ENHANCING THE PARALLEL PROGRAMMING ENVIRONMENT SIA/ACES4 WITH GPU SUPPORT AND DATA PREFETCHING

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 would like to thank my committee members Dr. Erik Deumens and Dr. Alin Dobra for their advice and feedback on my thesis and research.

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.

TODO: Funding Agencies.

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

ENHANCING THE PARALLEL PROGRAMMING ENVIRONMENT SIA/ACES4 WITH GPU SUPPORT AND DATA PREFETCHING

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 solve computation involving very large, possibly sparse, multidimentional arrays. SIA consists of a domain specific programming language, SIAL, and its runtime. ACES4 is a computational chemistry package built using the SIA. A novel feature of the programming language is that SIAL provides intrinsic support for partitioning arrays into blocks and the domain programmers express algorithms in terms of operations on blocks rather than individual floating point numbers. Two enhancements, improvement in the GPU support and hiding of the network latency using data prefetching, are presented in this work.

In ACESIII, the previous realization of the SIA, the SIAL programmer was required to manually transfer blocks between host memory and GPU memory as well as to explicitly mark parts of SIAL code to be executed on the GPU. Copying data between GPU memory and host memory is expensive and can have a high impact on overall performance of GPU execution. In this work, automatic memory management is developed and several techniques that reduce and in some cases eliminate cost of memory transfer between GPU and host memory are implemented.

A common code pattern in SIAL is to request a block of data from a server followed by computation on that block, followed by sending it back to a server, in a loop. This pattern makes inefficient use of network and computational resources, since one of them remains underutilized while the other is being used. In order to tackle this issue, the runtime has been

extended to automatically prefetch data blocks. Empirical results indicate the optimal number of blocks to prefetch.

CHAPTER 1
INTRODUCTION

Advanced Concepts in Electronic Structure (ACES) is a series of computational chemistry packages used to calculate electronic energies, emphasizing coupled cluster methods. ACESII [2] is a serial program, ACESIII [3] and ACES4 are both parallel programs built using SIA concepts. The Super Instruction Architecture (SIA)[1] 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. The SIA consists of a domain-specific programming language, Super Instruction Assembly Language (SIAL), and its runtime system. A novel feature of the programming language is that SIAL provides intrinsic support for partitioning arrays into blocks and the domain programmers express algorithms in terms of operations on blocks rather than individual floating point numbers. Two enhancements for ACES4, improvement in the GPU support and hiding of the network latency using data prefetching, are presented in this work.

1.1 Issues of Working with GPUs

In ACESIII[4] 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 keeping a version number associated with each device memory and tracking changes to blocks by every compute device.

Nonetheless, there is a need for the SIAL programmer to decide which portion of the SIAL code is suited for execution on GPUs. Transferring data between GPU and main memory is expensive[5], and can dominate the overall time spent in computation. Therefore, marking correct block of code, that 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 operations involved in the computation. 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 with the size of the block. Lastly, the same SIAL programs are used to calculate different results by supplying different input parameters. It is possible that, for the same program, the overall time required for execution utilizing GPUs be greater than CPU only for some size of data and less than CPU for another size of data. This may happen if the speed gains by executing on GPUs 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 a large array from a server, processing the block, computing resulting block and sending it back to a server. Though the operation of requesting a 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, which does the transfer concurrently with the block processing, has been implemented. 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 more details about SIA and ACES. Chapter 3 presents previous work done in use of GPUs for general purpose computing, efficient GPU memory transfer, and the technique of prefetching to hide latency in accessing the data. Chapter 4 describes the technique of block prefetching implemented in SIA. Chapter 5 presents the optimizations done for efficient GPU memory transfer. And finally, chapter 6 concludes this thesis by presenting the conclusion and suggestion of future work.

CHAPTER 2
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.

2.1 SIA

The SIA is a special purpose, domain agnostic, parallel programming framework which is engineered for solving computation on very large, possibly sparse, multidimensional arrays. Programmer using SIA expresses the computation in terms of blocks of multidimensional arrays and instructions which operate on these blocks.

In contrast, in conventional programming languages a 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 deal with tedious and error prone index arithmetic.

Algorithms in SIA can be directly expressed in terms of blocks. This gives the performance benefits of using blocks and at the same time relieves programmer from error-prone work of dealing with indices and looping. Expressing computations in terms of blocks has multiple performance advantages: moving data 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: the work can be distributed among participating nodes in a natural way. This is essential when multidimensional arrays are too large to hold in physical memory of a single processor.

2.2 Overview of ACES

ACES4 is an implementation of the 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 chemists try to find approximate solutions to the electronic Schrodinger equation using Coupled cluster methods. There exist other methods to calculate approximate solutions but Coupled cluster methods, although computationally expensive, 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 upto 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. The 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.

2.3 Architecture of the SIA

The 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.

The 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.

2.4 SIAL

SIAL is a programming language provided for expressing problems of the target domain. The idea behind SIAL is to keep the domain issues separate from the platform issues. 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, in addition to 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 operations or *super instructions*. It has several common block operations builtin. The parallel looping construct, `pardo`, loops over multiple indices and distributes loop iterations 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 any general purpose programming language by following the SIAL calling interface. There are implementations of super instructions in C++ and Fortran in ACES4. Since these languages are much closer to the hardware, these super instructions can be written in a very optimized way. Further more, 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.

2.4.1 SIAL Interpreter

The SIA includes a 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, 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 depend on the amount of physical memory present on the processor.

2.5 Block Structure

As described above, instead of processing data as a single floating point number, the SIA processes blocks of floating point numbers. These blocks are chunks of even larger multidimensional arrays. Blocks are represented inside SIAL interpreter using the `Block` class. The 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 the `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 logic build into the `Block` class to automatically synchronize memory for various devices, so that if one device modifies 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.

2.6 Executing Super Instructions on GPU

There are two ways in which GPUs are exploited in the 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 for GPU are builtin 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.

CHAPTER 3
RELATED WORK

In this chapter, prevous work done by others on General Purpose computing on Graphics Processing Units (GPGPU) as well as using GPUs specifically for evaluating scientific calculations and using prefetching to hide latency in accessing data, is discussed.

3.1 GPGPU

This section discusses previous developments in porting existing code to GPUs using directives, using CUDA directly to speed up scientific calculations and techniques to efficiently transfer data to GPU memory.

3.1.1 Directive Based GPU programming

There have been several developments made in the area of directive-based programming models. These developments include OpenACC [6], OpenMP for accelerator [7], and less updated developments including OpenMPC [8] and hiCuda [9]. The approach taken by these models is to make an implementation GPU-ready by placing directives which make execution of loops parallel.

Directives used in ACESIII, were to manage data transfer between GPU and host memory and to mark region of code suitable for execution on GPUs. These directives are very different from the directives mentioned in the above models, which hide the complexities of working with GPUs and allows developers to work at higher abstraction than CUDA. However, these models can be used to make an existing CPU implementation of a super instruction, GPU-ready, by inserting suitable directives.

3.1.2 GPU in Computational Chemistry

Various implementations of coupled cluster methods, an important class of problems in computational chemistry, on GPUs have been reported [10–12]. 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 tensor contraction operators on a GPU by Ma et al. [13] 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 were reported for CCSD(T) calculation on CPU/GPU hybrid systems. Such optimized operators could be incorporated into SIA as built-in super instruction for contraction.

3.1.3 GPU Programming in Other High-Level Programming Languages

Several attempts have been made to support GPUs in general purpose high-level programming languages. A few examples include PyCUDA [14], jCUDA [15] and Chapel [16]. These toolkits provide an interface to GPUs from high-level programming languages such as Python and Java, but the programmer still needs to deal with low-level GPU bookkeeping such as memory management and defining kernels. As described in section 3.1.4, this is similar to ACESIII, where the programmer has 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 such implementations and availability of 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.

3.1.4 GPUs in Previous Versions of ACES

Work has been done to exploit GPUs in previous version of ACES, ACESIII[4]. This implementation required the programmer to mark a region of code to be executed on a GPU using directives `gpu_on` and `gpu_off` and manually manage memory transfer to and from the GPU. This puts a burden of managing memory and recognizing a suitable suitable region of code for execution on GPUs on the domain programmer. If a region of SIAL code that is not suitable for execution on GPUs is marked, it can result in suboptimal use of resources.

Executing operations using small blocks on GPUs can be slower than executing on the 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 determine if a block is large enough so that benefit of executing it on GPUs becomes significant.

3.1.5 Optimizing GPU Memory Transfer

3.1.5.1 Host Memory Transfer

GPUs cannot work directly with pageable host memory. One of the key condition for effectively exploiting GPUs is to optimally manage data transfer between host memory and GPU memory. This is true for any computation running on GPUs and not just High Performance Computing computations. The work done by Fatica [17], in accelerating LINPACK describes various ways for optimizing memory transfers. Their implementation intercepts calls to `DGEMM` and `DTRSM` and executes them simultaneously on both GPUs and CPU cores. To achieve faster memory transfer, the implementation uses fast *PCI-E* transfers by page locking or *pinning* the host memory. Using PCI-E transfer, they observed a speed improvement of 5 GB/s from 2 GB/s. Kaldewey et al [18] describes similar technique to optimize memory transfer using common address space for CPUs and GPUs, known as Unified Virtual Addressing (UVA). This helps memory transfers in two ways: it allows GPUs to work with memory greater than available physical GPU memory and also relieves the programmer from 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 has been reported in their work. Host memory page locking is a common technique in both of the works, which enables GPU DMA controller to access host memory at PCI-E speeds. This technique can be used to speed up the SIA block transfers. This can be incorporated into an automatic memory synchronization module such that it becomes transparent to the domain programmer.

3.1.5.2 Network Transfer

GPUs are often deployed in a cluster to accelerate computationally intensive general-purpose tasks. Still, for communication with other nodes, 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 [19] introduced 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 [20] 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 GPUs and sending it back, bypassing the CPU. This can help in reducing host and GPU memory transfers.

3.2 Prefetching

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

Prefetching is an old technique [21–23] used to move data up in the 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 field of high performance computing.

Hardware prefetching techniques, which includes transferring separate cache blocks [24], were implemented before software techniques became available. Porterfield [25] 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 waiting for the data transfer to complete. 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 [26] implemented informed prefetching mechanism for IO intensive application to exploit highly parallel IO systems. The mechanism depends on disclosure of future accesses dynamically. In the 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.

