improvement of 5 GB/s from 2 GB/s is reported. Kaldewey et al [15] describes similar technique to optimize memory transfer using common address space for CPU and GPU, called as Unified Virtual Addressing (UVA). This helps memory transfers in two ways: it allows GPU to work with memory greater than available physical GPU memory and also relieves the programmer of the burden of managing two memory spaces. UVA requires that the host memory is page locked. Using UVA, sustained rate of 6.2 GB/s is reported. Host memory page locking is a common technique in both of the works, which enables GPU DMA controller to access main memory at PCI-E speeds. This technique can be used to speed up the SIA block transfers. This can be incorporated into the automatic memory synchronization so that it is transparent to the domain programmer.

2.1.5.2 Network Transfer

GPUs are often deployed in a cluster to accelerate computationally intensive general-purpose tasks. Still, for communication the GPUs need assistance from the CPUs. This requires

intermediate copies from the GPU memory to the host memory. To have a way around this inefficiency, there has been work done to make GPUDirect Remote Direct Memory Access (RDMA) available for InfiniBand clusters. Shainer et al [33] introduces GPUDirect that enables GPUs to transfer data via InfiniBand without involvement of the CPU or need of any buffer copies. Using GPUDirect RDMA, third-party devices such as network adapters can access the GPU memory buffer directly, over the PCI-E bus. Potluri et al [32] evaluated the GPUDirect RDMA for InfiniBand. They reported improvement in latency of MPI Send/MPI Recv by 69% and 32% for message size of 4Byte and 128KByte and improvement in uni-directional bandwidth achieved using 4KByte and 64KByte messages by 2x and 35%, respectively. Although, GPUs are not allocated to the SIA servers, GPUDirect feature can still be exploited to bypass the worker CPUs and to push the GPU memory buffer directly to the network adapter. This can also be used to completely avoid host memory by collecting the SIA block in GPU memory, operating on it using GPU and sending it back, bypassing CPU. This can help in reducing host and GPU memory transfers.

2.2 Prefetching

In this section we look at work done in general idea of prefetching such as nonblocking fetch operation, techniques used to determine optimal prefetching parameters such as the number of blocks to prefetch and prefetch cache size.

Prefetching is an old technique [1][36][37] used to get data up in memory hierarchy before it is actually needed by the processor. The main aim behind prefetching is to hide the data access latency due to the difference in speed to access data. A variety of approaches, including hardware prefetching, and caching, compiler techniques, pre-execution and runtime execution, have been implemented in high performance computing.

Hardware prefetching techniques, which includes transferring separate cache blocks [35], were implemented much before software techniques were implemented. Porterfield [31] introduced the idea of software prefetching using special instruction to preload values into the cache without blocking the computation. Using this instruction, the compiler can inform

the cache over 100 cycles before a load is required. Asynchronous MPI message can be used to request the server for the next block without blocking the control flow. The runtime can request for blocks several iteration before the blocks are required.

Prefetching is also exploited in the area of disk IO apart from feeding processor cache from memory. Patterson et al [30] implemented informed prefetching mechanism for IO intensive application to exploit highly parallel IO systems. The mechanism depends on disclosure of future access dynamically. In SIA, the runtime has information about how the array is going to be accessed by workers. Using this information it can predict accurately which blocks should be prefetched. It can also predict the order of blocks needed.

Dahlgren and Stenstrom [10] developed an algorithm to determine the number of memory blocks to prefetch. They proposed *adaptive sequential prefetching* policy, which allows the number of blocks to prefetch, K, to vary during runtime of the program to reflect spatial locality shown by the program. K is varied by calculating *prefetch efficiency* metric which is defined as the ratio of useful prefetched blocks to the total number of prefetched blocks. If the prefetch efficiency exceeds an upper threshold then K is incremented and decremented if it drops below a lower threshold. The value of K can reach 0, effectively disabling prefetching. Although, the SIA runtime can predict accurately the useful blocks, varying the number of blocks to prefetch, dynamically, can help control the network traffic. Due to several prefetching requests the network might get congested, resulting in delayed transfer of blocks. Workers can vary the number of blocks to prefetch by sensing delay in transfer of blocks.

There has been work done by Bhatia et al [5] to determine the size of the cache, to maximize hit rate in a sequential prefetching scheme. In this work, an online algorithm is devised which saves the blocks evicted from prefetch cache into another cache, *evicted cache*. If the incoming request is satisfied by the evicted cache, according to the algorithm the size of prefetch cache is too small and it increases it. If the request hits prefetch cache or there is a miss on both of caches then the algorithm leaves the size unchanged. To determine if the cache size is too large, the eviction cache is observed and if eviction cache receives no hits then

the size of the cache is decremented. In SIA, once a iteration of a loop ends, then worker can safely evict the block from its memory, since in the next iteration, change of indices guarantees a different block.

In SIA, blocks which are needed to be prefetched can be predicted with more accuracy than memory blocks. Further, the exact sequence in which blocks are needed is also known. This makes prefetching in SIA free of few of the problems faced in general memory block prefetching discussed in above section. However, this cannot be used to prefetch a large of number of blocks at once, since it might cause network congestion. Similar to the problems faced in general memory prefetching, it is difficult to predict how many blocks to prefetch. But once the block is prefetched and computed on, the block can be safely evicted from memory.

CHAPTER 3
BACKGROUND: SUPER INSTRUCTION ARCHITECTURE

This chapter introduces the SIA, ACES4, block-based programming, the design of workers and servers, SIAL, parallel looping constructs and design of GPU implementation.

3.1 SIA

The SIA is a special purpose, domain agnostic, parallel programming framework which is engineered for solving very large computation. SIA expresses the computation in terms of multi-dimensional arrays and instructions, which operate on complete arrays. This way of expressing a computation is different from conventional programming languages wherein computation is described in terms of individual floating point number and operations which work on these individual numbers. Aggregating individual numbers into blocks results in better performance and higher utilization of resources. However, this adds considerable complexity to programs since now the programmer has to be careful of the index arithmetic and designing algorithms to loop over entire blocks. Algorithms in SIA can be directly expressed in terms of blocks. This gives the performance benefits of using block and at the same time relieves programmer from error-prone work of dealing with indices and looping. Expressing computation in terms of blocks has multiple performance advantages: moving data in terms of blocks is more efficient since it has less overhead per individual number and runtime can overlap computation and network operation since transferring these blocks will take significant time, resulting in better overall utilization of resources. There is an added advantage of expressing computation in terms of blocks in a parallel framework, that the work can be distributed among participating nodes in a natural way.

Since these multi-dimensional arrays can be too large to hold in physical memory of a single processor, they are broken into blocks or super numbers. These super numbers are input to special instructions written to operate on blocks instead of floating point numbers. These instructions are called as super instructions. Section 3.2 describes how these blocks are managed among participating processors.

SIA can execute instructions on blocks in parallel on multiple processors. To facilitate movement of blocks among parallel execution units, server-client architecture is used. SIA provides SIAL, a block-oriented domain specific language (DSL), which supports expressing algorithms using blocks and a way to write super instructions, which are domain specific, in an optimized way. The following sections describe details of working of server-client architecture, SIAL, and constructs in it.

3.2 Architecture of SIA

SIA can execute instructions in parallel over multiple processors. It can be deployed and scaled on multiple nodes in a high performance computing cluster using MPI for internode communication. Since the multidimensional arrays involved in the calculations can be extremely large for the memory of a single processor, SIA divides the arrays into chunks of manageable sizes. These chunks are distributed to different processors and the processors can work on these chunks concurrently.

SIA supports arrays of size greater than the combined memory of all processors involved in computation by providing the facility of storing the chunks which are not *hot*, that is the chunks which are not going to be used soon, on larger, although slower, memory on hard drives. This swapping of blocks is automatic and transparent to the programmer. To facilitate the movement of data among processing units and swapping out blocks to hard drives, SIA divides available processors into two groups: workers and servers — the workers are responsible for actual execution of instructions on the blocks, while the servers make sure blocks are served to and from the workers. The division of the number of the servers versus the number of the workers is chosen by SIA but can be overridden by passing command line arguments.

3.3 SIAL

SIAL is a programming language provided for expressing problems of the target domain. The idea behind SIAL is to keep the domain problem separate from platform problem. SIAL programs are written by the domain experts whereas the intricacies involved in execution of

SIAL programs, like distribution of data, parallel execution, memory management, runtime optimizations, are handled by computing experts.

SIAL, apart from providing programmers with conventional constructs such as conditional constructs, looping constructs, procedures, way to import other SIAL programs like general purpose programming language, also provides special parallel looping construct and a way to define domain-specific block operation or *super instruction*. It has several common block operations built in. The parallel looping construct, `pardo`, loops over multiple indices and distributes blocks to different processors. This construct is of special interest to us since the optimizations done using GPUs are mostly done in the interpretation of this looping construct.

As mentioned above, domain experts can write their own domain-specific super instructions which take in single or multiple blocks and output a resulting block. These instructions can be written in C, C++ or Fortran. Since these languages are much closer to the hardware, these super instructions can be written in a very optimized way. Further, this facility can be exploited to port the super instructions to other computing devices such as GPU by writing these super instructions using Nvidia CUDA.

3.3.1 SIAL Interpreter

SIA consists of SIAL compiler which translates human-readable SIAL text to machine friendly bytecode. This bytecode is interpreted by SIAL interpreter. Since this interpretation happens at runtime, the interpreter is able to optimize the execution based on resources available at runtime. If interpreter finds GPU accessible, then it may execute some part of the SIAL program on GPU and if it doesn't find it, then it can automatically fallback to CPU. Similarly, there are various optimizations implemented which depends on the amount of physical memory present on the processor.

3.4 Block Structure

As described above, instead of processing data as a single floating point number, SIA processes blocks of floating point numbers. These blocks are chunks of even larger multidimensional arrays. Blocks are represented inside SIAL interpreter using `Block` class. SIA

supports heterogeneous computing using other computing devices such as GPUs. Since GPUs have their own device memory which is separate from main processor memory, there is a facility in `Block` class to represent the block memory in other computing devices. Along with member attributes which represent block metadata and member functions which act on the block, the Block class has pointers to the memory location on each computation unit: CPU, GPU and support for more computing device such as Intel Xeon Phi.

The `Block` class depending upon the active computing device will return the appropriate device memory address. There is also a logic build into the `Block` class to automatically synchronize memory for various devices, so that if one device edits the block and then in next instruction, another device wants to read the block, the block memory will be automatically synchronized and the next device will read updated memory. This is done by maintaining version numbers for each memory and then updating the memory based upon the version numbers when it is accessed.

3.5 Executing Super Instructions on GPU

There are two ways in which GPUs are exploited in SIA to obtain high concurrency. First, the super instructions can be written in CUDA. Using CUDA, these instructions can make use of low-level hardware features and domain knowledge to fine-tune the implementation. Secondly, some of the general purpose block operations such as matrix multiplication, addition, scaling, tensor contraction can be implemented for GPU in the interpreter itself. These operations can be imported from highly optimized libraries such as Nvidia CuBLAS.

The SIAL interpreter takes care of executing GPU or CPU implementation of an operation based upon availability of implementations and other factors such as the size of input and output.

3.6 Overview of the ACES

ACES4 is an implementation of SIA for chemical computation. It has been executed on a variety of architectures but is specially optimized to enable calculations on leadership-class supercomputers. Using ACES4, computational scientists try to find approximate solutions to

Schrodingers equation using Coupled cluster methods. There exist other methods to calculate approximate solutions but Coupled cluster methods, although expensive computationally, are one of the most accurate methods. The chemical computation done using ACES4 uses data of high dimensions. A typical calculation in this domain takes as input the geometry of a molecular system and a choice of single particle orbitals as the basis to expand the many-electron quantum-mechanical wave function. The complex algorithms, which produce properties of the molecular system, can easily require arrays of double precision floating point of size several hundred Gigabytes. Of these arrays, at least three need rapid access and are usually stored in RAM, the rest that are used less frequently can be stored on disk. SIA architecture is very suitable for these kinds of calculations since the workers can work on parts of array concurrently and the servers can swap out less frequently used arrays to disk.

CHAPTER 4
BLOCK PREFETCHING

This chapter presents the idea of prefetching data block from a server to hide the network transfer time.

4.1 Background

To process a large array, which cannot fit into the memory of a single node, a typical workflow in SIAL consists of requesting a block of an array from a server in a `pardo` looping construct by each participating worker. After processing it, the resulting block is sent back to a server. This common pattern can be summarized as single or multiple network bound operation surrounding one or more compute bound operations.

Algorithm 4.1. *Processing large array*

1: **loop** ▷ *SIAL do/pardo loop*
2: $GET\ A[i,j]$ ▷ *Request data from a server asynchronously*
3: $GET\ B[j,k]$ ▷ *Network bound*
4: $t\_result[i,k] \leftarrow A[i,j] \times B[j,k]$ ▷ *Wait for A, B to be ready*
5: $s\_result \leftarrow A[i,j] \times A[i,j]$ ▷ *No need to wait for A*
6: $CALL\ compute\_fun(t\_result[j,k])$ ▷ *Compute bound*
7: $PUT\ AB[i,k] \leftarrow t\_result[i,k]$ ▷ *Network bound*
8: **end loop**

It is clear from the pseudocode that the computing resources are wasted while waiting for the data to be ready. To improve the wait time of the compute operation, non-blocking MPI call `MPI_Irecv` was exploited to prefetch the blocks from servers over the network.

4.2 Implementation of Prefetching

4.2.1 pardo Loop Implementation

To determine which block should be prefetched, the runtime needs to predict the next block in the loop. This depends on what type of loop is being executed. While the `do` loop which iterates over the indices one by one in SIAL is simple, there are multiple implementations of `pardo` loop, which differ in the distribution of indices and thus the distribution of blocks over workers:

- `SequentialPardoLoop`: it behaves similar to a simple do loop, except this loop, can loop over multiple indices.
- `StaticTaskAllocParallelPardoLoop`: the indices for this loop are determined statically by distributing the block over workers in a cyclic fashion.
- `BalancedTaskAllocParallelPardoLoop`: to support symmetric arrays, SIAL has `where` construct in loops which prunes the iteration based on some programmer-defined condition. Due to such pruning, there is a non zero probability that all of the iterations are assigned to one particular worker. This loop evaluates the `where` clauses and distributes the valid iteration over workers in a balanced way.
- `FragLoopManager` and its subclasses: SIAL supports large sparse arrays. To loop over them efficiently SIAL has various implementations of fragmented pardo loops. These loops have knowledge of the internal structure of the sparse arrays and thus can skip over rows and columns having no useful values.