There has been work done by Bhatia et al [28] 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. In order to determine if the cache size is too large, the eviction cache is observed and if it does not receives any hits then the size of the cache is decremented. In SIA, SIAL scope rules indicate when blocks can be evicted.

In SIA, blocks which are needed to be prefetched can be predicted with more accuracy than memory blocks. Further more, the exact sequence in which blocks are needed is also known. This makes prefetching in SIA free of some of the problems faced in general memory block prefetching discussed in above section. However, a large of number of blocks should not be prefetched at once, since it might cause network congestion.

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.

Listing 4.1. Code fragment from ACESIII for CCSD calculation

```
1  PARDO i1, i, a, i2 # parallel looping construct
2
3    # sends a request for block of array VSpipi to the server
4    # Asynchronous operation, the runtime doesn't wait for the transfer to be
       completed
5    GET VSpipi[a,i2,i,i1]
6    GET t1a_old[a,i2]
7
8    # the runtime has to first wait for data to be ready before executing block
       operation
9    Tii[i1,i]      = VSpipi[a,i2,i,i1]*t1a_old[a,i2]
10
11   # sends data back to the server
12   # synchronous operation, the runtime has to wait until the transfer is
       completed
13   PUT Fmi_a[i1,i] += Tii[i1,i]
14
```

```
15 ENDPARDO i1, i, a, i2
```

It is clear from the SIAL code fragment in Listing 4.1 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. There are two types of looping construct in SIAL: the `do` loop that iterates over the indices one by one and the `pardo` loop that distributes the iterations among workers. 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 4.1. *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 4.2. *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 4.3. *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 4.4. *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
```

- `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 4.5. *update_indices() → bool*

```
1:  function UPDATE_INDICES
2:      current_indices ← peek_indices(IndexList.begin())
```

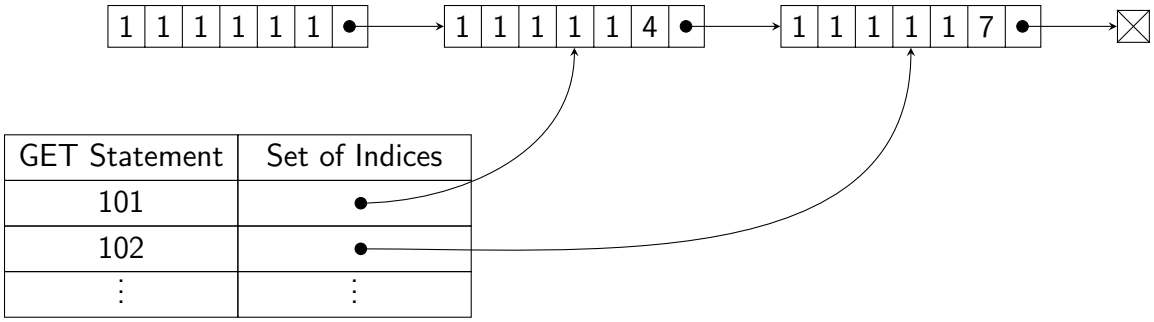

Figure 4-1. `prefetch_indices` saves mapping of line number to position in the list

```
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. Function `prefetch` indices is free to point at any set of indices on the list.

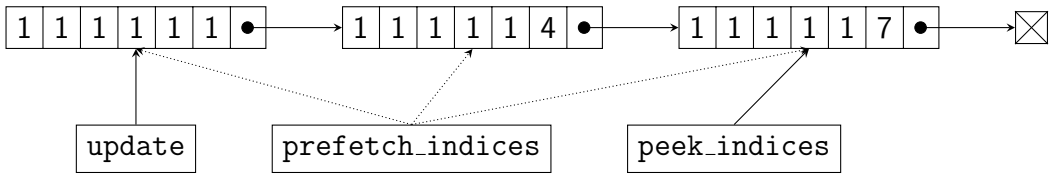

Figure 4-2. `update` consumes, `peek_indices` produces if needed, `prefetch_indices` is free to move along list

4.3 Experiments

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

4.3.1 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:

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

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.

4.3.2 Index Range

The range of indices is the range of values the indices in the loop can take. The range of indices are defined by the SIAL programmer at the start of the program. The range of indices can have a high impact on prefetching. To study the relation between index range and prefetching, hit_ratio is observed by varying the range of indices. This is presented in figure 4-3.

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 range is 1, the hit_ratio with prefetching is equal to with no prefetching. As the 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.

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 activates

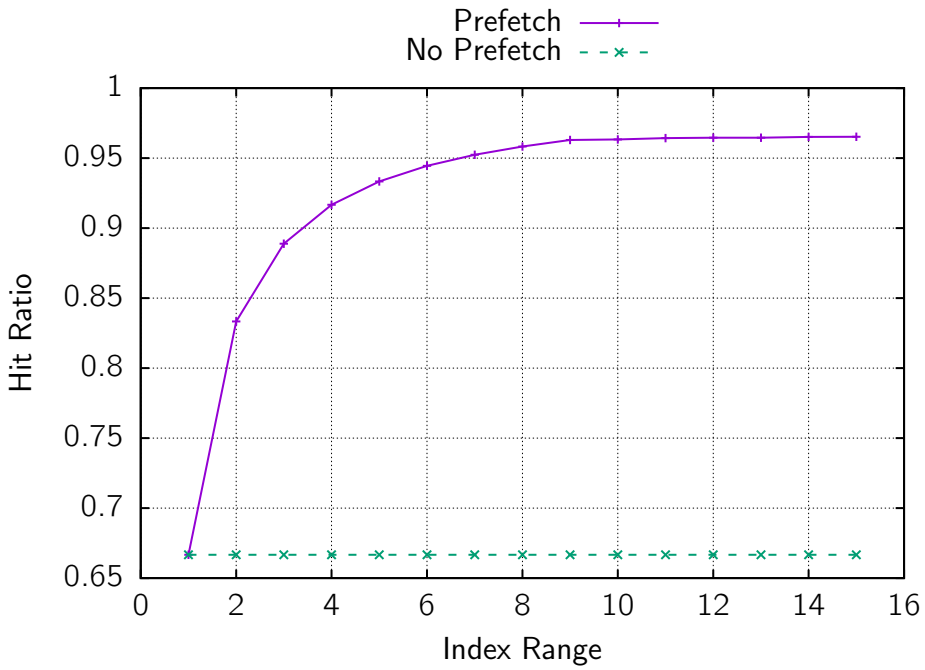

Figure 4-3. Index range vs `hit_ratio`

prefetching, network congestion increases and the first request to the server becomes costlier. This is presented in figure 4-4.

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 index range should be sufficient enough. The mean time taken per iteration is plotted against the index range in figure 4-5.

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

4.3.3 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_`

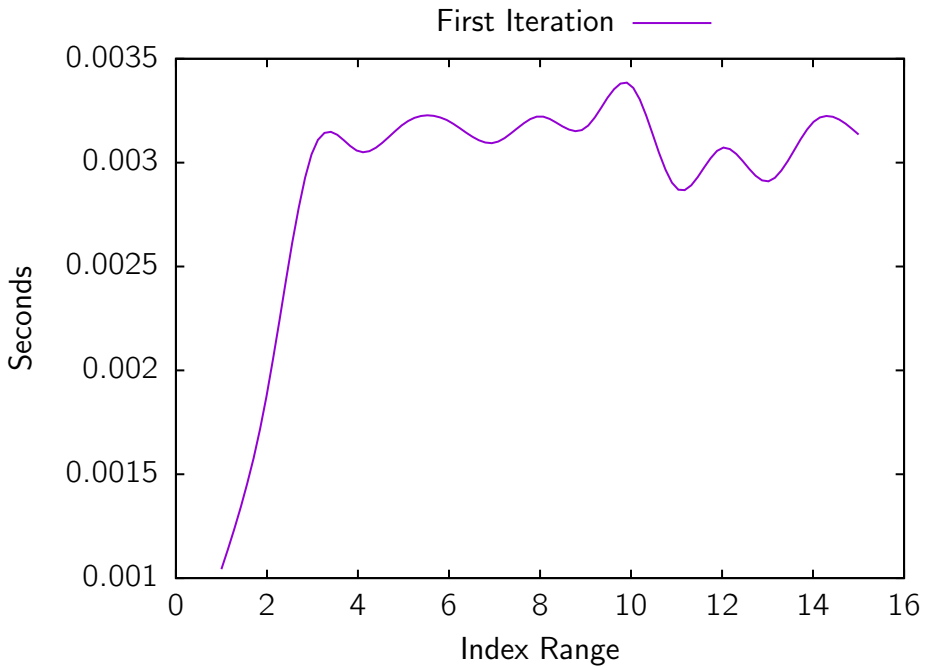

Figure 4-4. Index range vs `wait_time_` for first iteration

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 4-6.

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 4-7.

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 4-8

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.

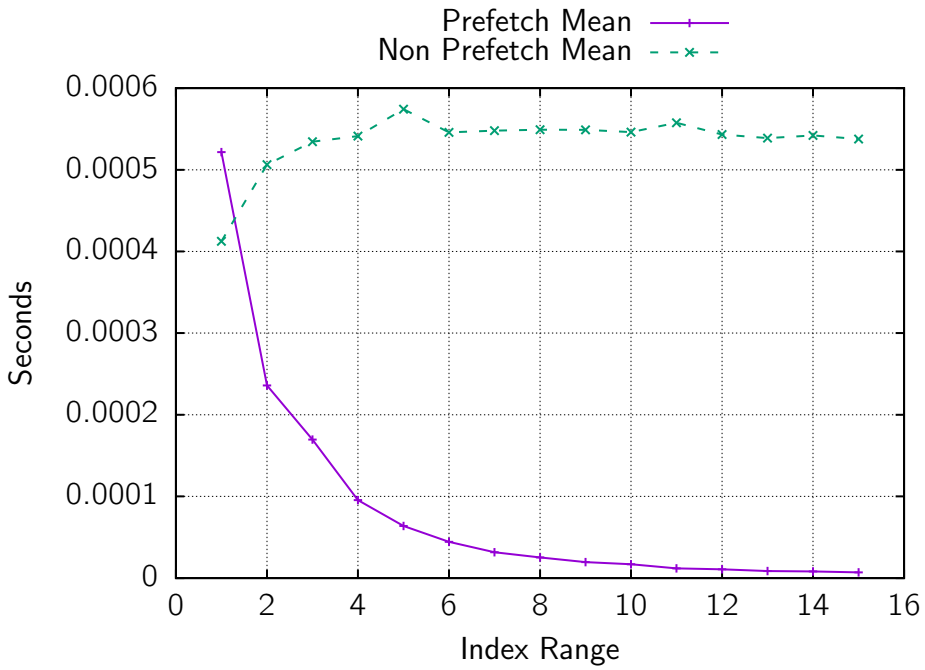

Figure 4-5. Index range vs `wait_time_` per iteration in prefetched and no prefetched loop

4.3.4 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 4-9.

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 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 4-10.

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

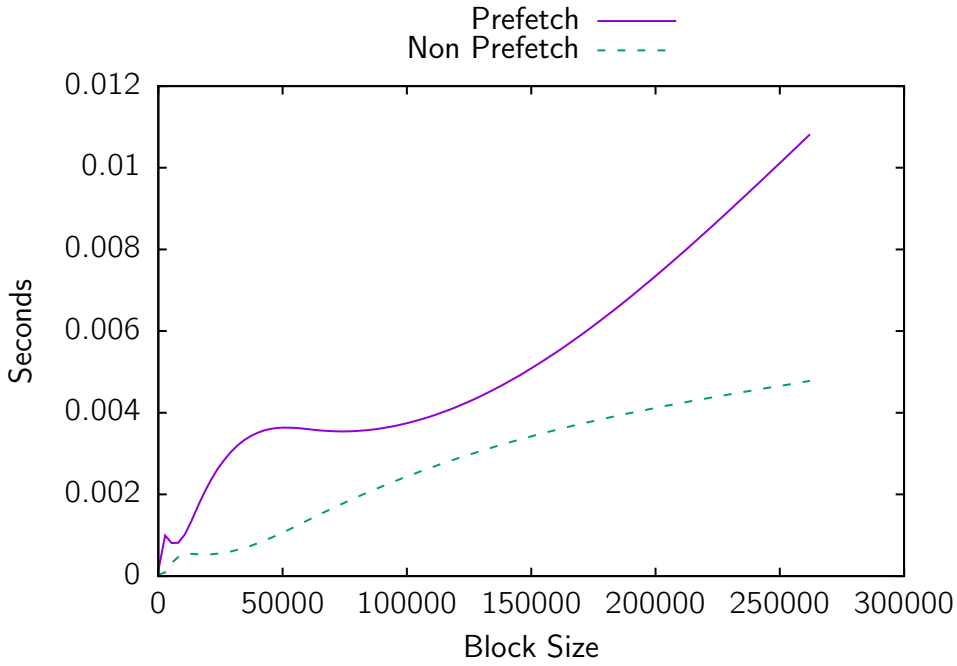

Figure 4-6. Block size vs `wait_time_` for first iteration

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 4-11 by plotting number of blocks to prefetch against `hit_ratio`.

The 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.

4.3.5 Overall Improvement

4.3.5.1 $C_{12}H_{10}(BP)$ Molecule

To study the effect of prefetching on real-world application, a job from ACES4 on $C_{12}H_{10}$ molecule in a CC-PVDZ basis 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 4-12.

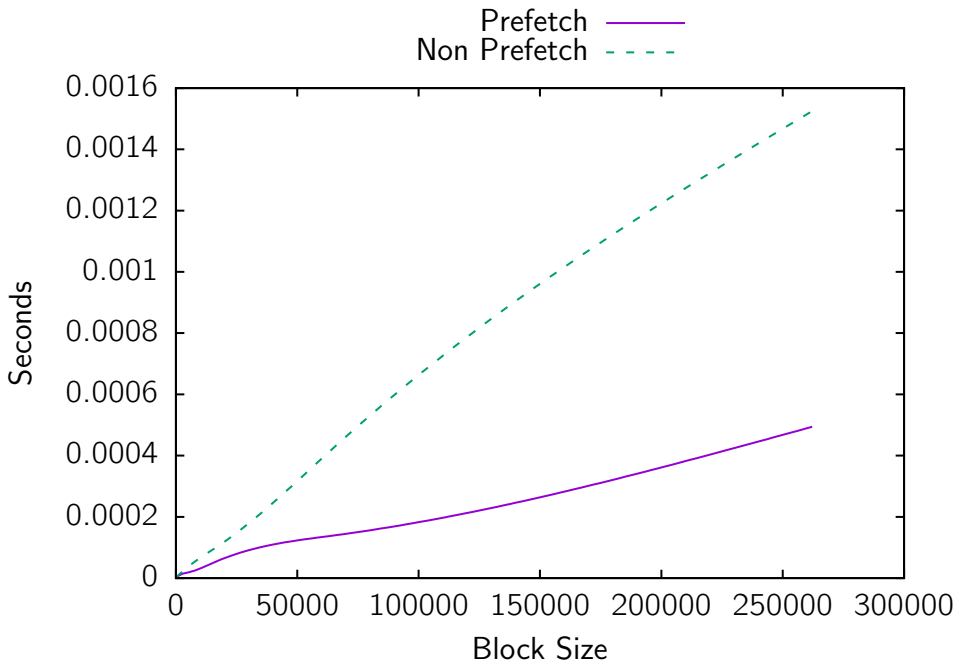

Figure 4-7. Block size vs mean `wait_time_` for prefetched and no prefetch loop

With the increase in the number of servers, the overall runtime for programs with prefetching decreases. This is clear from figure 4-12, as the difference in the runtime of 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 4-13.

In the figure 4-13, 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. The overall improvement in `wait_time` for CCSD calculation is about 27%.

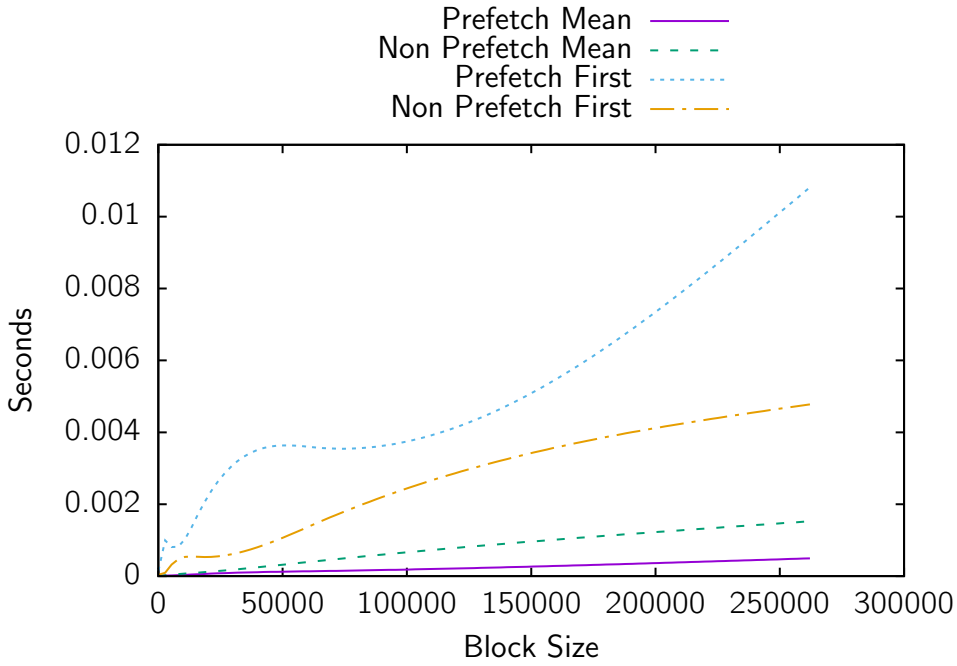

Figure 4-8. Block size vs mean `wait_time_` for prefetched and no prefetched loop

4.3.5.2 C_{20} Molecule

Another, much bigger, calculation was executed to test the effect of prefetching in presence of higher number of participating nodes. There were two runs of the calculation, first with 512 CPUs and second with 32 CPUs. In both of cases, the number of workers and servers were kept equal to maximize the effect of prefetching.

Results of executing CCSD calculation on C_{20} molecule in a AUG-CC-PVDZ basis is plotted in figure 4-14.

The overall improvement in the case of 512 CPUs is not significant. It is 2%. This can be attributed to the number of participating workers in the calculation. Since the number of workers was very high, the index range got distributed over a large number of workers and each worker got insufficient range for prefetching. To confirm this is the case, another experiment with same molecule and same basis function was conducted, with much lesser number of workers. The result for this calculation is presented in figure 4-15.

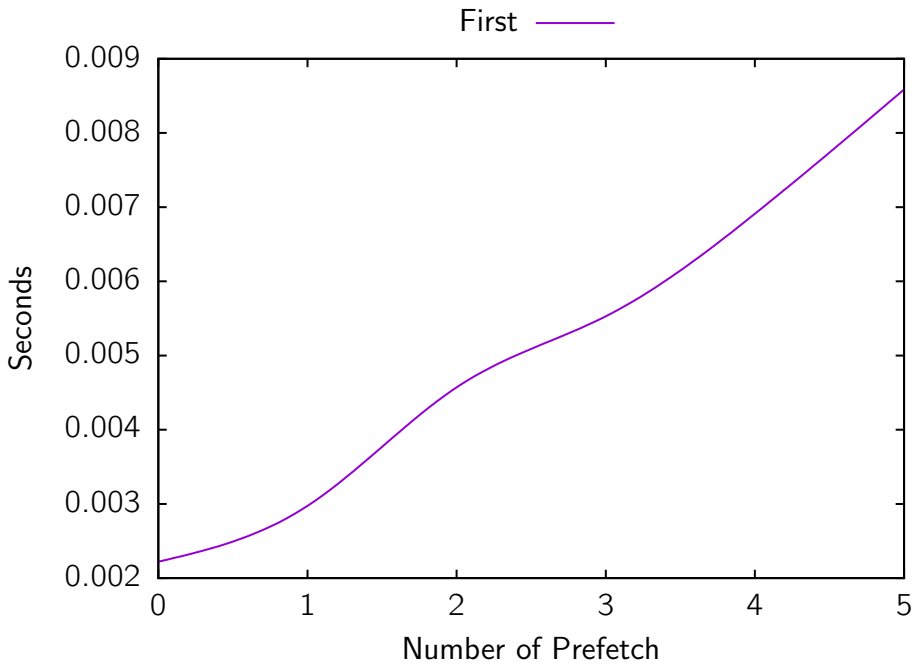

Figure 4-9. Number of block prefetched vs `wait_time_` for the first request

The overall improvement, in `wait_time` in the case of 32 CPUs, is 49%. The range of index was divided over much smaller number of workers and hence each one of the 16 workers had enough index range to prefetch data blocks.