Due to so many varieties of implementation of `pardo` looping construct and to support future implementations of indices generation schemes, it is important to keep the mechanism of prefetching separate from indices generation. For this, a lazy prefetching mechanism was implemented which will probe for indices as needed, dynamically. A lazy implementation would also give freedom to vary the number of prefetched blocks at runtime.

4.2.2 Lazy Indices Probing

Each class implementing `pardo` have a function `update` which calculates the values of the set of indices and populates the interpreter state. This state is used by interpreter to calculate blocks using an array and index values.

Algorithm 4.2. *update_indices*() → *bool*

```
1:  procedure UPDATE_INDICES
2:      for all indices in loop do
3:          old_index_val ← interpreter_state.indices[index_slot]      ▷ get current index value
4:          new_val ← old_index_val + 1                                ▷ Increment the index as needed
5:          if new_val ≥ upper_bound[index_slot] then
6:              new_val ← lower_seg[index_slot]
7:          end if
8:          interpreter_state.indices[index_slot] ← new_val            ▷ update the interpreter state
9:      end for
10:     if all indices reached upper_bound then
11:         return false
12:     else
```

```
13:         return true
14:     end if
15: end procedure
```

To implement lazy probing, the work done by procedure `update` is divided into multiple procedures:

- `get_next_indices` produces set of *next* indices purely based on indices passed as a parameter rather than getting directly from interpreter state. This allows us to produce series of indices independent of the state of the interpreter.
  Algorithm 4.3. *get_next_indices([index]) → [index]*

```
1: function GET_NEXT_INDICES(current_indices)
2:     for all index_id in loop do
3:         old_index_val ← current_indices[index_id]          ▷ get current index value
4:         new_val ← old_index_val + 1                        ▷ Increment the index as needed
5:         new_indices ← new_val                              ▷ update the index into new set of indices
6:     end for
7:     return new_indices                                     ▷ return new set independent of interpreter state
8: end function
```

- `peek_indices` returns set of indices and internally takes care of maintaining and creating a list of indices *lazily*. It calls the procedure `get_next_indices` to produce next set of indices by passing the last set of indices in the list as needed. It increases the length of the list by 1 if the set of indices requested for is the last one on the list and there are more indices in the loop.
  Algorithm 4.4. *peek_indices(IndexList::iterator) → [index], IndexList::iterator*

```
1: function PEEK_INDICES(it)
2:     if IndexList.empty() then
3:         return [ ]
4:     else
5:         peekedIndices ← ∗it
6:         if next(it) == IndexList.end() then
7:             new_indices ← get_next_indices(∗it)
8:             IndexList.insert_after(it, new_indices)
9:         end if
10:        return peekedIndices, it
11:    end if
12: end function
```

- `prefetch_indices` remember the last set of indices returned to each `GET` statement and returns the next set of indices when it is called. It remembers that by mapping the

position of indices in the list to `line numbers` of each `GET`. This makes varying the number of prefetched blocks for each `GET` possible.

Algorithm 4.5. *prefetch_indices() → [index]*

```
1:  function PREFETCH_INDICES
2:      line_number ← Interpreter.current_line_number()
3:      if prefetch_map.contains(line_number) then
4:          it ← prefetch_map[line_number]
5:      else
6:          it ← prefetch_map.begin()
7:      end if
8:      {prefetched_indices, it} ← peek_indices(it)
9:      prefetch_map[line_number] ← it
10:     return prefetch_indices
11: end function
```

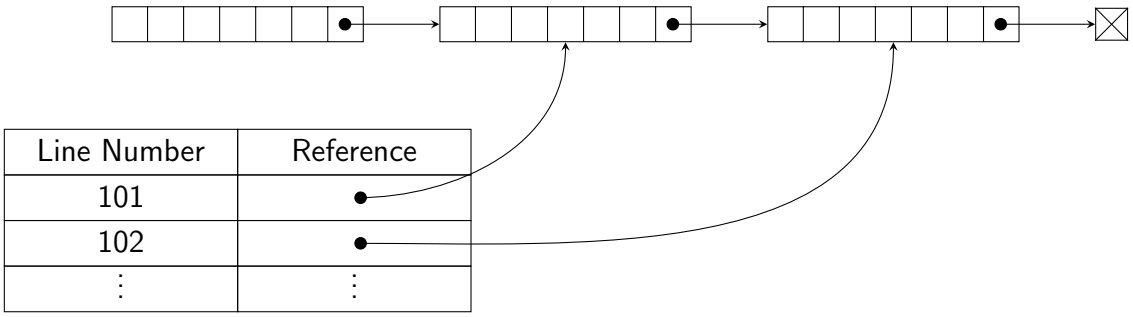

Figure 4-1. `prefetch_indices` saves mapping of line number to position in the list

- `update` is now changed to simply pop the first set of indices from the list and update interpreter state so that other modules can find the value of current indices. This decreases the length of the list by 1.

  Algorithm 4.6. *update_indices() → bool*

```
1:  function UPDATE_INDICES
2:      current_indices ← peek_indices(IndexList.begin())
3:      if length(current_indices) > 0 then          ▷ is there any more iteration
4:          Indexlist.pop()
5:          for all index_slot in current_indices do
6:              interpreter_state.indices[index_slot] ← current_indices[index_slot]
7:          end for
8:          return true
9:      else
10:         return false
11:     end if
12: end function
```

In all, the functions `peek_indices` and `update` can be modeled as producer and consumer problem on a bounded buffer. And function `prefetch` indices is free to point at any set of indices on the list.

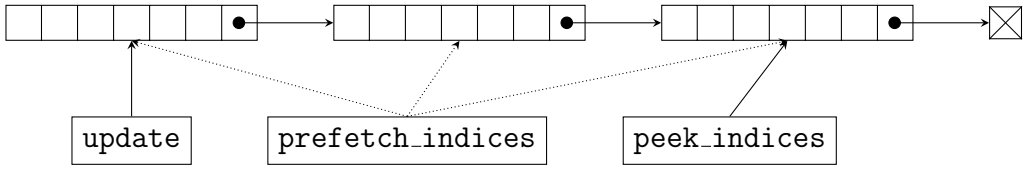

Figure 4-2. `update` consumes, `peek_indices` produces if needed, `prefetch_indices` is free to move along list

CHAPTER 5
EXPLOITING GPU

5.1 Background

`do` and `pardo` looping constructs are one of the most important constructs in SIAL since they allow SIAL programmers to express operations on blocks. Using `pardo` construct the calculations can be done in parallel. SIAL runtime is responsible for the distribution of work over all the worker nodes. In this chapter, we present the problems faced in using GPU to execute the looping constructs and how they were solved.

5.1.1 Attempts in ACESIII

There have been attempts[14] made in the previous version of ACES to use GPU to speed up computation. In this work, the SIAL programmer had to deal with a lot of low-level GPU memory operations such as allocating memory on GPU, copying blocks to and from GPU and deallocating memory on GPU. Since not all calculations are suitable to be executed on GPU, the SIAL programmer had to mark regions of SIAL code suitable for execution on GPU.

Listing 5.1. Code Fragment from ACESIII for CCSD calculation

```
1  #start of GPU region
2  gpu_begin
3  #allocate and initialize blocks on GPU
4  gpu_put aoint(lambda,mu,sigma,nu) #allocate and copy data from CPU
5  DO i1
6  DO j1
7      gpu_put LT2AOab1(mu,i1,nu,j1)
8      gpu_put LT2AOab2(nu,j1,mu,i1)
9      gpu_put LTAOab(lambda,i1,sigma,j1)
10 ENDDO j1
11 ENDDO i1
12 gpu begin
13 DO i
```

```
DO j
    #perform computations on GPU
    Yab(mu,i,nu,j) = 0.0
    Y1ab(nu,j,mu,i) = 0.0
    gpu_allocate Yab(mu,i,nu,j) # allocate temp blocks on GPU
    gpu_allocate Y1ab(nu , j ,mu, i )
    #contraction Y1ab(nu,j,mu,i) = Yab(mu,i,nu,j) #permutation
    Yab(mu,i,nu,j) = aoint(lambda,mu,sigma,nu)*LTAOab(lambda,i,sigma,j)
    LT2AOab1(mu,i,nu,j) += Yab(mu,i,nu,j) #elementwise sums
    LT2AOab2(nu,j,mu,i) += Y1ab(nu,j,mu,i) #elementwise sums
    gpu_free Yab(mu,i,nu,j) #free temp blocks on GPU
    gpu_free Y1ab(nu,j,mu,i)
ENDDO j
ENDDO i
#copy results to CPU , free blocks on GPU
DO i1
DO j1
    gpu_get LT2AOab1(mu,i1,nu,j1)
    gpu_get LT2AOab2(nu,j1,mu,i1)
    gpu_free LT2AOab1(mu,i1,nu,j1)
    gpu_free LT2AOab2(nu,j1,mu,i1)
    gpu_free LTAOab(lambda,i1,sigma,j1)
ENDDO j1
ENDDO i1
gpu_free aoint(lambda,mu,sigma,nu)
gpu_end
#end of GPU region
```

A code fragment from ACESIII for CCSD calculation is presented in 5.1. In line 2 and 39, the region is marked to be executed on GPU and blocks of lines 5-11 and 29-37 deal with managing memory to and from GPU and main memory. The actual calculations are done by lines 13-27.

5.2 Runtime Memory Management

To manage the block memory and to automate the memory transfer between GPU and CPU, metadata about the state of memory is stored in the interpreter. The `Block` class which represents the SIA *block* in the interpreter was modified to now store this metadata and pointer to the memory location in GPU and main memory in form of an object of another class `Device_Info`.

Due to this additional layer, supporting multiple compute devices is now possible. The metadata includes flags for block data being *dirty* and *valid*. It also includes a field, **version number**, which is incremented each time the block is changed by the compute device. This helps in keeping memory of different compute devices synchronized.

Listing 5.2. `Device_Info` Class structure

```
1 class DeviceInfo {
2   double* data_ptr_;
3   unsigned int data_version_;
4   bool isDirty; // is block data modified/dirty
5   bool isAsync; // are there any pending operations on device
6 }
```

The code shown in 5.2 is one of the ways of expressing the class `Device_Info` in C++ language.

Using this metadata, specifically `data_version_`, the runtime can keep track of changes to data by different devices and find the device with the latest data by comparing version numbers:

Algorithm 5.1. *Block::get_latest_device()* $\rightarrow$ *Device_Info*

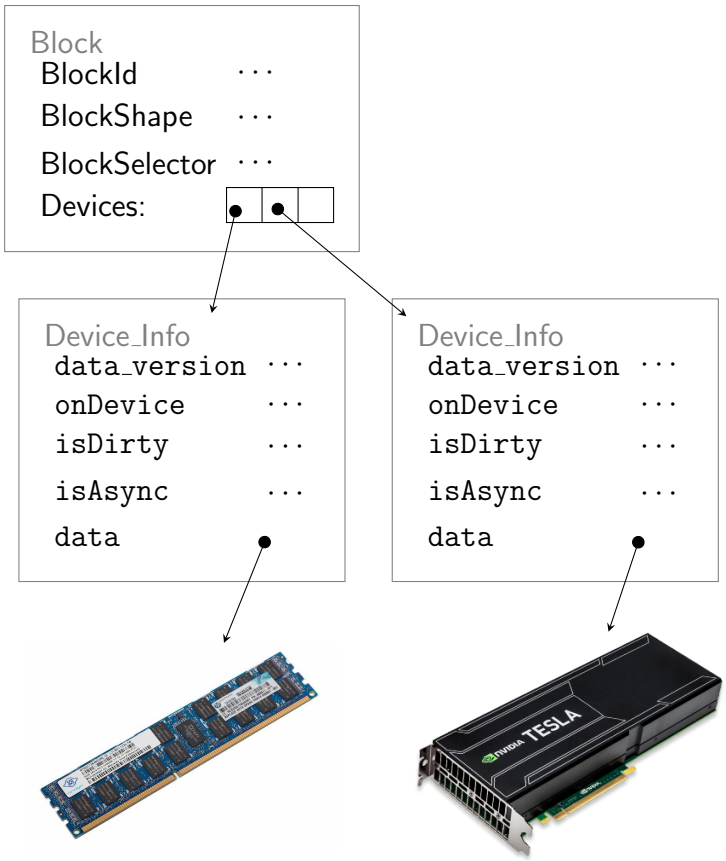

Figure 5-1. `Block` and `Device_Info` structure

```
1:  function BLOCK::GET_LATEST_DEVICE
2:      highest_version_found ← 0
3:      highest_version_device ← null
4:      for all device_info in this.devices do
5:          if device_info.data_version > highest_version_found then
6:              highest_version_found ← info.data_version
7:              highest_version_device ← device_info
8:          end if
9:      end for
10:     return highest_version_device
11: end function
```

After determining the device with latest version of data, the runtime can update other devices, if needed. The only interfacing function exposed outside of `Block` class is `get_data(deviceid)` which returns a pointer to the memory location on device identified

by `deviceid`. The logic to always return the latest version of data can be embedded into `get_data(device)`:

Algorithm 5.2. *Block::get_data(deviceid)* $\rightarrow$ *double**

```
1: function BLOCK::GET_DATA(deviceid)
2:     latest_device ← this.get_latest_device()
3:     if latest_device.data_version > this.devices[deviceid].data_version then
4:         memcpy(latest_device.data, this.devices[deviceid].data)
5:         this.devices[deviceid].data_version ← latest_device.data_version
6:     end if
7:     return this.devices[deviceid].data
8: end function
```

5.3 Optimizing Block Copying

An enormous cost has to be paid [2][8] for transferring memory to GPU as compared to the time taken for actual computation on GPU. Hence, to have significant speed gains by executing on GPU over CPU, it is necessary to minimize the time spent on block transfers. This has been achieved by deploying multiple optimizations that are explained in this section.

5.3.1 Reducing Block Transfers

The first optimization includes saving unnecessary block synchronizations. The SIAL runtime has the information about the intent of each block request, whether the block is being requested to be read, updated or written. Using this information the number of synchronizations between devices can be reduced. Blocks which are going to be written need not be synchronized, however, blocks requested for being read or updated need to be synchronized for the requested device.