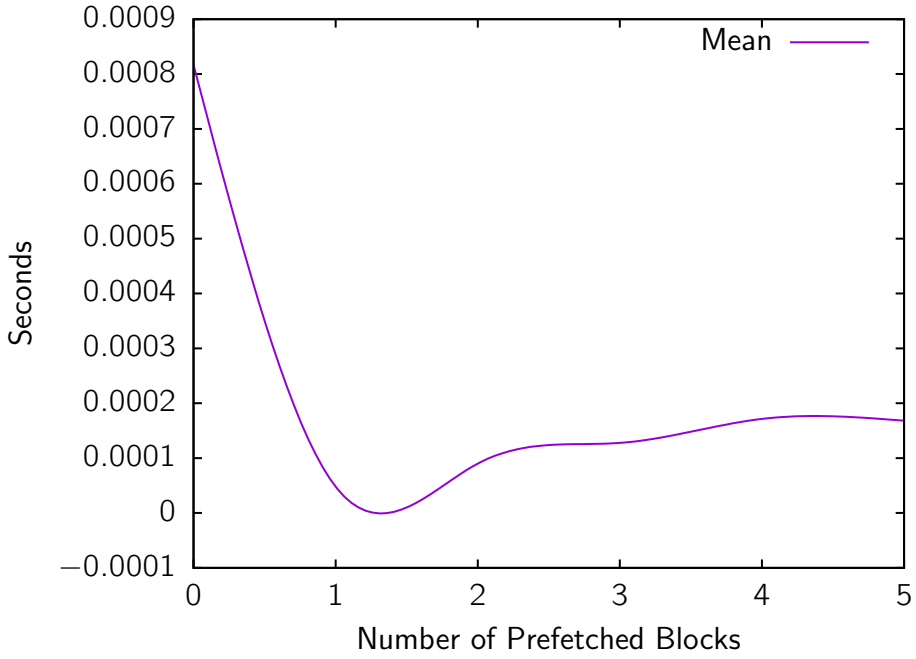

Figure 4-10. Number of block prefetched vs mean `wait_time_`

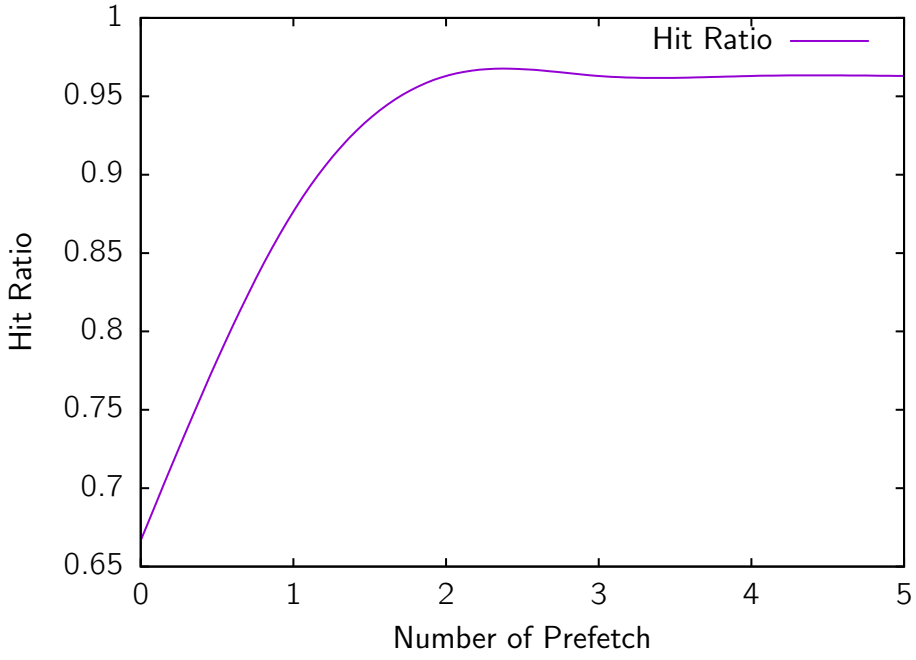

Figure 4-11. Number of block prefetched vs the hit ratio for the first request

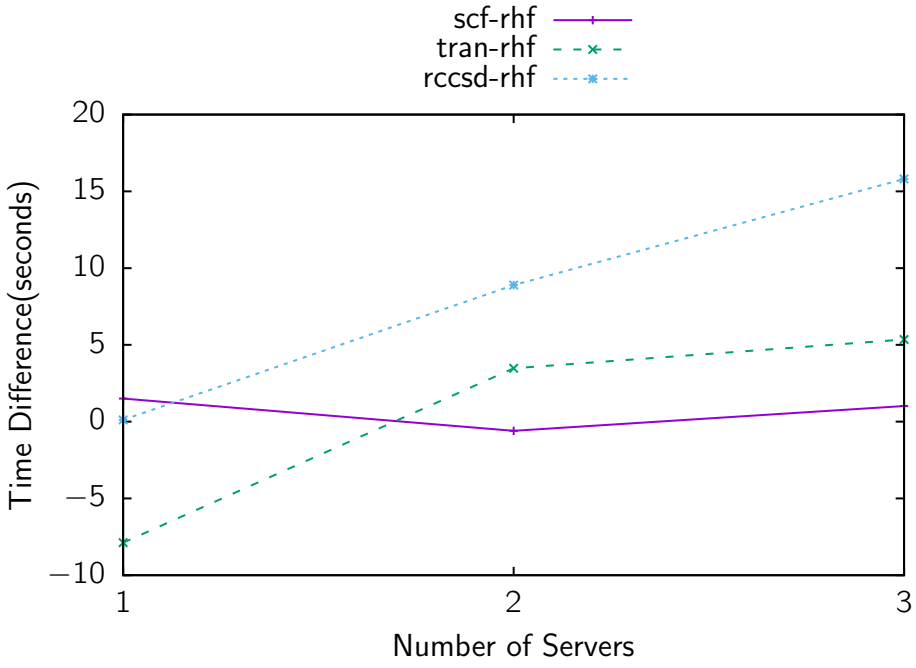

Figure 4-12. Effects of varying number of servers on $C_{12}H_{10}$ molecule.

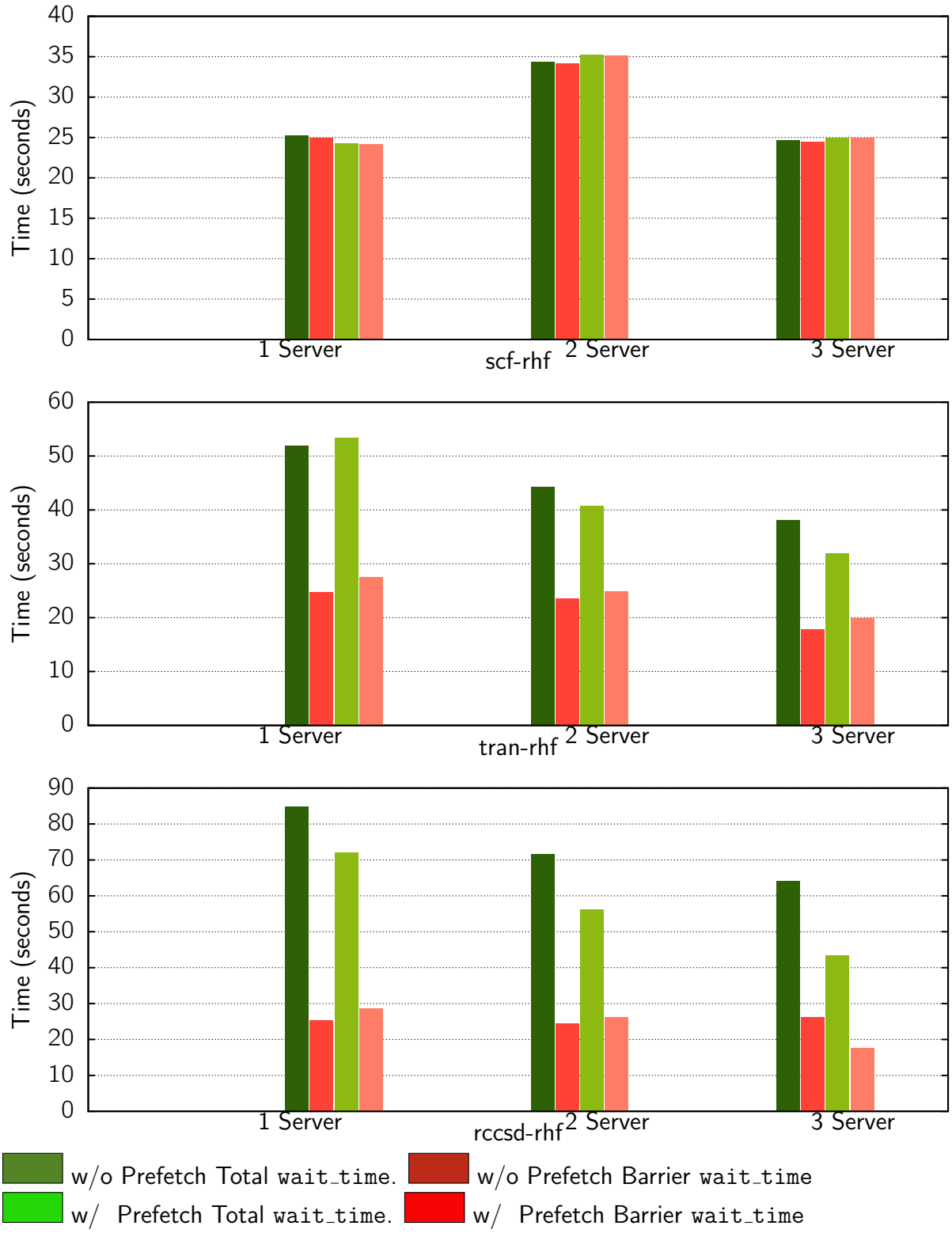

Figure 4-13. Overall `wait_time` improvement in $C_{12}H_{10}$ molecule in a CC-PVDZ basis

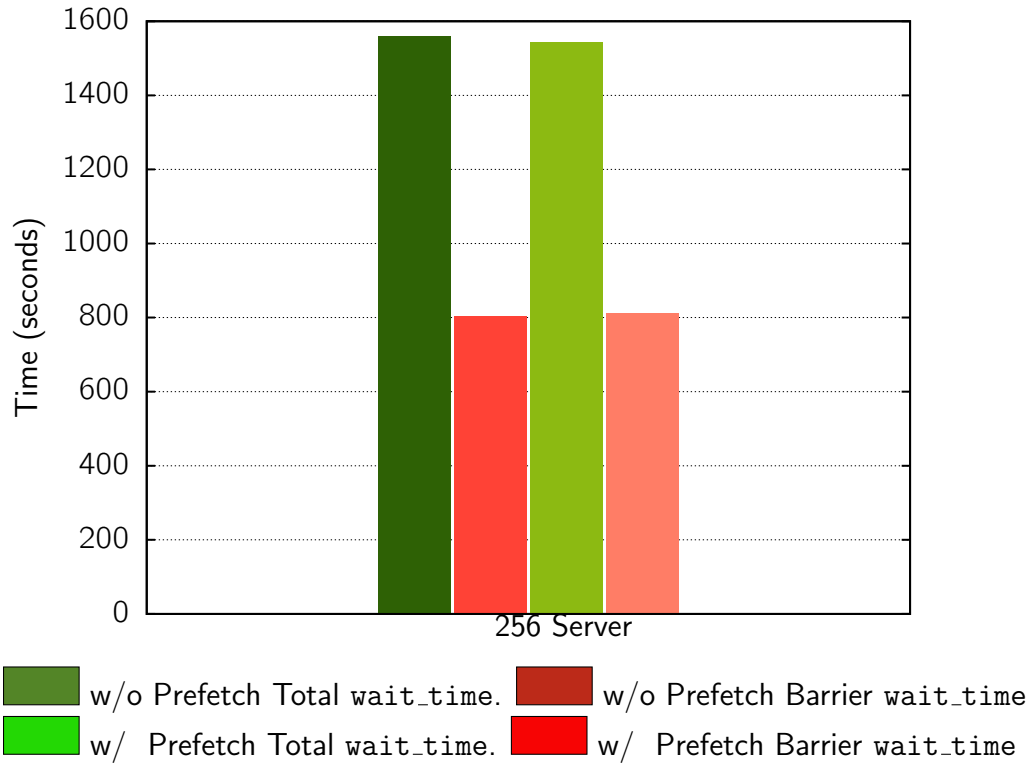

Figure 4-14. Overall `wait_time` improvement in CCSD calculation for C_{20} molecule in a AUG-CC-PVDZ basis with 512 CPUs

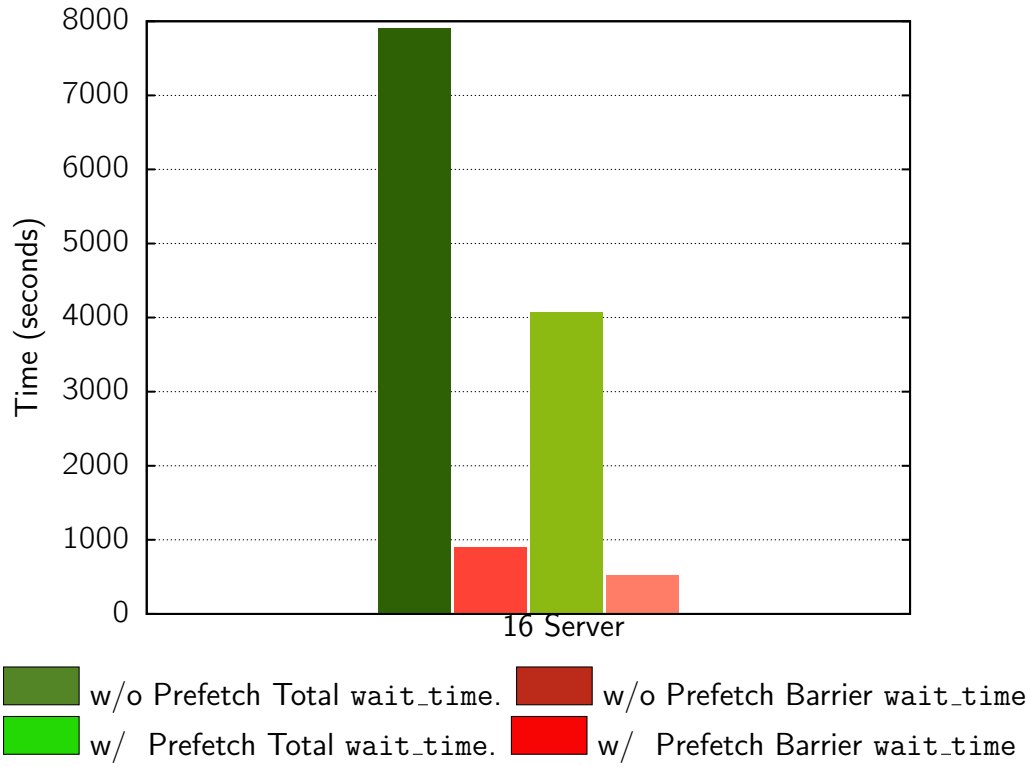

Figure 4-15. Overall `wait_time` improvement in CCSD calculation for C_{20} molecule in a AUG-CC-PVDZ basis with 32 CPUs

CHAPTER 5
EXPLOITING GPU

5.1 Background

5.1.1 Prior Work

ACESIII supported exploitation of GPUs. However, the SIAL programmer has 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 has to explicitly 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
14 DO j
15     #perform computations on GPU
16     Yab(mu,i,nu,j) = 0.0
17     Y1ab(nu,j,mu,i) = 0.0
18     gpu_allocate Yab(mu,i,nu,j) # allocate temp blocks on GPU
```

```
19      gpu_allocate Y1ab(nu , j ,mu, i )
20      #contraction Y1ab(nu,j,mu,i) = Yab(mu,i,nu,j) #permutation
21      Yab(mu,i,nu,j) = aoint(lambda,mu,sigma,nu)*LTAOab(lambda,i,sigma,j)
22      LT2AOab1(mu,i,nu,j) += Yab(mu,i,nu,j) #elementwise sums
23      LT2AOab2(nu,j,mu,i) += Y1ab(nu,j,mu,i) #elementwise sums
24      gpu_free Yab(mu,i,nu,j) #free temp blocks on GPU
25      gpu_free Y1ab(nu,j,mu,i)
26  ENDDO j
27  ENDDO i
28  #copy results to CPU , free blocks on GPU
29  DO i1
30  DO j1
31      gpu_get LT2AOab1(mu,i1,nu,j1)
32      gpu_get LT2AOab2(nu,j1,mu,i1)
33      gpu_free LT2AOab1(mu,i1,nu,j1)
34      gpu_free LT2AOab2(nu,j1,mu,i1)
35      gpu_free LTAOab(lambda,i1,sigma,j1)
36  ENDDO j1
37  ENDDO i1
38  gpu_free aoint(lambda,mu,sigma,nu)
39  gpu_end
40  #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 host memory. The actual calculations are done by lines 13-27.

5.2 GPU Support in ACES4

The rest of this chapter describes how the support for GPUs in ACES4 was improved by providing automatic memory management and techniques devised to optimally transfer the data between the GPU memory and the host memory.

5.3 Unoptimized 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 host 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 *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 isAsync; // are there any pending operations on device
5 }
```

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*

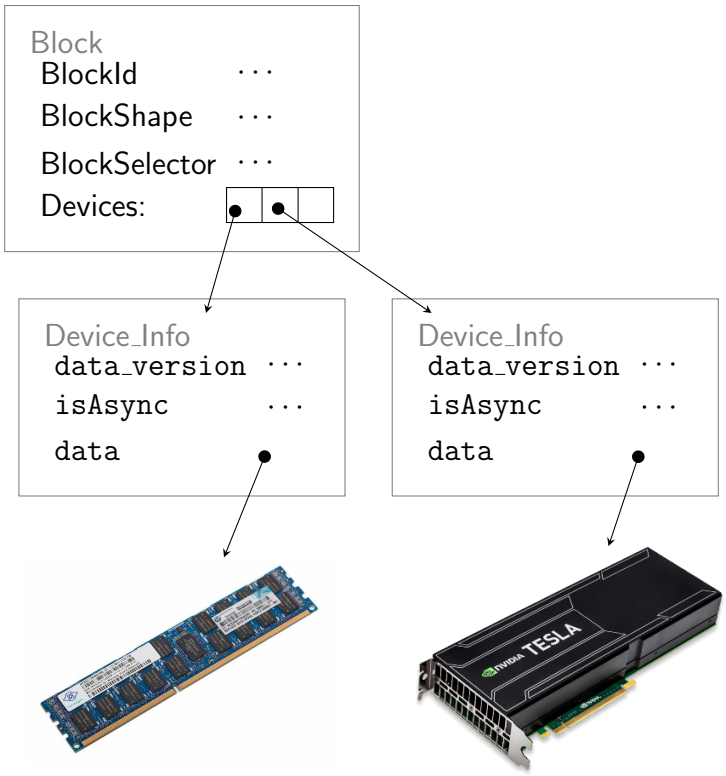

Figure 5-1. `Block` and `Device_Info` structure

```
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 the 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
```

5.4 Optimizing Block Copying

An enormous cost has to be paid [29, 30] 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.

5.4.1 Reducing Block Transfers

The first optimization includes saving unnecessary block synchronizations by exploiting the semantics of SIAL. The runtime has information about whether each of the parameter passed to builtin super instructions is going to be read, written or updated. When each element of a block is modified, it is defined as block writing. Whereas, if only some of the elements of a block are modified, then it is defined as block updating. The runtime also has this information for the user defined super instructions, through the user provided attributes, describing the intent for each parameter. In listing 5.3, we can see for the user defined super instruction `fill_block_sequential` and the attributes `wr`. This denotes that the first parameter passed to `fill_block_sequential` is going to be written and the second is going to be read.

Listing 5.3. SIAL fragment showing user defined super instruction attributes