The algorithm presented in 5.2 can be split into `get_data` and an explicit `update_data`:

Algorithm 5.3. *Block::get_data(deviceid)* $\rightarrow$ *double**

```
1: function BLOCK::GET_DATA(deviceid)
2:     return this.devices[deviceid].data
3: end function
4:
5: function BLOCK::UPDATE_DATA(deviceid)
6:     latest_device ← this.get_latest_device()
7:     if latest_device.data_version > this.devices[deviceid].data_version then
8:         memcpy(latest_device.data, this.devices[deviceid].data)
```

```
9:          this.devices[deviceid].data_version ← latest_device.data_version
10:     end if
11:     return this
12: end function
```

With this change, the runtime can decide whether synchronizing block contents is necessary. If the synchronization is needed, it can call `block->update_data(device)->get_data(device)` which is the case with block read and update. If no synchronization is needed, as in case of a block being written, the runtime can simply get the pointer on device memory by calling `block->get_data(device)`. This change in the way of accessing and updating block data along with the information about the intent of the request can reduce several unnecessary synchronizations.

5.3.2 Memory Pinning

The memory allocated on CPU or main memory is pageable by default. GPUs cannot access data on such pageable host memory [20] [25] directly. For this reason, when a memory copy operation is requested from a host to a GPU, the CUDA driver first allocates a temporary **page locked**, or *pinned*, memory on the host (main memory), copies the host memory to the temporary pinned memory and then finally transfers the data from pinned memory to GPU device memory.

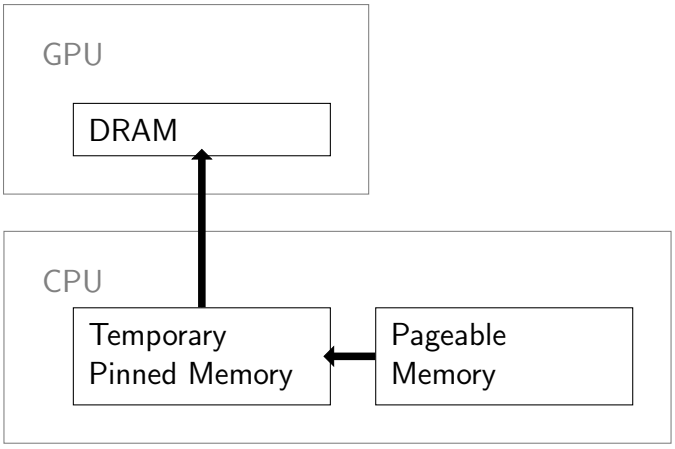

Figure 5-2. `memcpy` w/o memory pinning

Due to this copying of data to a temporary page-locked memory, extra time is spent in the redundant copy operation. To overcome this, page locked memory can directly be allocated on the host memory. For this CUDA gives API `cudaMallocHost()`, `cudaHostAlloc()` to allocate memory and `cudaFreeHost()` to deallocate the memory. Using this technique the data flow presented in 5-2 is modified to flow presented in 5-3.

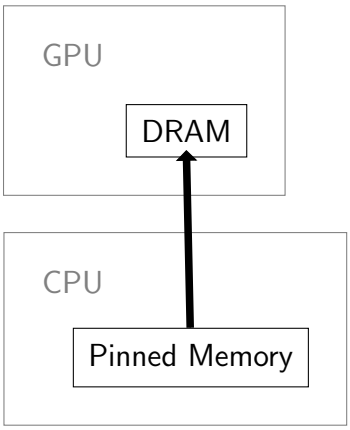

Figure 5-3. `memcpy` with memory pinning

5.3.2.1 Page Locked Memory Bandwidth

Apart from saving a redundant copy operation, memory page locking also helps in reducing time to copy because when memory is page locked, the GPU can invoke Direct Memory Access (DMA) controller [26][29][23][24] to transfer memory, bypassing the CPU. It is reported[15] that, using this, GPU can read main memory at PCI-E speeds.

5.3.2.2 Asynchronous `memcpy`

Since CUDA controller can invoke DMA controller to copy memory if the host memory is page locked, this operation can be carried out asynchronously not only to host but also asynchronously to GPU kernel execution engine. Thus both host and GPU can carry on with the execution of calculation while the blocks are synchronized. This is elaborated in section 5.3.4.1.

5.3.3 Memory Pinning Overhead

It was observed that while pinned memory made memory transfer faster, it carried a large overhead as compared to the allocation of non-pinned memory. Similar observations by Boyer et al [9][7][6] have been reported. They reported memory allocation call `cudaMalloc` to be 30-40% more expensive than `malloc` and `cudaHostMalloc` to be slower than `malloc` by a factor of 100. To overcome this, a caching scheme was developed to reuse the pinned memory pages.

5.3.3.1 Reusing Pinned Memory Pages

The problem of caching page locked memory pages is similar to the problem of dynamic allocation of memory, which is handled by Operating System (OS). Extensive work [38] is done in solving the problems involved in efficiently allocating memory dynamically. The issues involve dealing with external and internal fragmentation while minimizing the time taken to find a suitable block of memory. Caching and serving page locked memory blocks is similar to dynamic memory allocation, it needs to find suitable size of memory block such that least space in form of internal fragmentation is wasted. But the memory blocks cannot be split or coalesced, like in dynamic allocation.

5.3.3.2 Implementation of Caching Mechanism

When a page locked block of memory is marked for deallocation, instead of un-pinning it and calling `delete[]` on it, it is saved in a list. Two maps are used for bookkeeping. One of it, `free_list`, presented in figure 5-4, maps the block size to a list of pointers. When a request for a block of memory is received, this list is searched for a block of size equal to or greater than the size of block requested. If no such block is found then a new block is created, pinned and returned. It is possible that there is not enough free memory in the system to allocate a new block, in that case, few blocks of smaller sizes are unpinned and returned to OS. After there is enough free memory, a new, bigger, block is created. If there are no blocks in `free_list` and requested block size is greater than free memory, then the system is out of memory and exception is thrown.

The other map serves as a reverse lookup, presented in figure 5-5, from pointer to size of the block. This reverse lookup is essential, when a block is returned to be deallocated. The reverse lookup helps to know the actual size of the block since when a block is requested, the caching mechanism can return a block of greater size. Hence, the size of the requested block cannot be used to calculate the size of the allocated block to populate `free_list`.

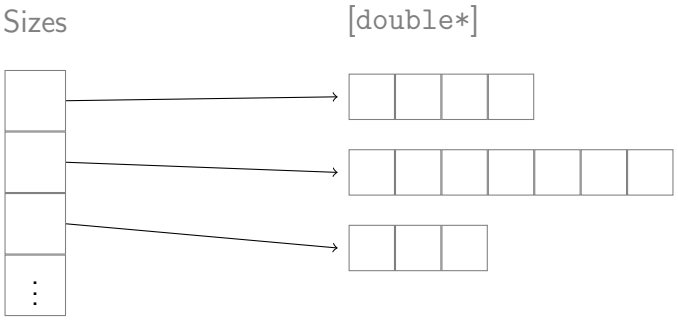

Figure 5-4. Structure of `free_list` map used for mapping block size to list of free blocks.

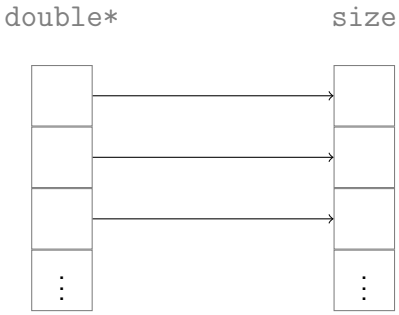

Figure 5-5. Structure of `reverse_lookup` map used for mapping back blocks to actual size of block.

Algorithm 5.4. *Page Locked Block Allocation*

```
1:  function ALLOCATE(size)
2:      cached_block_list ← free_list.get_greater_or_equal(size)
3:      if cached_block_list.length == 0 then
4:          if allocated_bytes + size × sizeof(double) > total_memory then
5:              freed_elements ← 0
6:              while freed_elements < size do
7:                  if free_list.empty() then
8:                      throw" outofmemory"
9:                  end if
10:                 for all block in free_list.top()− > second do
```

```
11:                     freed_elements+ = free_list.top()− > first
12:                     cudaHostUnregister(block)
13:                     delete[]block
14:                 end for
15:             end while
16:         end if
17:         new_block ← newdouble[size]
18:         cudaHostRegister(new_block, size)
19:         allocated_bytes+ = size × sizeof(double)
20:         reverse_lookup[new_block] ← size
21:         return new_block
22:     else
23:         cached_block ← cached_block_list.pop()
24:         return cached_block
25:     end if
26: end function
27:
28: function FREE(block)
29:     actual_size ← reverse_lookup[block]
30:     free_list[actual_size].append(block)
31: end function
```

5.3.4 GPU Streams

A *stream* in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host [21]. Streams can be considered as pipes of operations in which the operations get evaluated in First In First Out (FIFO) fashion. Operations in different streams can be interleaved and CUDA driver might execute them concurrently.

5.3.4.1 Non Blocking Device Synchronization

GPU streams were used to implement non blocking SIA block synchronization. Using multiple CUDA streams and page locked memory, DMA controllers can be invoked to transfer memory from host to GPU, and vice-versa, asynchronously to both CPU and GPU kernel execution engine. Current generation of GPUs have two DMA controllers [23][24] and one kernel execution engine. This means that the GPU can handle 2 asynchronous memory copying operation concurrently. These hardware features have been exploited by implementing asynchronous memory copying for block synchronization.

To support non blocking synchronization, during runtime, multiple streams are created and stored in the module `gpu_super_instructions`. This is the same module that initializes and maintains the GPU state. The number of streams created is configurable and currently is set to 2 since current generation of GPU devices have 2 DMA controllers. Each `Device_Info` object stores `gpu_stream_id` which is set when a copy operation on `Block` object is initiated. At the same time, the bit `isAsync` is set. This bit denotes that the `Block` object has pending asynchronous operations and before de-referencing the memory pointer, runtime should wait for the operation to finish by using CUDA call `cudaStreamSynchronize`.

Algorithm 5.5. *Asynchronous Block Synchronization*

```
1:  function GPU_MEMCPY(src_device, dest_device, num_elements)
2:      next_stream ← get_next_stream()
3:      cudaMemcpyAsync(src_device.data, dest_device.data, num_elements, next_stream)
4:      dest_device.stream_id ← next_stream
5:      dest_device.isAsync ← true
6:  end function
7:
8:  function BLOCK::GET_DATA(deviceid)
9:      if this.devices[deviceid].isAsync then
10:         cudaStreamSynchronize(this.devices[deiceid].stream_id)
11:         this.devices[deviceid].isAsync ← false
12:     end if
13:     return this.devices[deviceid].data
14: end function
```

5.4 Optimizing GPU Internode Communication

SIAL makes it possible to work on extremely large arrays, by dividing the array into multiple blocks. Servers distribute these blocks to workers and workers return resulting blocks back to servers. Workers and Server nodes communicate using highly optimized MPI library. Since GPU have their own memory, using GPU for computation in a multinode environment can add an additional layer of communication between workers and servers. In this section, optimizations for this communication are presented.

5.4.1 Background

Since GPUs have their own memory on which they compute, when a block is needed to be transfered to another node from GPU memory, it has to be copied to 3 different buffers: the block is first copied from GPU memory to temporary pinned memory, then to pageable memory and finally to pinned fabric buffer. This results in total 7 copy operations, including 3 similar operations on other side, and a network transfer operation. These operations are represented in Figure 5-6 by black arrows.

5.4.2 MPI Transfers Using DMA

Section 5.3.2 discusses memory pinning, which can be used for optimizing memory transfer between GPU memory and host memory. If the host memory is pinned, then the GPU and the network fabric can share [22] the pinned buffer. Thus, two sets of copy operations can be saved by requesting page locked buffers. This transfer is represented in Figure 5-6 by blue arrows.

5.4.3 MPI Transfers Using RDMA

The intermediate copying operations to host memory can be avoided further by taking advantage of RDMA. RDMA is a technique using which the GPU can send buffers from GPU memory to network adapter, without staging through host memory. OpenMPI supports RDMA [28] and thus all the extra copy operations can be avoided. Further RDMA transfers work independent of the CPU and thus it not only saves extra copy operations, the transfer is done over PCI-E and is independent of the CPU memory bandwidth and the memory bus traffic congestion. This is conceptually represented in Figure 5-6 using orange arrows which denotes one transfer from source GPU memory to destination GPU memory.

However, servers in SIA are not allocated GPU since servers are not responsible for any heavy calculation on blocks. They manage transfer of blocks from workers and swap *inactive* blocks to disk. Hence, a more accurate representation of the use of RDMA in SIA is presented in Figure 5-7.

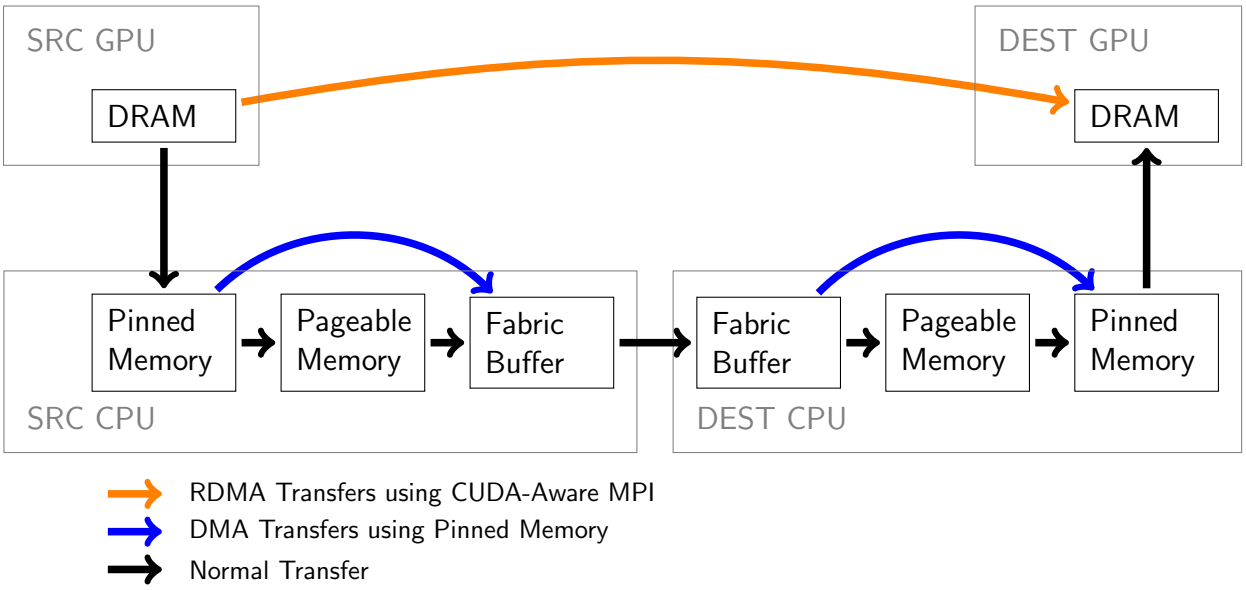

Figure 5-6. RDMA, DMA and normal transmission between two nodes with GPU

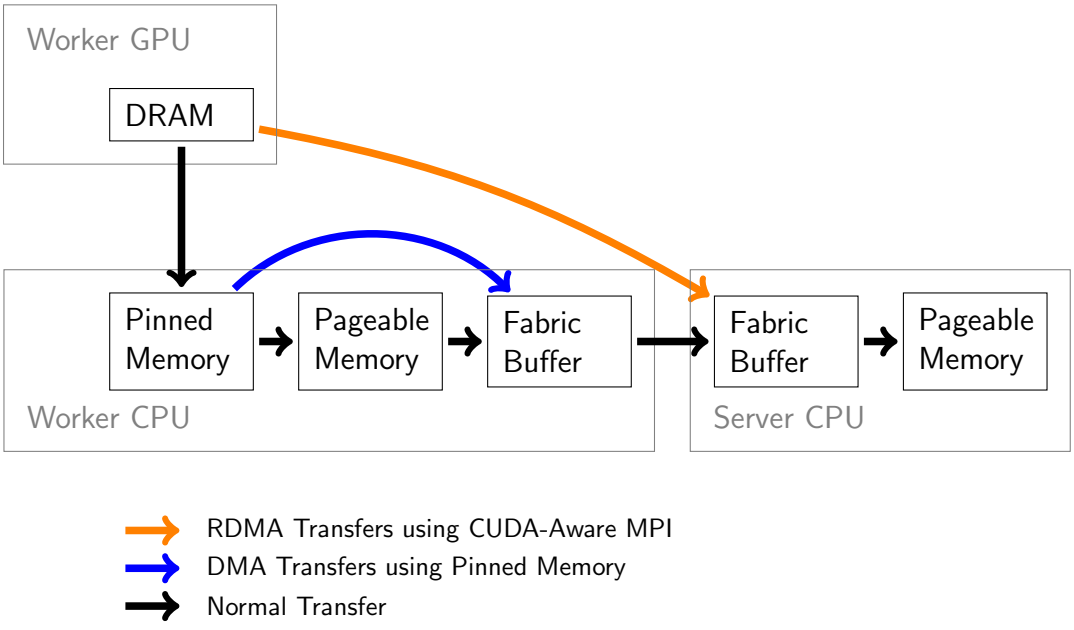

Figure 5-7. RDMA, DMA and normal transmission in SIA

CHAPTER 6
EXPERIMENTS AND RESULTS

This thesis presents a few of the possible performance optimizations for reducing the network latency while requesting for blocks from a server and transferring blocks from GPUs. This section describes series of experiments conducted by varying parameters such as block size, number of blocks to prefetch, number of servers, etc and its effects on the performance of the system.

6.1 Environment

These experiments were carried on HiperGator Computer at UF. Table 6-1 describes the specification of HiperGator 2. Table 6-2 explains the specifications of HiperGator 2 **compute** nodes and table 6-3 explains the specifications of HiperGator 2 **GPU** nodes. Table 6-4 describes the specification for the node interconnect in HiperGator 2.

Table 6-1. HiperGator 2 Spec Sheet

Name	Specifications
Total Cores	30,000
Memory	120 Terabytes
Storage	1 Petabytes
Max Speed	1,100 Teraflops

Table 6-2. HiperGator 2 **Compute** Node

Name	Specification
Manufacturer	Dell Technologies
Processor	Intel E5-2698v3
Base Processor Frequency	2.3 GHz
Sockets	2
Cores per socket	16
Thread(s) per core	1
Memory per node	128 Gigabytes
Memory Frequency	2133 MHz DDR4

6.2 Prefetching

This section presents several experiments conducted to investigate the optimal parameters and tradeoffs involved in the selection of parameters.

Table 6-3. HiperGator 2 **GPU** Node

Name	Specification
Manufacturer	Dell Technologies
Processor	Intel E5-2683
Base Processor Frequency	2.0 GHz
Sockets	2
Cores per socket	14
Thread(s) per core	1
GPU	Tesla K80
Memory per node	128 Gigabytes
Memory Frequency	2133 MHz DDR4

Table 6-4. HiperGator 2 Node interconnect specification

Name	Specification
Node Connection	Mellanox 56Gbit/s FDR InfiniBand interconnect
Core Switches	100 Gbit/s EDR InfiniBand standard

6.2.1 `hit_ratio`

To understand the performance of the prefetching mechanism, a new metric is introduced. Prefetch `hit_ratio` is defined as the ratio of the number of times the SIA runtime did not have to block for a certain data block to be ready and the total number of times the data block was accessed:

$$\texttt{hit\_ratio} = \frac{\textit{number of times no blocking required}}{\textit{total number of times data accessed}}$$

The `hit_ratio` represents the number of times prefetching was successful to hide network transfer cost. In the following experiments `hit_ratio` will be used to measure the effectiveness of parameters in prefetching.

6.2.2 Index Length

The length of indices is the length of the range of indices involved in the loop. The length of indices can have a high impact on prefetching. To study the relation between index length and prefetching, `hit_ratio` is observed by varying the range of indices. This is presented in figure 6-1.

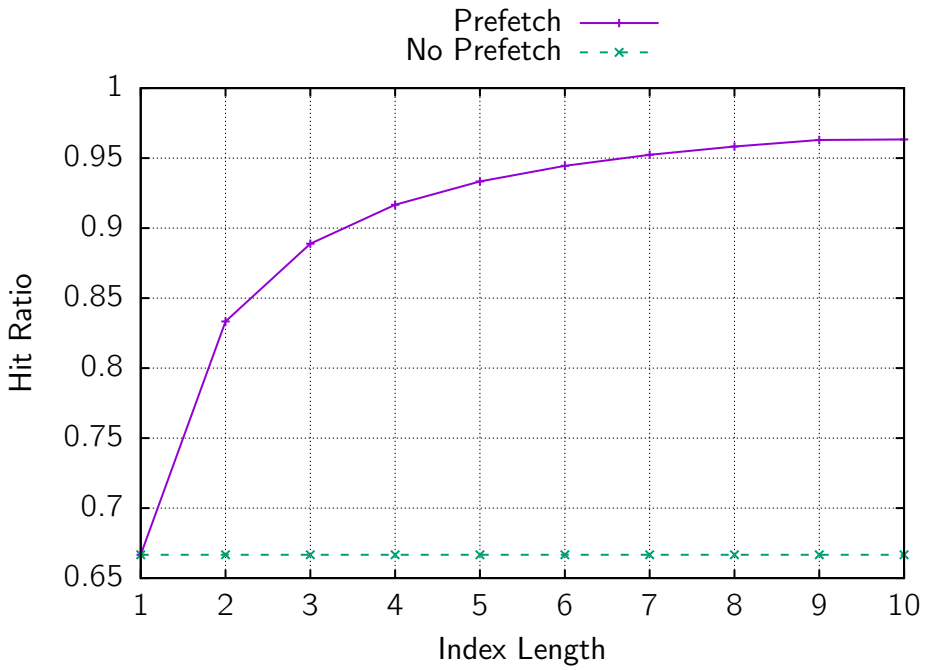

Figure 6-1. Index Range Length v/s `hit_ratio`

Note that the runtime has to block for data only the first time it accesses a block. Subsequent accesses do not need any blocking since the data is ready. Hence, the `hit_ratio` with no prefetching is non zero.

If an index spans only 1 then there is no scope for the runtime to do prefetching. This is evident from the plot when index length is 1, the `hit_ratio` with prefetching is equal to with no prefetching. As the length of range of index is increased, prefetching got working. This can be easily observed from exponential growth in `hit_ratio`. Eventually, the curve for `hit_ratio` flattens out after 6, since no significant improvement is achieved by increasing the index range length.

It is observed that as the runtime prefetches multiple blocks, the first request to the server takes longer as the number of index range increases. This side effect can be explained using the preceding observation about `hit_ratio`. Since the increase in index range length activates prefetching, network congestion increases and the first request to the server becomes costlier. This is presented in figure 6-2.

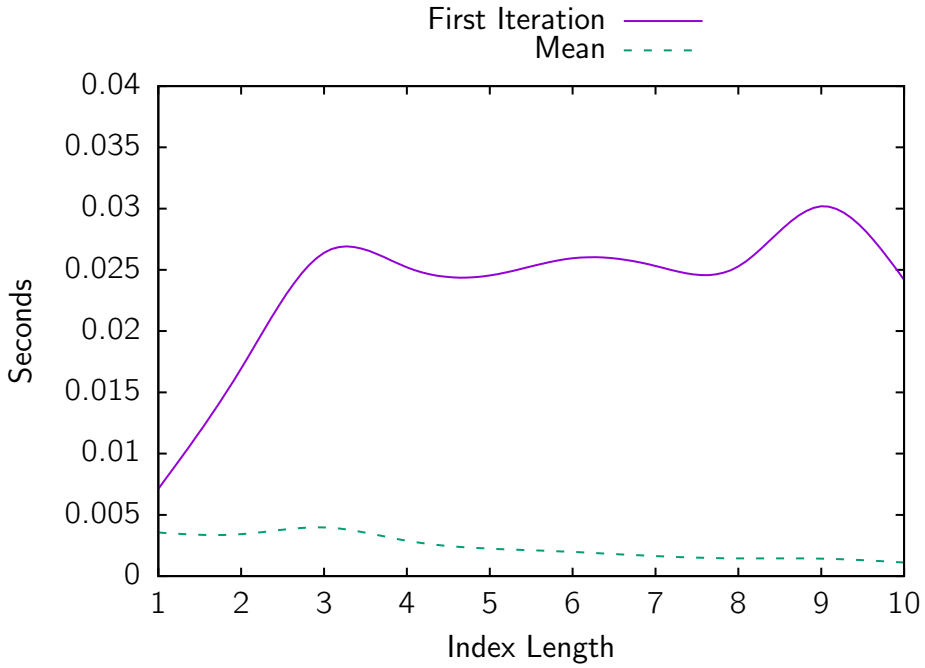

Figure 6-2. Index Range v/s `wait_time_` for first iteration

It can be concluded from previous observation that prefetching increases the wait time for the first request to the server. Thus, to compensate for the high cost of the first iteration by offsetting it in subsequent iterations, the length of the index range should be sufficient enough. The mean time taken per iteration is plotted against the length of index range in figure 6-3.

The length of an index should be greater than 5 to decrease the `wait_time_` by a factor of 2. The mean `wait_time` per iteration with prefetching can reduce up to 3 times, as compared to with no prefetching, if the length of index range is greater than 9.

6.2.3 Block Size

Since the time to transfer block over the network is proportional to the size of the block, the block size affects the first request made during prefetching. Along with the first request, multiple requests for prefetching subsequent blocks are made. This makes the `wait_time_` for first call sensitive to block size. This is evident from the graph plotting Block Size against mean `wait_time` for the first iteration in figure 6-4.

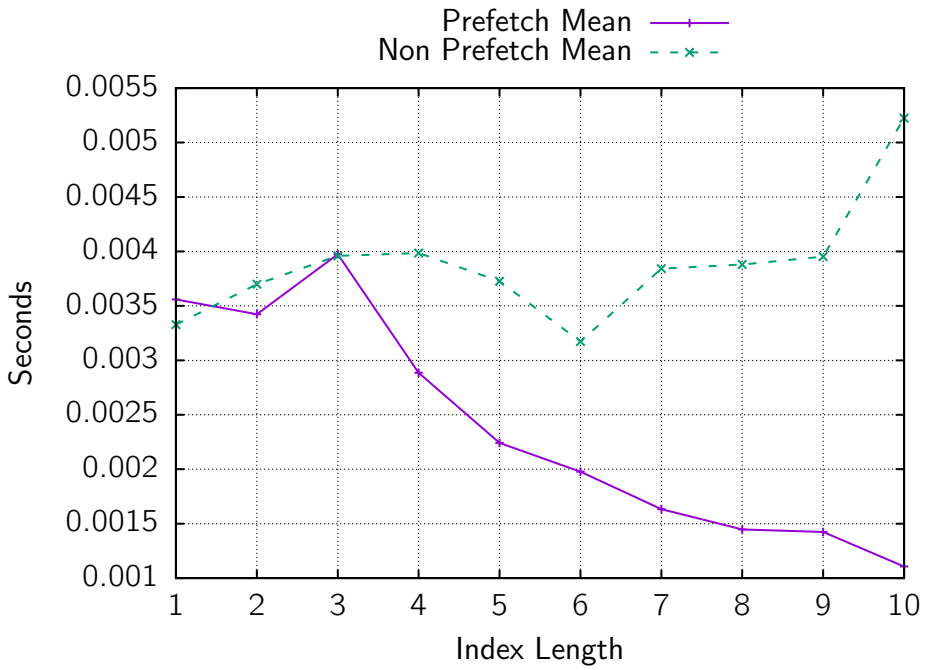

Figure 6-3. Index Range v/s `wait_time_` per iteration in Prefetched and no Prefetched Loop

For the first iteration, `wait_time_` in the case of loops with prefetching grows faster than loops without prefetching. At size of block greater than 500 elements, `wait_time_` for the first iteration with prefetching is almost twice the corresponding `wait_time_` without prefetching.

It seems that subsequent iterations are not affected by the block size in the loops with prefetching, since the runtime need not block for subsequent blocks. This results in reduction of overall mean `wait_time_`. This trend is presented in figure 6-5.

The mean `wait_time_` grows much slower for loops with prefetching as compared to loops without prefetching.

All of these trends of block size against first and mean `wait_time_` for loops with and without prefetching are summarized in figure 6-6

Although the `wait_time_` for the first iteration grows at the highest rate, prefetching compensates for it and keeps the mean `wait_time_` lower than loops without prefetching.