```
1 sial contract_to_scalar
2   predefined int norb
3   special fill_block_sequential wr
4   special fill_block_cyclic wr
5   special print_block r
6   special print_static_array r
```

```
7   aoindex i = 1:1
8   aoindex j = 1:1
9   temp  a[i,j]
10  ...
```

Using this information about the intent of each block request, 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.

5.4.2 Memory Pinning

The memory allocated on CPU or host memory is pageable by default. GPUs cannot access data on such pageable host memory [5, 31] 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 (host memory), copies the host memory to the temporary pinned memory and then finally transfers the data from pinned memory to GPU device memory.

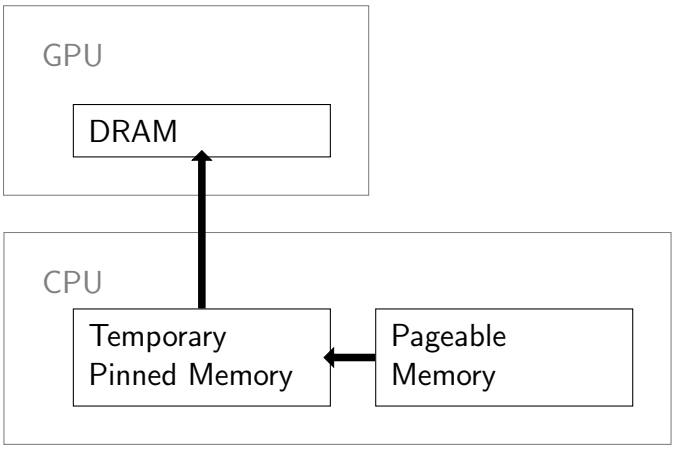

Figure 5-2. `memcpy` w/o memory pinning

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.

5.4.2.1 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 [32–35] to transfer memory, bypassing the CPU. It is reported[18] that, using this, GPU can read host memory at PCI-E speeds.

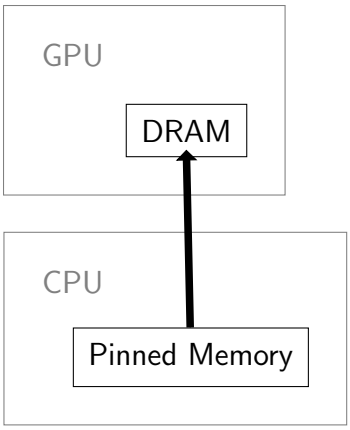

Figure 5-3. `memcpy` with memory pinning

5.4.2.2 Asynchronous `memcpy`

Since the 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 the host and GPU can carry on with the execution of calculation while the blocks are synchronized. This is elaborated in section 5.4.4.1.

5.4.3 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 [36–38] 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 in the SIA to reuse the pinned memory pages.

5.4.3.1 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 an operating system. Extensive work [39] 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.

5.4.3.2 Implementation of Caching Mechanism in the SIA

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 the OS. Once there is enough free memory, a new, bigger, block is created. If there are no blocks in `free_list` and the 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`.

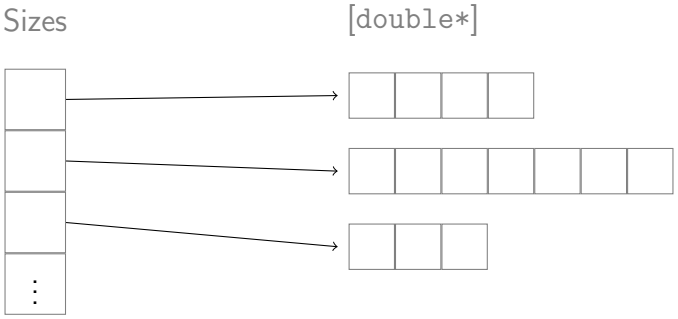

Figure 5-4. Structure of `free_list` map used for mapping block size to list of free blocks.

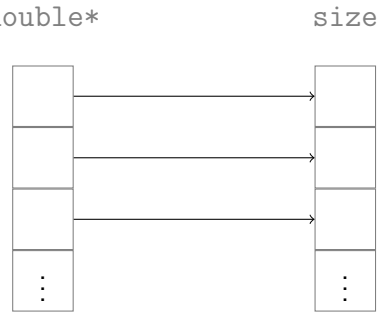

Figure 5-5. Structure of `reverse_lookup` map used for mapping back blocks to actual size of block.

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**

5.4.3.3 Reusing Page Locked Memory Pages Allocated to Temporary Blocks

Temporary blocks in SIAL go out of scope at end of each iteration in a loop. Thus page locked memory allocated to temporary blocks can be reused very efficiently. An effective caching mechanism is implemented by allocating page locked memory for the largest possible block of all arrays involved in a loop. This way, it is guaranteed that page locked memory allocated for a block will be sufficient for any other block of the same array in the loop. This results in paying for page locking just once per array in a loop. Furthermore, this strategy can be effectively combined with the caching mechanism, described in section 5.4.3.2, to reuse page locked memory buffers across loops.

5.4.4 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 [40]. 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.

5.4.4.1 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 [34, 35] 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 multiple streams are created and stored in the module `gpu_super_instructions` during runtime. 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
```

5.5 Optimizing GPU Internode Communication

SIAL makes it possible to work on extremely large arrays, by dividing the array into multiple blocks. Servers send 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.

5.5.1 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 three 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 seven copy operations,

including three similar operations on other side, and a network transfer operation. These operations are represented in Figure 5-6 by black arrows.

5.5.2 MPI Transfers Using DMA

Section 5.4.2 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 [41] 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.

5.5.3 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 [42] 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, server processes in the 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.

5.6 Experiments

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

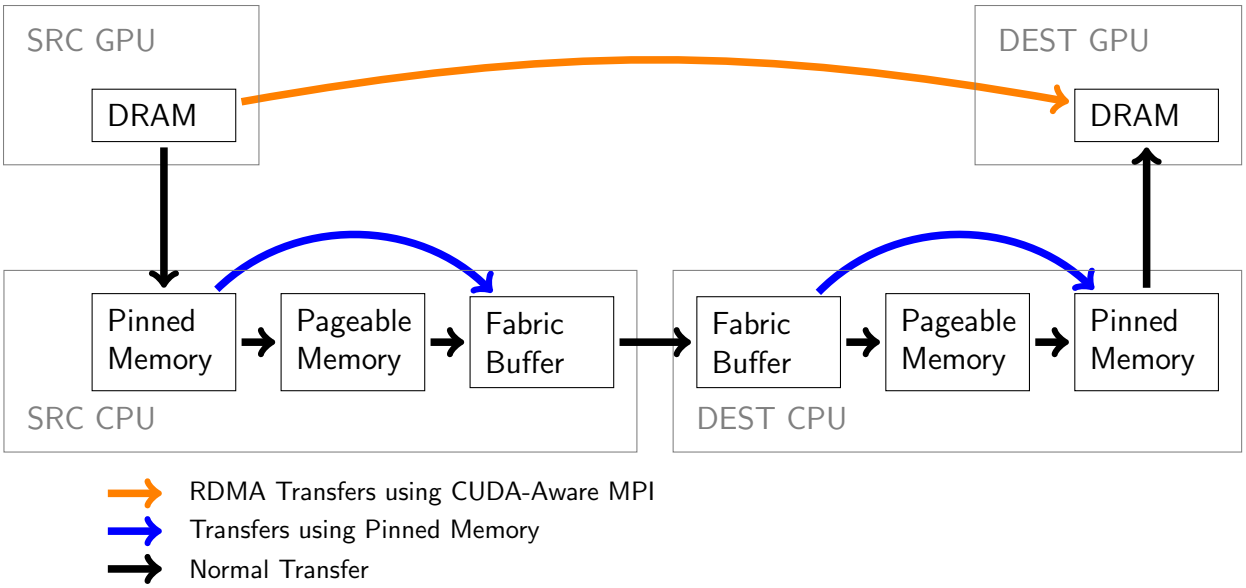

Figure 5-6. RDMA, DMA and normal transmission between two nodes with GPU

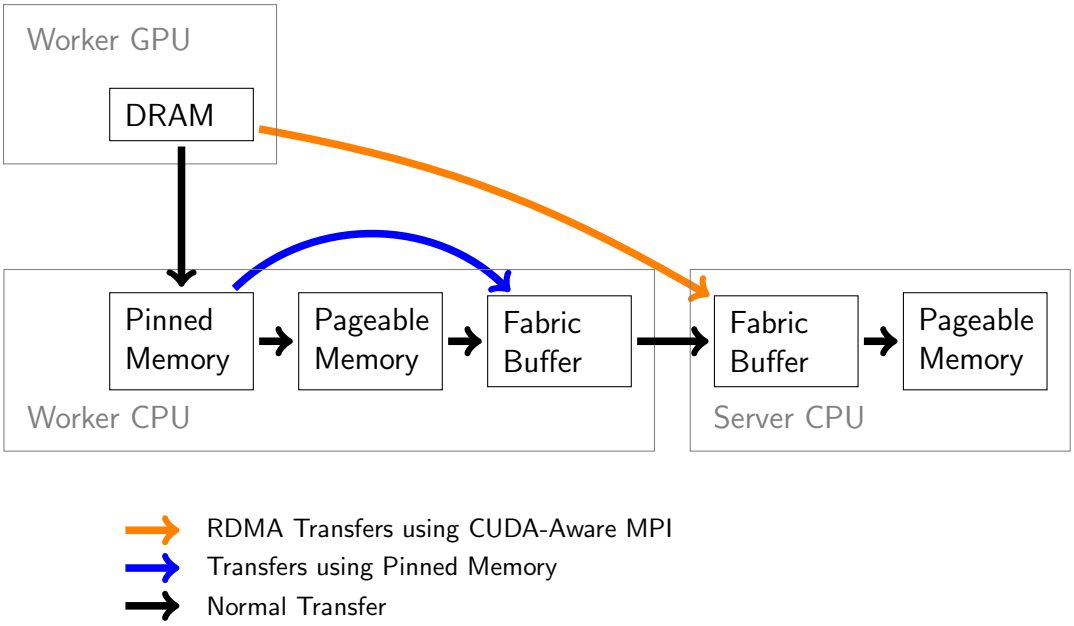

Figure 5-7. RDMA, DMA and normal transmission in SIA

5.6.1 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.