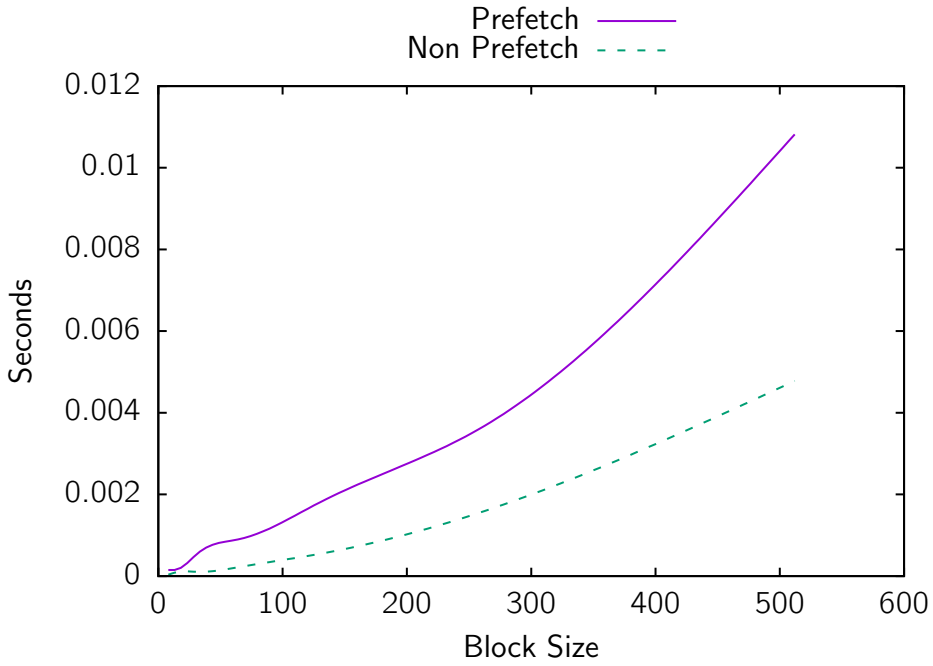

Figure 6-4. Block Size v/s `wait_time_` for first iteration

6.2.4 Number of Blocks to Prefetch

As stated in previous sections, prefetching affects the first request to the server and the size of the block also affects the first request. To observe the effect of the number of blocks to prefetch on the initial request, the number of blocks were varied and plotted against mean `wait_time_` for the first request. This is presented in figure 6-7.

It is clear from the plot that the mean `wait_time_` for the first request grows linearly with the number of blocks to prefetch. Thus the number of blocks to prefetch cannot be set at very high number unless the length of index range is known to be large.

To determine the effect of the number of blocks to prefetch to prefetching, the number of blocks to prefetch was varied and is plotted against the mean `wait_time_`. This plot is presented in figure 6-8.

For the case when the number of blocks prefetched is 0, which is in the case of no prefetching, the mean `wait_time_` is highest. It drops sharply as the number of blocks to

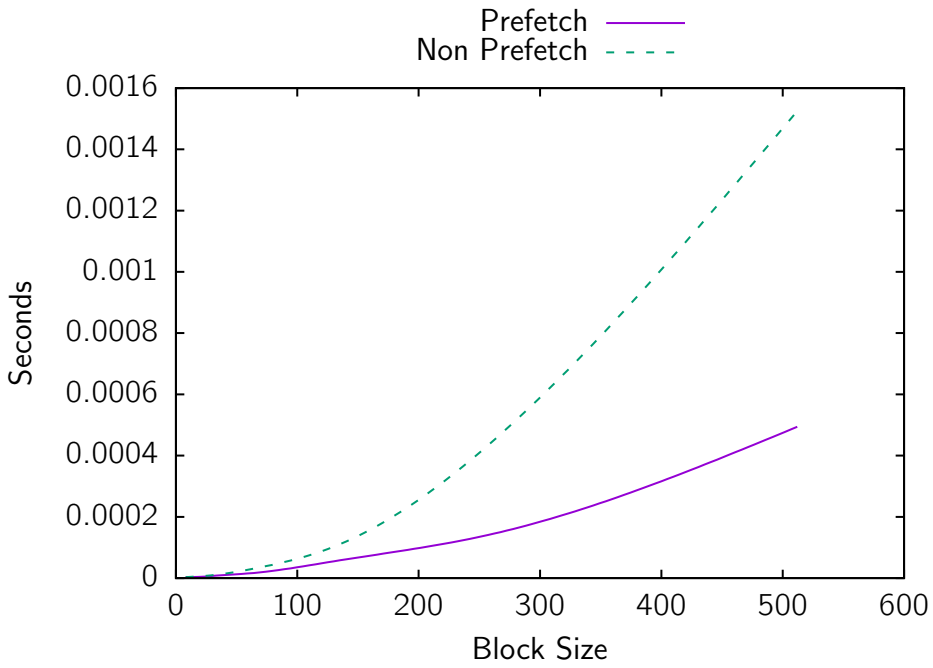

Figure 6-5. Block Size v/s Mean `wait_time_` for Prefetched and No Prefetch Loop

prefetch increases. It grows again, since with an increase in the number of blocks to prefetch, the first request to the server gets expensive.

As the number of blocks to prefetched is increased, more blocks are available for runtime without blocking. This is presented in figure 6-9 by plotting number of blocks to prefetch against `hit_ratio`.

Hit ratio saturates after hitting a critical amount. After that increase, the number of blocks to prefetch has no effect. This explains the rise in mean `wait_time_` as `wait_time_` for the first request grows and the number of blocks available without blocking stays constant.

6.2.5 $C_{12}H_{10}(BP)$ Molecule

To study the effect of prefetching on real-world application, a job from ACES4 on $C_{12}H_{10}$ molecule was executed. The total number of workers were 3 and number of servers were varied to study its effect on prefetching. This is plotted in figure 6-10.

With the increase in the number of servers, the overall runtime for programs with prefetching decreases. This is clear from figure 6-10, as the difference in the runtime of

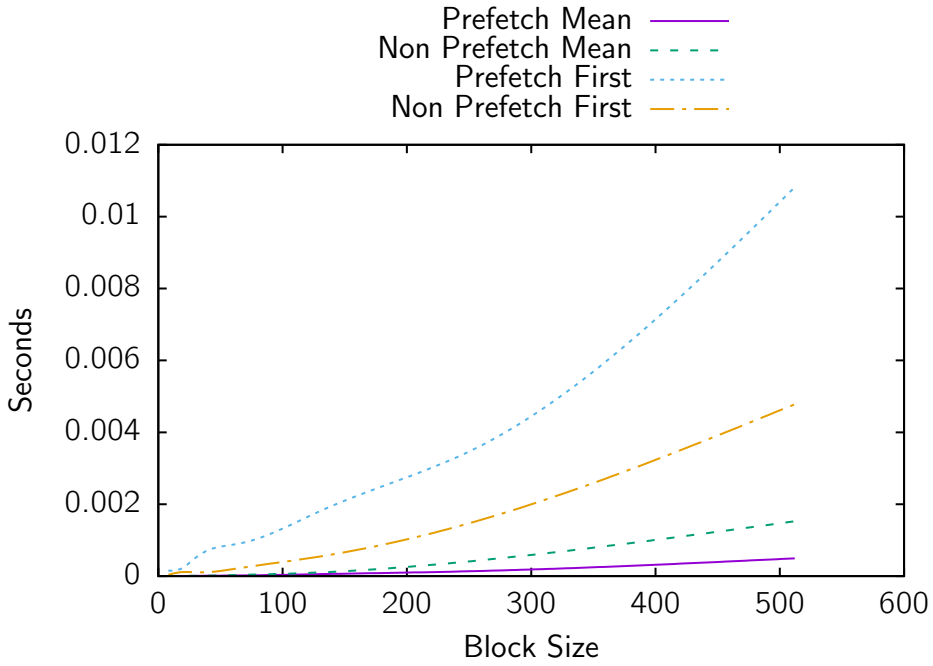

Figure 6-6. Block Size v/s Mean `wait_time_` for Prefetched and No Prefetched Loop

programs with and without prefetching increase. It is worth noticing that the only couple of programs show a significant increase in difference while one of the programs shows no difference in prefetching and no prefetching. To study why this is the case, the total `wait_time` for the programs is plotted in figure 6-11.

In the figure 6-11, along with `wait_time`, `barrier wait_time` is also plotted for all the programs involved. The `wait_time` of program `scf_rhf` is mostly due to `barrier wait_time` and rest two have considerably less `barrier wait_time` as compared to overall `wait_time`. This explains why only two of the 3 programs showed significant improvement by prefetching since prefetching helps in reducing `wait_time` for the block to be transferred over the network. The program `sch_rhf` did not have much `wait_time` for network transfer.

6.3 GPU

This section presents the experiments performed to benchmark the performance of using GPU.

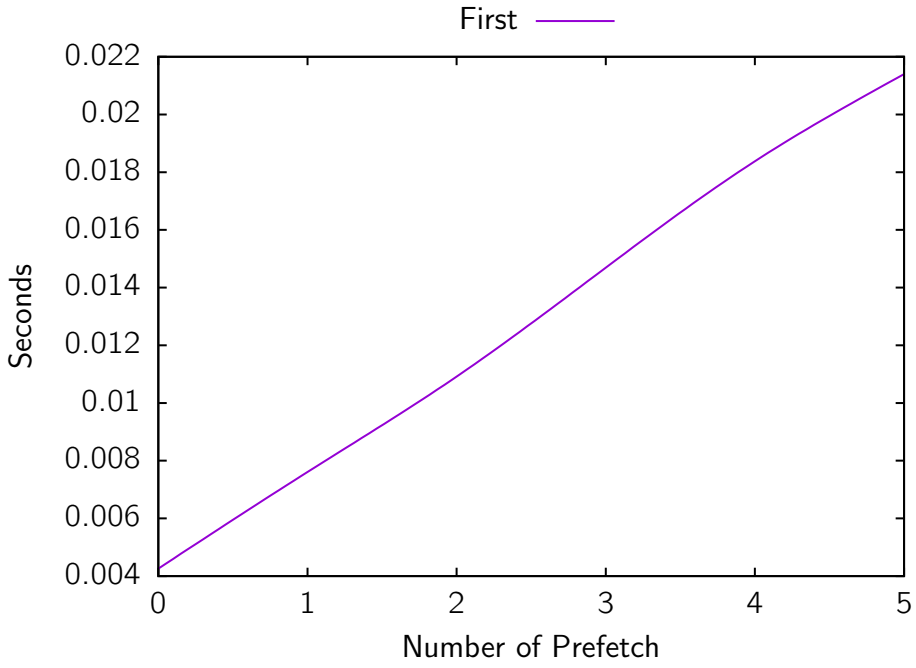

Figure 6-7. Number of Block Prefetched v/s `wait_time_` for the first request

6.3.1 Memory Pinning

Many of the optimization of involves page locking the host memory so that GPU can bypass processor and access the memory directly. This section presents various operations which exploit page locked memory.

6.3.1.1 Copy Speed

GPU can access only page-locked memory. Without explicit page-locked memory, the GPU driver first copies the memory to a temporary page locked memory and then copies to GPU buffer. By explicitly page locking memory, saving one copy operation is expected. Figure 6-12 presents varying block size against time to copy block.

It is evident from the plot and table 6-5 that explicit page locking only results in faster memory copy if the size of the block is above 1600 elements. After the size of block crosses a threshold, the pinned blocks are copied almost twice as fast as compared to non-pinned blocks.

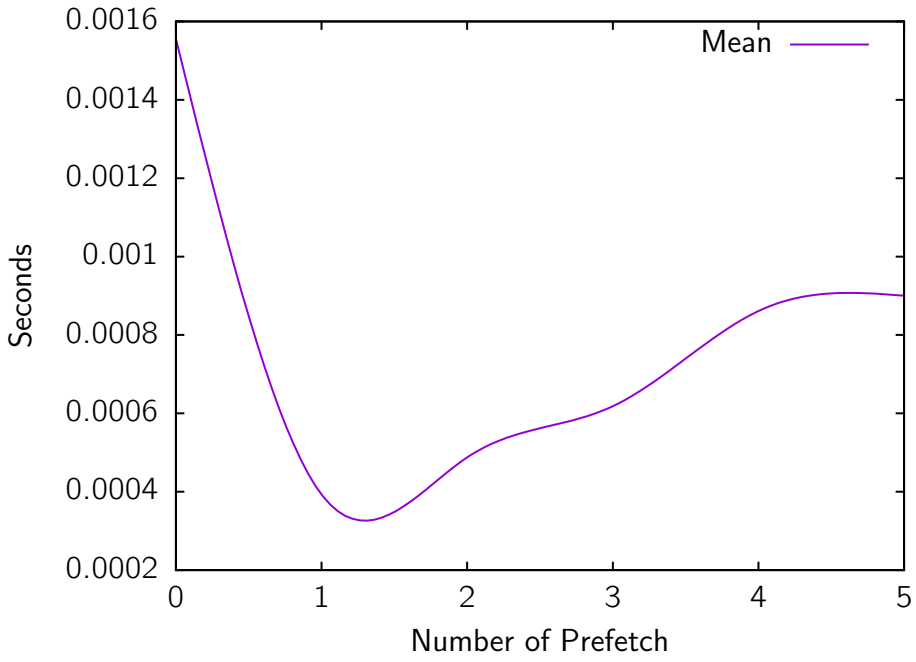

Figure 6-8. Number of Block Prefetched v/s mean `wait_time_`

6.3.2 Optimized Transfer

The block transfer optimization considers whether the block is requested with intention of being read or written. Hence, to benchmark this optimization, real-world test case is needed. Here, `rccsd_rhf.sialx` is used to investigate the number of transfers saved by exploiting the intent. This is presented in figure 6-13.

6.3.3 Memory Pinning Overhead

To benchmark the page lock memory allocation against non-page locked memory allocation such as allocated using `malloc` function call, two tests were done. The first test benchmarks allocation and second benchmarks deallocation. These two operations are presented in figure 6-14 and figure 6-15 by plotting time taken by page locked operations against non-page locked operations.

6.3.3.1 `alloc`

Table 6-6 describes the time taken for page locked memory allocation and non-page locked memory allocation.

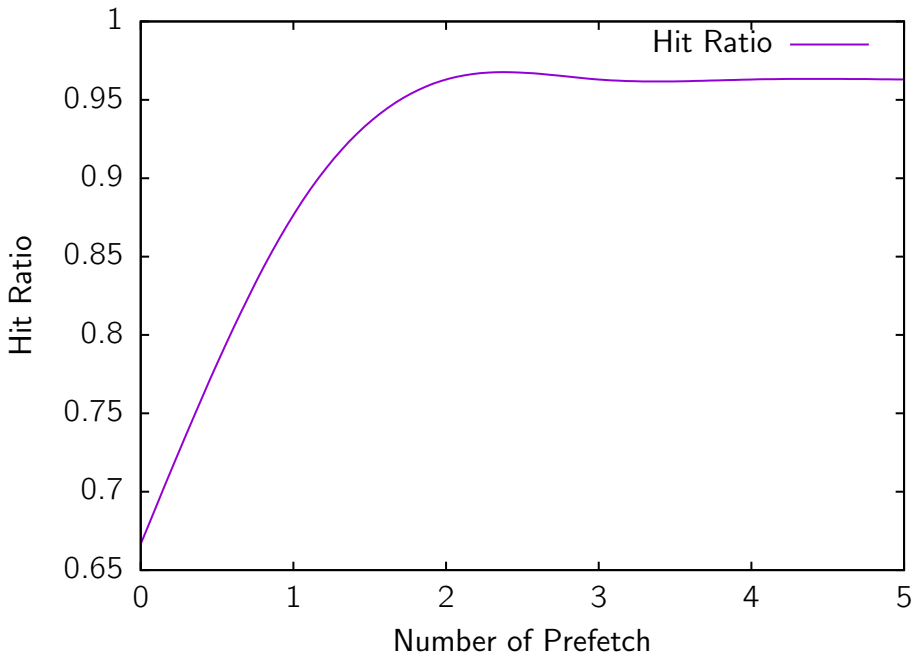

Figure 6-9. Number of Block Prefetched v/s Hit Ratio for the first request

It is clear from the table 6-6 and the figure 6-14 that the time required to allocate both page locked and non-page locked memory is independent of the size of memory requested. But the allocating page locked memory can be around 10 to 100 times costlier than non-page locked memory.

6.3.3.2 `free`

A similar test was performed to benchmark page locked and non-page locked memory deallocation. Table 6-7 describes the time taken by the deallocation operation against the varying size of the block.

The results are similar to page locked and non-page locked allocation, the time taken for page locked and non-page locked memory deallocation is independent of the size of the block. Page locked memory deallocation is around 100 times costlier than non-page locked memory deallocation.

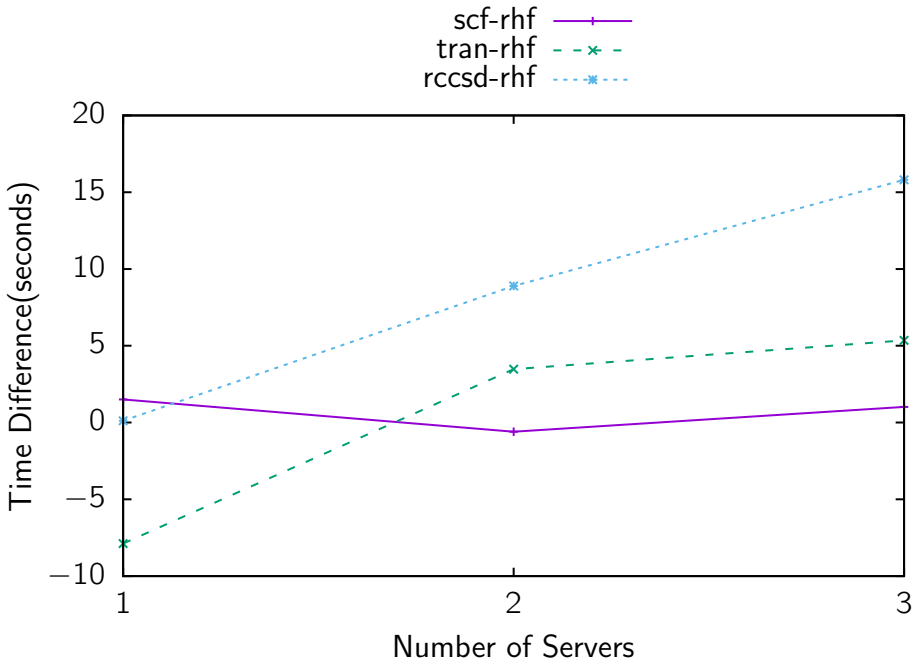

Figure 6-10. Effects of varying number of servers on $C_{12}H_{10}$ molecule.

6.3.4 RDMA

To benchmark RDMA, CUDA aware implementation of MPI was used. The size of the block was varied against time taken to complete the MPI transfer functions. The results for `GET` operation are presented in figure 6-16.

6.3.4.1 `GET`

The highest amount of time is taken by passing main memory address to MPI function, followed by passing page locked memory address to MPI and passing GPU memory address to CUDA Aware MPI implementation. Here, there are two important things to notice:

1. If CUDA aware MPI implementation is used, then DMA is used for transferring data from network fabric to GPU buffers directly.

2. It seems that just by page locking main memory, even without using GPU, DMA is invoked by MPI implementation for transfer of data from network fabric to main memory.

Table 6-5. Block size against time taken to copy data in page locked and non-page locked memory

Block Size	Pinned	Not Pinned
225	6.50641e-05	5.64773e-05
400	5.69895e-05	4.88665e-05
625	5.77066e-05	4.97829e-05
900	7.04844e-05	5.80885e-05
1225	6.93239e-05	5.98077e-05
1600	7.10636e-05	8.65366e-05
2025	7.18348e-05	6.23930e-05
2500	7.40327e-05	6.16256e-05
3025	7.27288e-05	6.21453e-05
3600	7.76853e-05	6.55819e-05
50625	5.90589e-05	0.000120046
160000	0.000131335	0.000248844
390625	0.000305546	0.000510937
810000	0.000607461	0.001069850
1500625	0.001107430	0.002165080
2560000	0.001884320	0.003161620
4100625	0.003069470	0.004814120
6250000	0.004606550	0.006790370
9150625	0.006778680	0.010549700
12960000	0.009527120	0.015224000

Table 6-6. Block Size against Page locked and non-page locked memory allocation

Block Size	Pinned Memory Alloc	Non Pinned Memory Alloc
225	3.59733e-05	9.35048e-07
400	1.99117e-05	4.13507e-07
625	2.71294e-05	2.93367e-06
900	1.76281e-05	5.90459e-07
1225	2.44398e-05	6.18398e-07
1600	1.55177e-05	2.54251e-06
2025	2.82563e-05	3.18140e-06
2500	2.04109e-05	3.00072e-06
3025	1.91461e-05	3.29316e-06

6.3.4.2 PUT

There is not much of difference in time in PUT operation, as presented in figure 6-17. The time taken by passing main memory address, page locked main memory address and GPU memory address to MPI function varies slightly as the size of block increases.

Table 6-7. Block size against Page locked and non-page locked memory deallocation

Block Size	Pinned Memory Free	Non Pinned Memory Free
225	2.78205e-05	4.19095e-07
400	2.54586e-05	4.04194e-07
625	2.88431e-05	5.94184e-07
900	3.09199e-05	4.95464e-07
1225	2.51215e-05	2.44007e-07
1600	2.58479e-05	2.98023e-07
2025	2.80552e-05	3.98606e-07
2500	3.52804e-05	6.81728e-07
3025	2.70978e-05	2.64496e-07

6.3.4.3 Total Transfer

Even though there is not much improvement in speed if the address of GPU memory is passed to MPI function, by passing GPU address directly, the complete operation can be executed on GPU without requiring the data to be copied to main memory. This saves two GPU and main memory copy operations and thus get a substantial improvement in overall operation. The time spent on overall operation is plotted in figure 6-18.

There is significant speedup as block size grows beyond 2000 with CUDA Aware implementation getting around 5 times faster than original main memory implementation for block sizes of around 3000.

6.3.5 Caching Page Locked Blocks

Caching page locked memory blocks is important since allocating page locked memory is 100 times costlier than non-page locked memory. To test caching, two parameters have been selected as most appropriate: hit ratio and time spent in allocation with and without caching.

Hit Ratio is simply the ratio of number of times allocation request was fulfilled by cached block and the total number times memory allocation request was made.

$$\texttt{hit\_ratio} = \frac{\textit{number of times memory allocation request fulfilled by cached block}}{\textit{total number of memory allocation requests made}}$$

The Hit Ratio does not depend on the size of memory block requested for, rather depends on the pattern of allocation and deallocation. Hence, the hit ratio is calculated for real

quantum chemistry calculation program. Table 6-8 presents the hit ratio for given calculation programs.

Table 6-8. Page Locked Memory Blocks hit rate

File	Hit	Miss	Ratio
scf_rhf_coreh	7975	214	0.97386738
tran_rhf_no4v	8028	229	0.97226596
rccsd_rhf	12859	1435	0.89960823
rccsdpt_aaa	14899	3012	0.83183519
rccsdpt_aab	15822	3696	0.81063634

It is evident that cost of allocation of page locked memory is paid for only less than 20% of the time for these programs.

To study the effect of caching on actual allocation time, the time taken by actual allocation in the above-mentioned programs is calculated for the cases when blocks are cached and not cached. Time taken when memory is not page locked is also calculated to study the cost paid for page locking even after caching. The results are plotted in figure 6-19.

By caching the page locked memory blocks, the allocation time is brought down by almost factor of 10. However, this is still greater than allocation time spent in case of non-page locked memory. This difference is difficult to plot, the numerical values of allocation time are presented in table 6-9.

Table 6-9. Page Locked Cached v/s Page Locked Uncached v/s Non-Page Locked allocation and deallocation times

File	Not Cached	Cached	Unpinned
scf_rhf_coreh	0.1717890	0.00710591	0.00138408
tran_rhf_no4v	0.0119096	0.00223255	3.35677e-05
rccsd_rhf	0.7167320	0.07038730	0.00620503
rccsdpt_aaa	0.0136935	0.00248791	8.10297e-05
rccsdpt_aab	0.0198253	0.00259629	4.97848e-05

The caching improves the time spent in allocation by a factor of 10. However the time spent in non-page locked memory allocation is still less than caching by a factor of 10 and in some cases around 100.

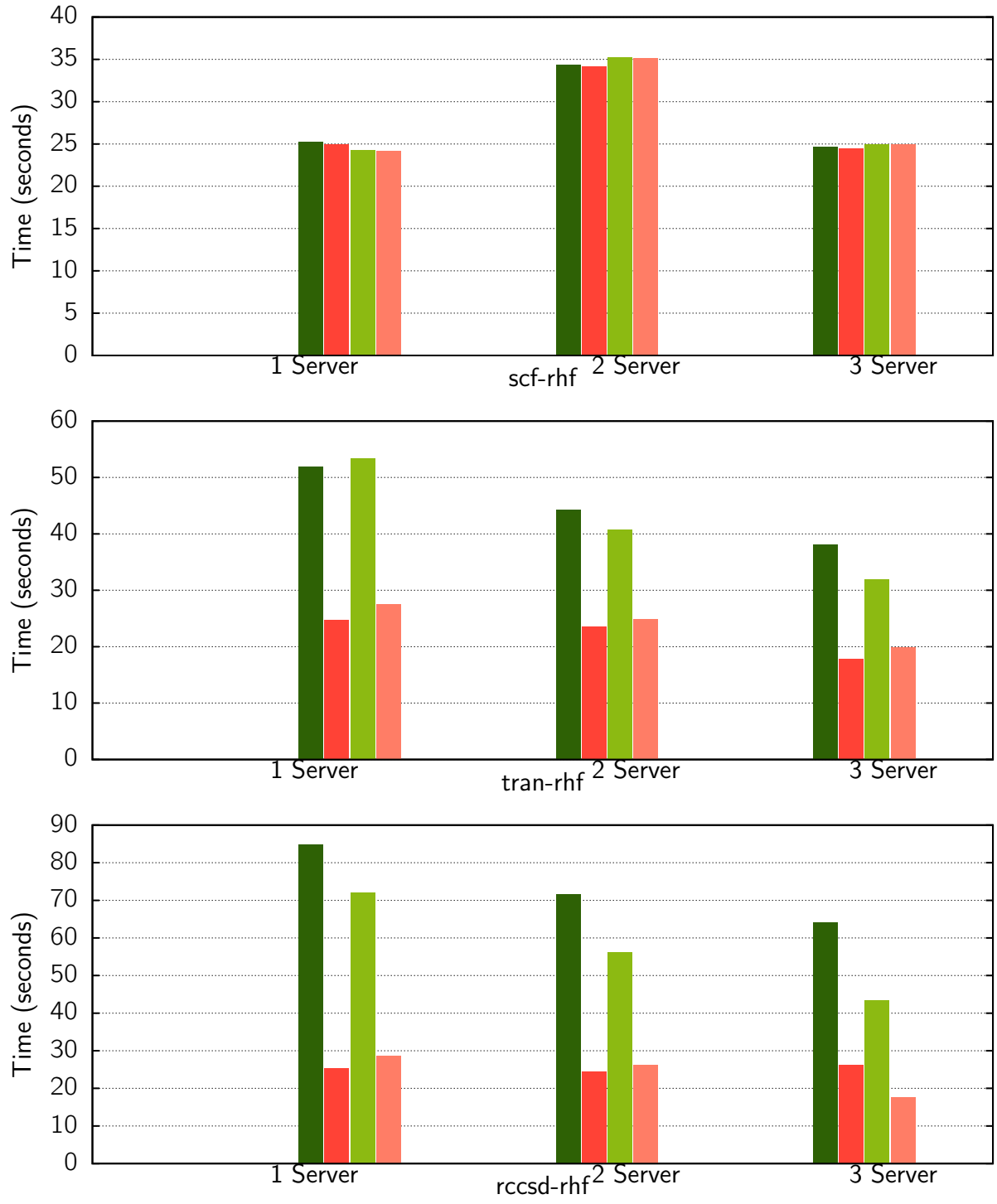

Figure 6-11. `wait_time` and barrier `wait_time` variation againsts number of servers.