5.6.1.1 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 5-8 presents varying block size vs time to copy block.

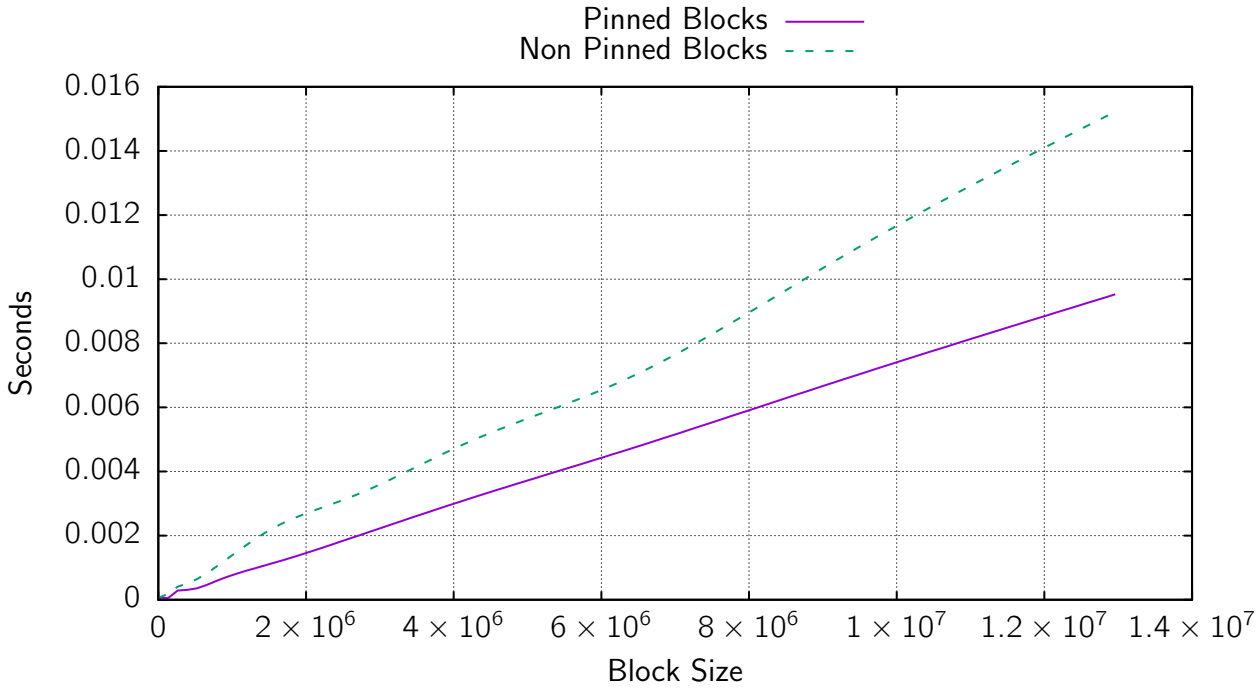

Figure 5-8. Time taken to transfer block to GPU for *pinned* and *non pinned* blocks

It is evident from the plot and table 5-1 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.

5.6.2 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 5-9.

Table 5-1. Block size vs 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

5.6.3 Memory Pinning Overhead

To benchmark the page lock memory allocation vs 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 5-10 and figure 5-11 by plotting time taken by page locked operations vs non-page locked operations.

5.6.3.1 `alloc`

Table 5-2 describes the time taken for page locked memory allocation and non-page locked memory allocation.

It is clear from the table 5-2 and the figure 5-10 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.

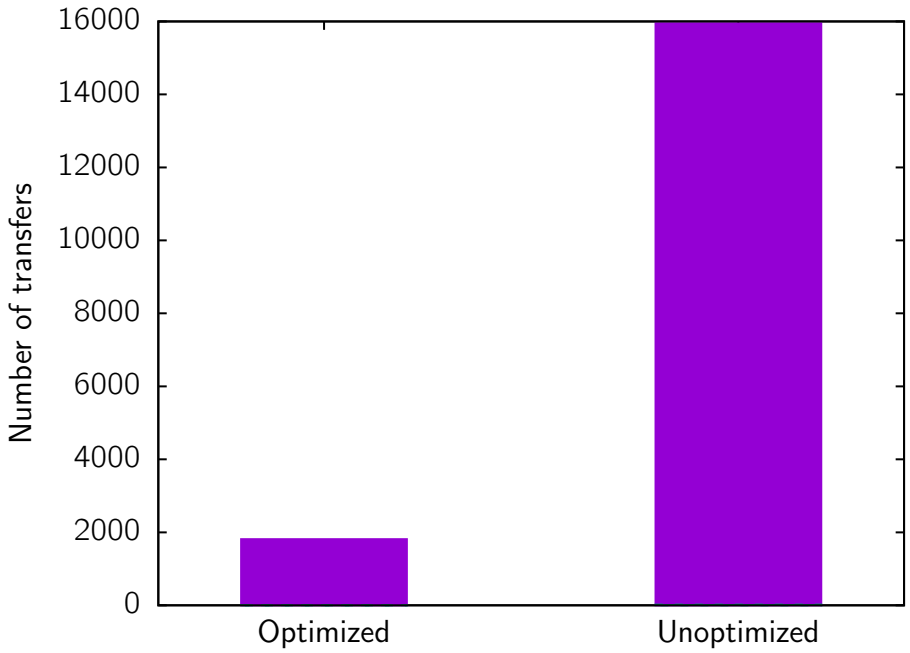

Figure 5-9. Optimized vs unoptimized block transfers for `rccsd_rhf.sialx`

Table 5-2. Block size vs 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

5.6.3.2 `free`

A similar test was performed to benchmark page locked and non-page locked memory deallocation. Table 5-3 describes the time taken by the deallocation operation vs 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.

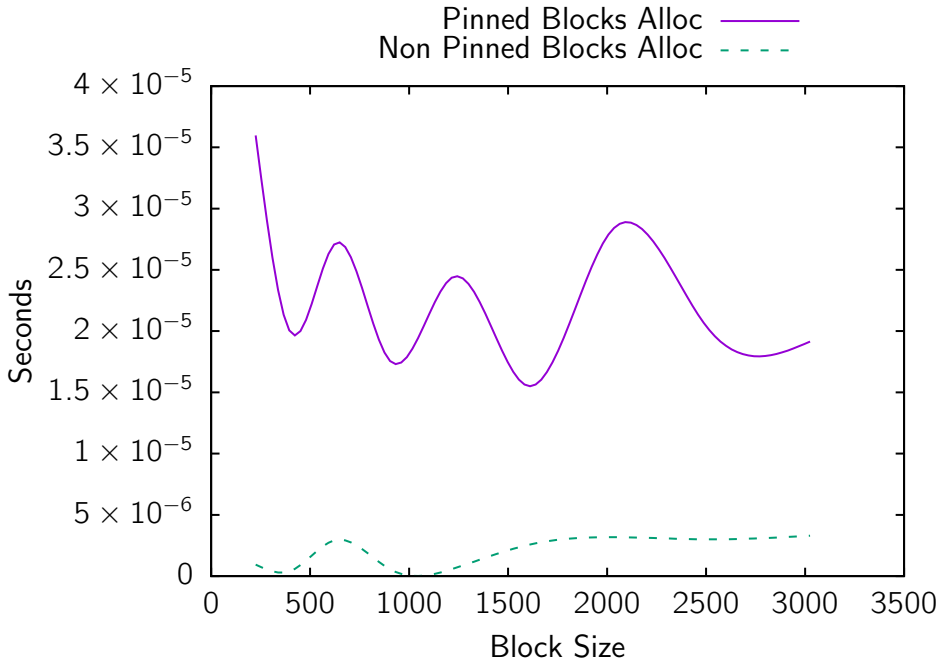

Figure 5-10. Time taken for pinned and non pinned memory allocation

Table 5-3. Block size vs 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

Page locked memory deallocation is around 100 times costlier than non-page locked memory deallocation.

5.6.4 RDMA

To benchmark RDMA, CUDA aware implementation of MPI was used. The size of the block was varied vs time taken to complete the MPI transfer functions. The results for `GET` operation are presented in figure 5-12.

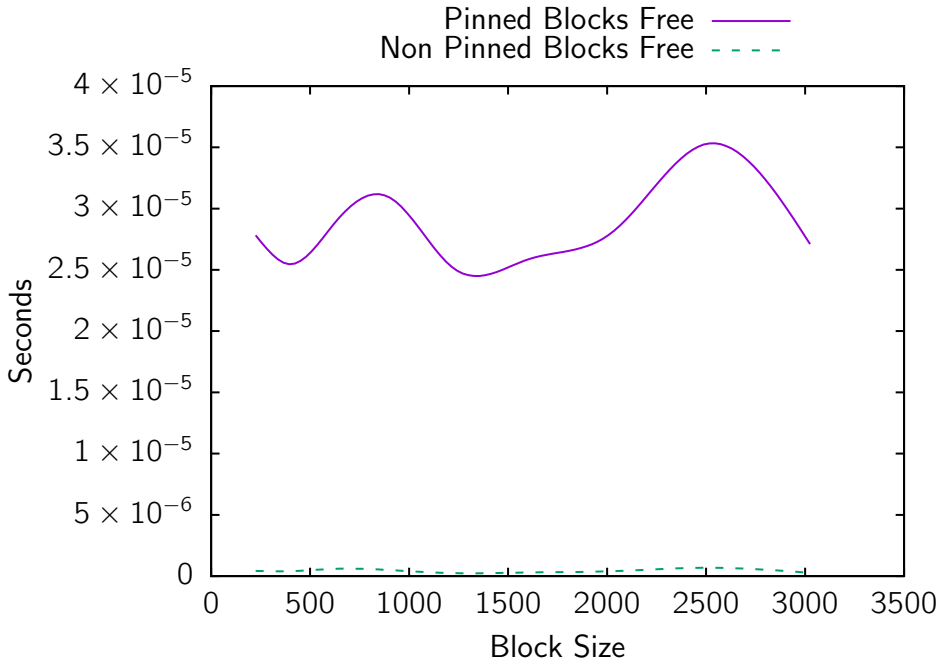

Figure 5-11. Time taken for pinned and non pinned memory de-allocation

5.6.4.1 GET

The highest amount of time is taken by passing host 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 host memory, even without using GPU, DMA is invoked by MPI implementation for transfer of data from network fabric to host memory.