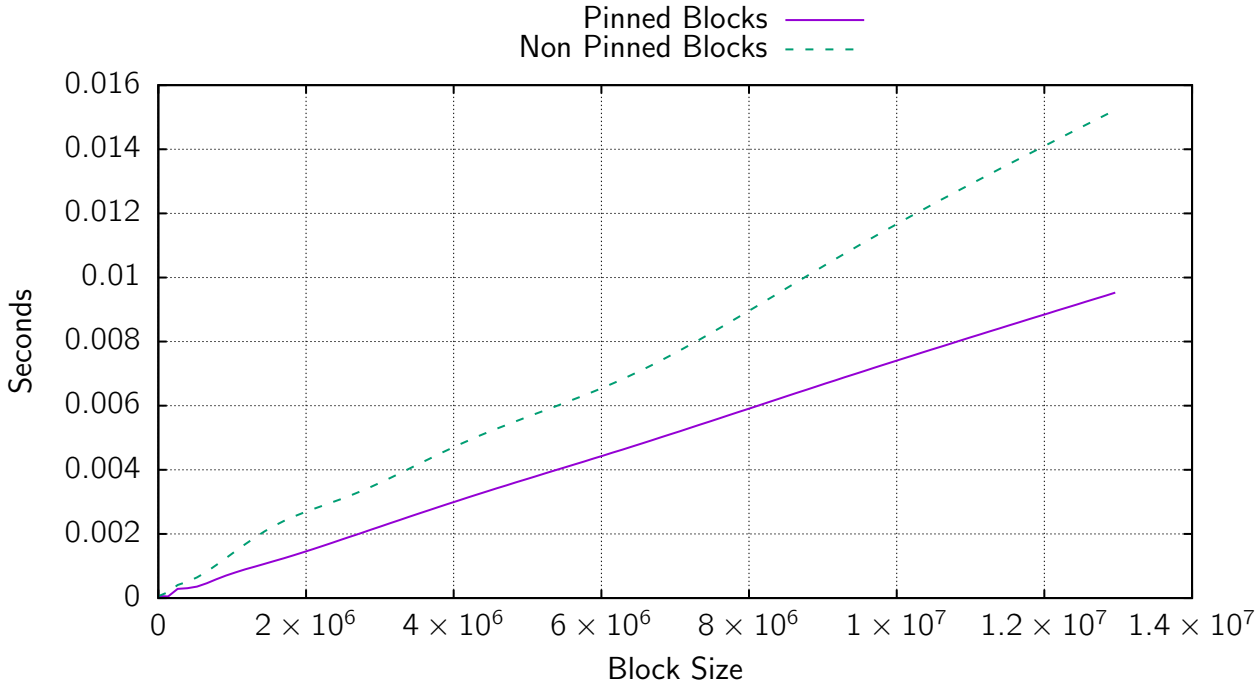

Figure 6-12. Time taken to transfer block to GPU for *pinned* and *non pinned* blocks

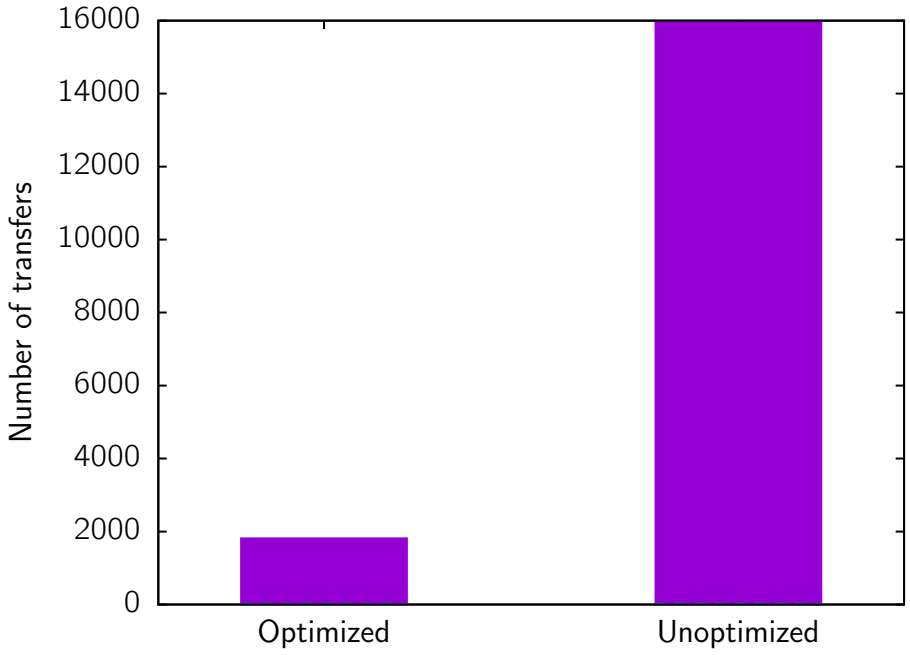

Figure 6-13. Optimized v/s Unoptimized Block Transfers for `rccsd_rhf.sialx`

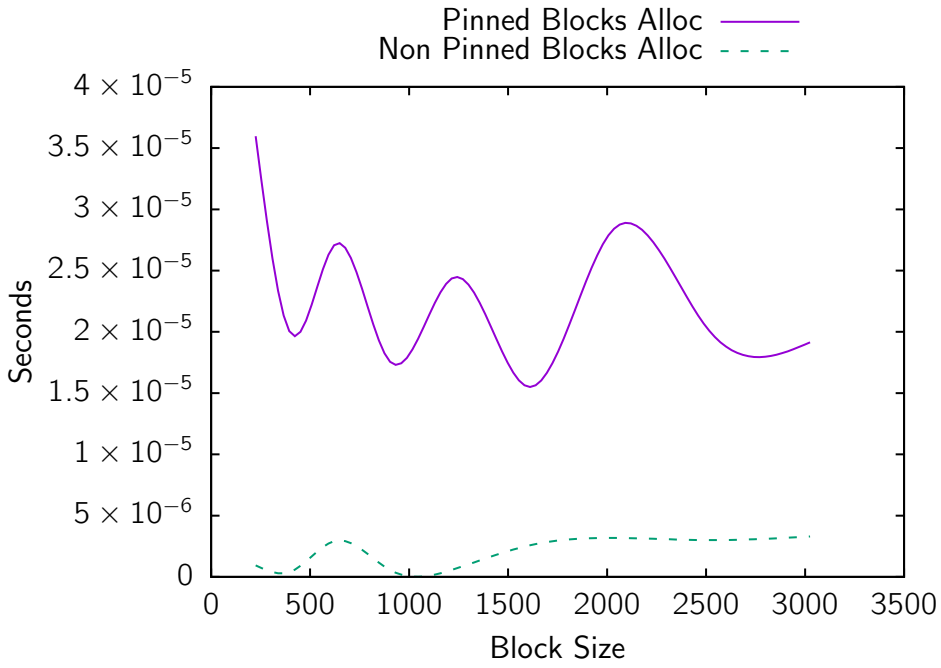

Figure 6-14. Pinned and non Pinned memory allocation

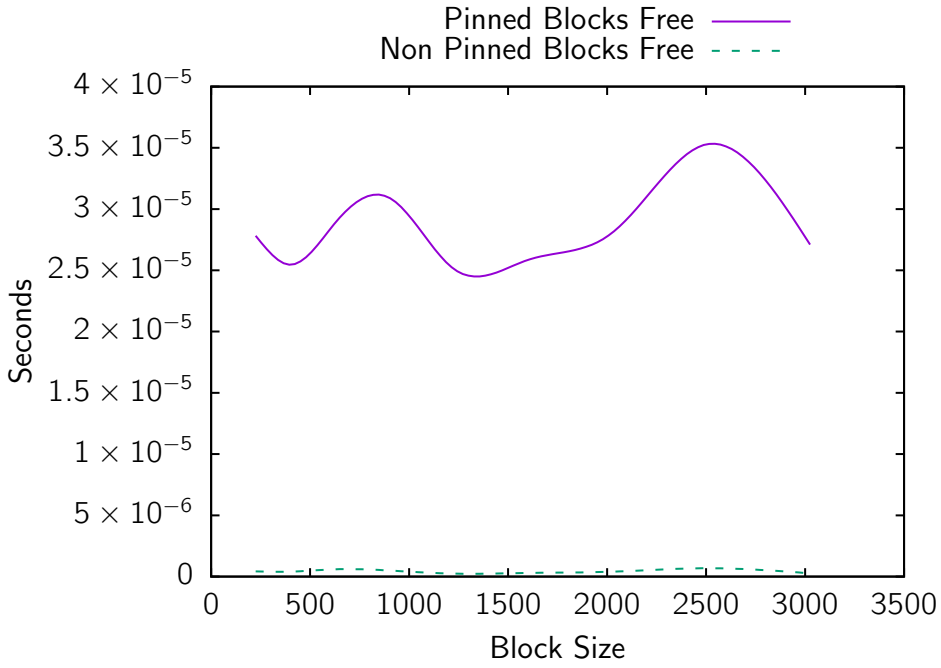

Figure 6-15. Pinned and non Pinned memory de-allocation

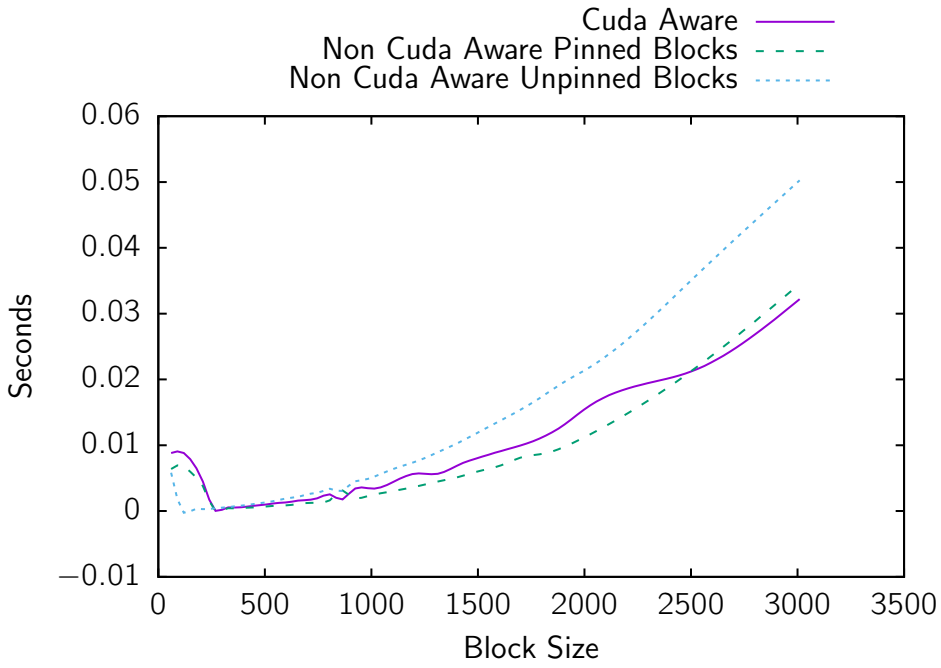

Figure 6-16. GPU and CPU buffer passed to `GET`

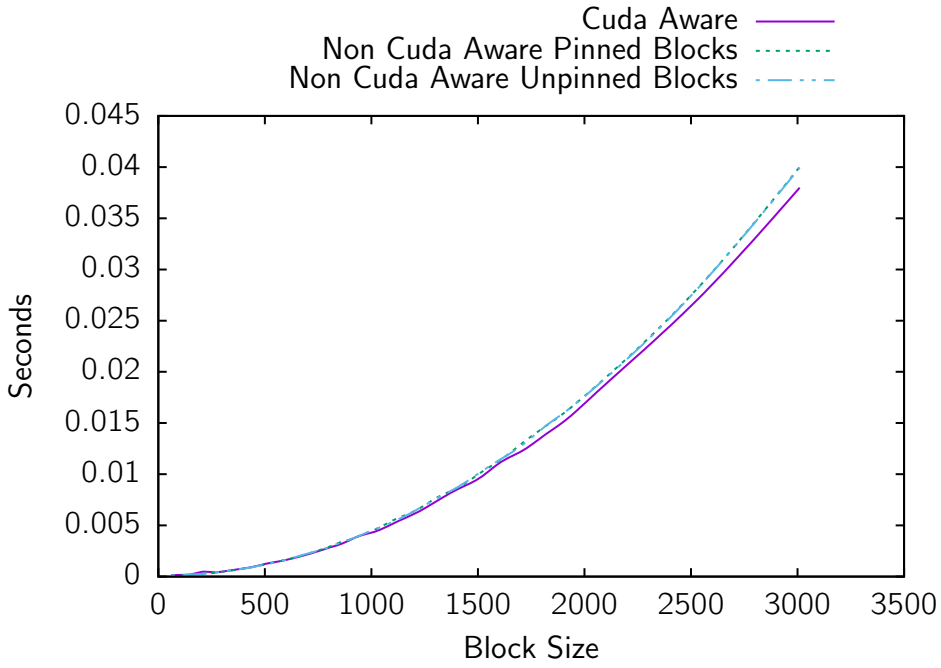

Figure 6-17. GPU and CPU buffer passed to `PUT`

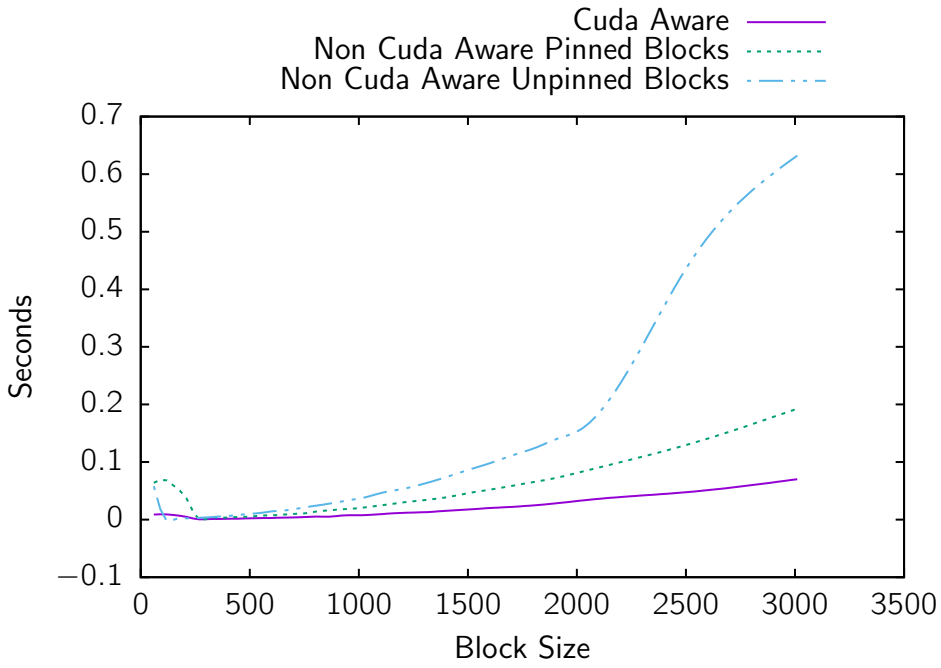

Figure 6-18. Total MPI transfer compared to CUDA Aware MPI transfer

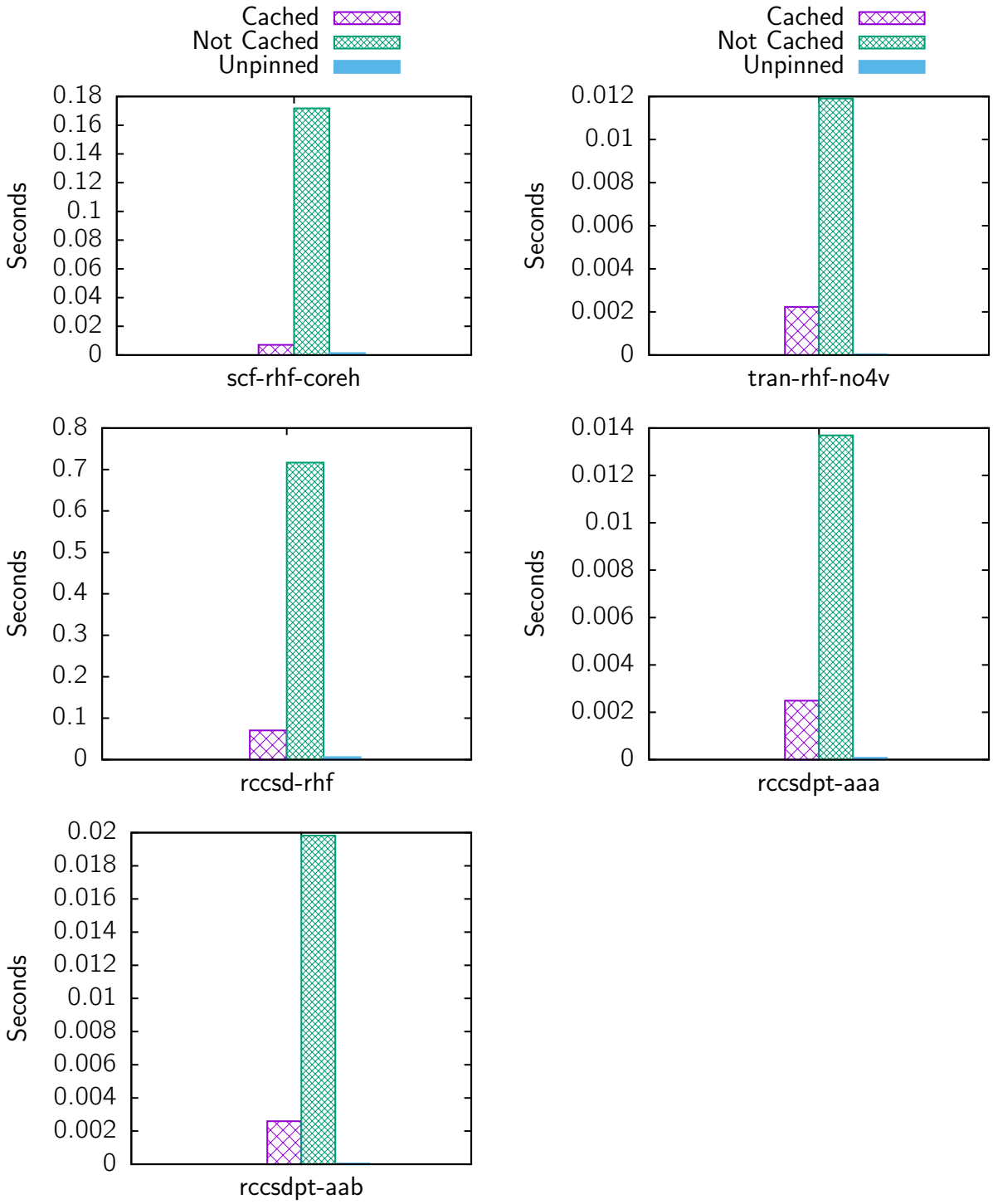

Figure 6-19. Page Locked Cached v/s Page Locked Uncached v/s Non-Page Locked allocation and deallocation times

CHAPTER 7
CONCLUSION AND FUTURE WORK

Two approaches to enhance the looping constructs and improve the runtime efficiency of SIAL interpreter are presented in this thesis. One applies the old technique of data block prefetching to SIA. Other improves the utilization of GPU to execute Super Instructions on GPU and make way to support other compute devices in future.

Though prefetching makes subsequent requests to a server efficient by reducing or eliminating `wait_time`, it makes the first request to the server expensive. Thus to make up for it, the length of the range of indices should be long enough. Hit ratio, as defined in section 6.2.1, can be used as a metric to evaluate the performance of prefetching. As seen in section 6.2.5, prefetching helps in reducing overall `wait_time` when most of the `wait_time` is due to blocking for data block transfer over the network.

Future work includes exploiting the provision to dynamically vary the number of blocks to prefetch to strike a balance between high cost of the initial request and relatively cheap subsequent requests.

Executing Super Instructions on GPU can speed up the computation for bigger block sizes. Transferring blocks between GPU and main memory is expensive but it can be avoided or reduced by directly collecting to or sending blocks from GPU memory. Page locked memory can improve memory transfer speed by invoking DMA and bypassing CPU. But page locked memory is expensive as compared to dynamically allocated memory. Page locked memory blocks can be cached and served with high cache hit ratio.

Further improvement in utilization of GPU can be achieved by implementing more super instructions on GPU and a way to easily port user defined super instructions to GPU; By exploiting the asynchronous memory synchronization between GPU memory and main memory by looking ahead for instruction which needs the memory to be transferred and initiating the asynchronous transfer as soon as data is ready. And exploring the feasibility of having access to GPU on servers and implementing RDMA to transfer memory blocks between workers and

servers. Lastly, implementing an advanced caching mechanism for page locked memory blocks can be worked upon to have more efficient caching mechanism.

APPENDIX
SIALX PROGRAM USED FOR BECHMARKING

Listing A.1 presents the sialx program used for benchmark various parameters while evaluating block prefetch.

Listing A.1. `put_test.sialx`: sialx program used for benchmarking prefetching

```
sial put_test
  predefined int norb
  aoindex i = 1:norb
  aoindex j = 1:norb
  distributed a[i,j]
  distributed b[i,j]
  temp t[i,j]

  scalar x

  print "starting loop"
  pardo i
    do j
      t[i,j] = (scalar)((i-1)*norb + (j-1)) + 1.0
      put a[i,j] = t[i,j]
    enddo j
  endpardo i

  sip_barrier

  pardo i
    do j
    get a[i,j]
    x = a[i,j] * a[i,j]
```

```
25      put b[i,j] = a[i,j]
26      enddo j
27    endpardo i
28
29  endsial put_test
```

Listing A.2 presents the sialx program used for benchmarking parameters used while evaluating GPU exploitation.

Listing A.2. `contraction_small_test.sialx`: sialx program to benchmark GPU exploitation

```
1  sial contraction_small
2    special fill_block_cyclic wr
3    aoindex i = 1:1
4    aoindex j = 1:1
5    aoindex k = 1:1
6    aoindex l = 1:1
7    aoindex m = 1:1
8
9    temp a[i, j, k, l]
10   temp b[j, k]
11   local c[i, l]
12
13   gpu_on
14   do k
15     do j
16       execute fill_block_cyclic b [j, k] 1.0
17       do i
18         do l
19           allocate c[i, l]
```

```
          execute fill_block_cyclic a[i, j, k, l] 1.0
          c[i, l] = a[i, j, k, l] * b[j, k]
        enddo l
      enddo i
    enddo j
  enddo k

  gpu_off

endsial contraction_small
```

REFERENCES

[1] Wilhelm Anacker and Chu Ping Wang. 1968. Performance Evaluation of Computing Systems with Memory Hierarchies. *IEEE Transactions on Electronic Computers* EC-16 (01 1968), 764 – 773.

[2] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706

[3] James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R. de Supinski. 2011. OpenMP for Accelerators. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era (IWOMP'11)*. Springer-Verlag, Berlin, Heidelberg, 108–121. http://dl.acm.org/citation.cfm?id=2023025.2023037

[4] Kiran Bhaskaran-Nair, Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Hubertus J. J. van Dam, Edoardo Apr, and Karol Kowalski. 2013. Noniterative Multireference Coupled Cluster Methods on Heterogeneous CPUGPU Systems. *Journal of Chemical Theory and Computation* 9, 4 (2013), 1949–1957. https://doi.org/10.1021/ct301130u arXiv:https://doi.org/10.1021/ct301130u PMID: 26583545.

[5] Swapnil Bhatia, Elizabeth Varki, and Arif Merchant. 2010. Sequential Prefetch Cache Sizing for Maximal Hit Rate. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '10)*. IEEE Computer Society, Washington, DC, USA, 89–98. https://doi.org/10.1109/MASCOTS.2010.18

[6] Michael Boyer. 2018. Choosing Between Pinned and Non-Pinned Memory. (2018). https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html

[7] Michael Boyer. 2018. Memory Management Overhead. (2018). https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_management_overhead.html

[8] Michael Boyer. 2018. Memory Transfer Overhead. (2018). https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html

[9] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. 2009. Accelerating Leukocyte Tracking Using CUDA: A Case Study in Leveraging Manycore Coprocessors. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/IPDPS.2009.5160984

[10] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. 1993. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 01 (ICPP '93)*. IEEE Computer Society, Washington, DC, USA, 56–63. https://doi.org/10.1109/ICPP.1993.92

[11] A. Eugene DePrince and Jeff R. Hammond. 2011. Coupled Cluster Theory on Graphics Processing Units I. The Coupled Cluster Doubles Method. *Journal of Chemical Theory and Computation* 7, 5 (2011), 1287–1295. https://doi.org/10.1021/ct100584w arXiv:https://doi.org/10.1021/ct100584w PMID: 26610123.

[12] Massimiliano Fatica. 2009. Accelerating Linpack with CUDA on Heterogenous Clusters. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. ACM, New York, NY, USA, 46–51. https://doi.org/10.1145/1513895.1513901

[13] Tianyi David Han and Tarek S. Abdelrahman. 2011. hiCUDA: High-Level GPGPU Programming. *IEEE Trans. Parallel Distrib. Syst.* 22, 1 (Jan. 2011), 78–90. https://doi.org/10.1109/TPDS.2010.62

[14] Nakul Jindal, Victor Lotrich, Erik Deumens, and Beverly A. Sanders. 2016. Exploiting GPUs with the Super Instruction Architecture. *Int. J. Parallel Program.* 44, 2 (April 2016), 309–324. https://doi.org/10.1007/s10766-014-0319-4

[15] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN '12)*. ACM, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[16] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.* 38, 3 (March 2012), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

[17] Seyong Lee and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.36

[18] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, and Karol Kowalski. 2011. GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems. *Journal of Chemical Theory and Computation* 7, 5 (2011), 1316–1327. https://doi.org/10.1021/ct1007247 arXiv:https://doi.org/10.1021/ct1007247 PMID: 26610126.

[19] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Karol Kowalski, and Gagan Agrawal. 2013. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. *Cluster Computing* 16, 1 (01 Mar 2013), 131–155. https://doi.org/10.1007/s10586-011-0179-2

[20] NvidiaStaff. 2012. How to Optimize Data Transfer in CUDA C/C++ — NVIDIA Developer Blog. (2012). https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

[21] NvidiaStaff. 2012. How to Overlap Data Transfers in CUDA C/C++. (2012). https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

[22] NvidiaStaff. 2013. An Introduction to CUDA-Aware MPI. (2013). https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[23] NvidiaStaff. 2013. NVIDIA Tesla GPU Accelerators Datasheet. (2013). http://www.nvidia.com/content/PDF/kepler/Tesla-KSeries-Overview-LR.pdf

[24] NvidiaStaff. 2013. NVIDIA Tesla K series GPU Accelerators Datasheet. (2013). http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf

[25] NvidiaStaff. 2018. Programming Guide CUDA. (2018). http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[26] NvidiaStaff. 2018. Standard DMA Transfer: GPUDirect RDMA: CUDA Toolkit Documentation. (2018). http://docs.nvidia.com/cuda/gpudirect-rdma/index.html#standard-dma-transfer-example-sequence

[27] OpenACC-standard.org. 2018. OpenACC. (2018). https://www.openacc.org

[28] openmpi.org. 2018. FAQ: Running CUDA-Aware Open-MPI. (2018). https://www.open-mpi.org/faq/?category=runcuda#mpi-apis-cuda

[29] osgx. 2011. Why is CUDA pinned memory so fast? (2011). https://stackoverflow.com/questions/5736968/why-is-cuda-pinned-memory-so-fast

[30] R. Hugo Patterson and Garth A. Gibson. 1994. Exposing I/O Concurrency with Informed Prefetching. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems (PDIS '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 7–16. http://dl.acm.org/citation.cfm?id=381992.383614

[31] Allan Kennedy Porterfield. 1989. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. Ph.D. Dissertation. Rice University, Houston, TX, USA. AAI9012855.

[32] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing (ICPP '13)*. IEEE Computer Society, Washington, DC, USA, 80–89. https://doi.org/10.1109/ICPP.2013.17

[33] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R. Trott, Greg Scantlen, and Paul S. Crozier. 2011. The development of Mellanox/NVIDIA GPUDirect over InfiniBand—a new model for GPU to GPU communications. *Computer Science - Research and Development* 26, 3 (01 Jun 2011), 267–273. https://doi.org/10.1007/s00450-011-0157-1

[34] Albert Sidelnik, Mara Garzarn, David Padua, and Bradford L Chamberlain. 2011. *Using the High Productivity Language Chapel to Target GPGPU Architectures*. Technical Report. University of Illinois.

[35] A. J. Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11, 12 (Dec. 1978), 7–21. https://doi.org/10.1109/C-M.1978.218016

[36] Alan Jay Smith. 1982. Cache Memories. *ACM Comput. Surv.* 14, 3 (Sept. 1982), 473–530. https://doi.org/10.1145/356887.356892

[37] Steven P. Vanderwiel and David J. Lilja. 2000. Data Prefetch Mechanisms. *ACM Comput. Surv.* 32, 2 (June 2000), 174–199. https://doi.org/10.1145/358923.358939

[38] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management (IWMM '95)*. Springer-Verlag, London, UK, UK, 1–116. http://dl.acm.org/citation.cfm?id=645647.664690

[39] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*. Springer-Verlag, Berlin, Heidelberg, 887–899. https://doi.org/10.1007/978-3-642-03869-3_82

BIOGRAPHICAL SKETCH

Anurag Peshne completed his schooling from Somalwar High School, Maharashtra, India and in September 2013 he received his bachelor's degree in Computer Science & Engineering from Visvesvaraya National Institute of Technology, Nagpur, India. After working for 3 years as software engineer, he joined the University of Florida to pursue his master's degree in Computer Science.