5.6.4.2 PUT

There is not much of difference in time in PUT operation, as presented in figure 5-13. The time taken by passing host memory address, page locked host memory address and GPU memory address to MPI function varies slightly as the size of block increases.

5.6.4.3 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

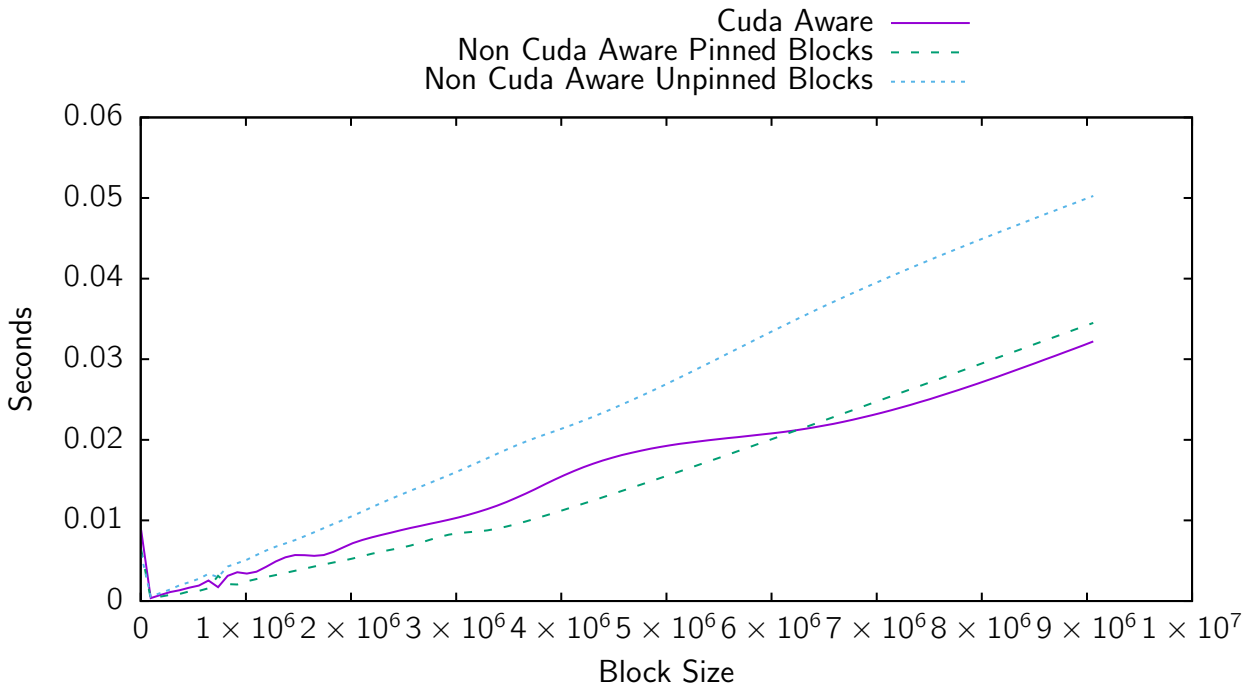

Figure 5-12. Time taken to transfer GPU and CPU buffer in `GET` operation

executed on GPU without requiring the data to be copied to host memory. This saves two GPU and host memory copy operations and thus get a substantial improvement in overall operation. The time spent on overall operation is plotted in figure 5-14.

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

5.6.5 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.

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

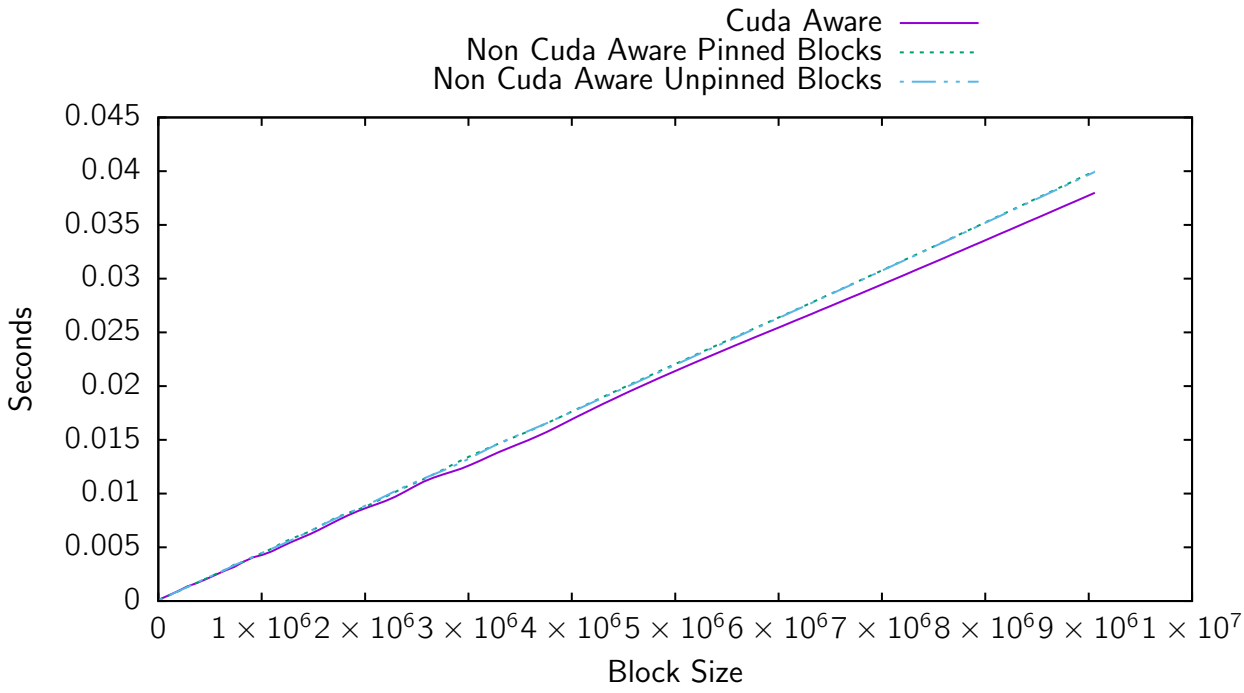

Figure 5-13. Time taken to transfer GPU and CPU buffer in PUT operation

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 5-4 presents the hit ratio for given calculation programs.

Table 5-4. 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

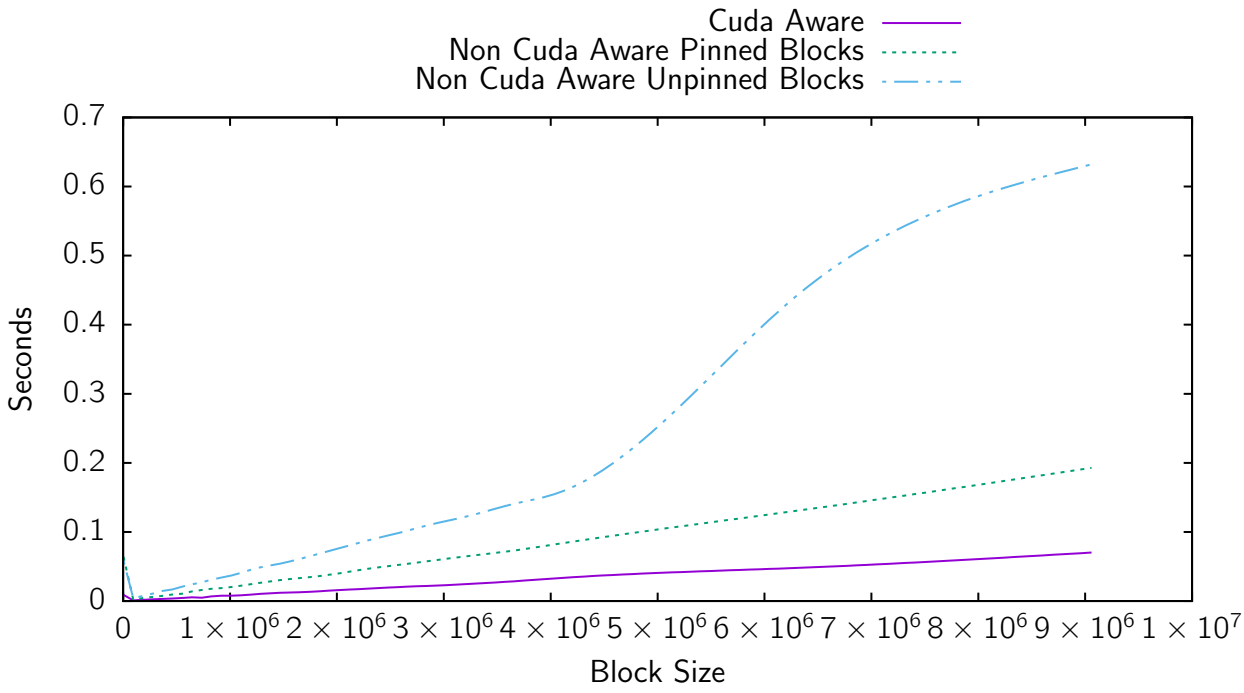

Figure 5-14. Total MPI transfer compared to CUDA Aware MPI transfer

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 5-15.

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 5-5.

Table 5-5. Page locked cached vs page locked uncached vs 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.

5.6.6 Overall Improvement

To determine the overall improvement in the total runtime of the calculation, an experiment was conducted by executing CCSD calculation on $C_{12}H_{10}$ molecule in a CC-PVDZ basis. To study effect of each technique used, the same calculation was executed using different version of aces4, with some features turned off. Results are presented in figure 5-16.

The version used was without page locking memory pages and without using CUDA aware MPI. In this case we see some time spent in memory copy operation. The next version, using page locked memory, reduces the time spend in block copy into half. But, the overall time increases due to time spend in expensive page locking operation. In the next version, the block copying time was eliminated using CUDA aware MPI. But this version doesn't page lock memory pages, so any optimizations which need page locked memory pages will not work. In the next version, time spent in memory copying is eliminated using CUDA aware MPI, as well as, the memory is page locked, hence optimization based on page locking can be used. In the final version, caching of the page locked memory is used along with CUDA aware MPI. Here, the time spent in copy operation is eliminated, as well as, the time spent in page locking is also minimized. The overall time is comparable to CUDA aware with pageable memory version, but in the last version all the optimizations based on page locking can be exploited.

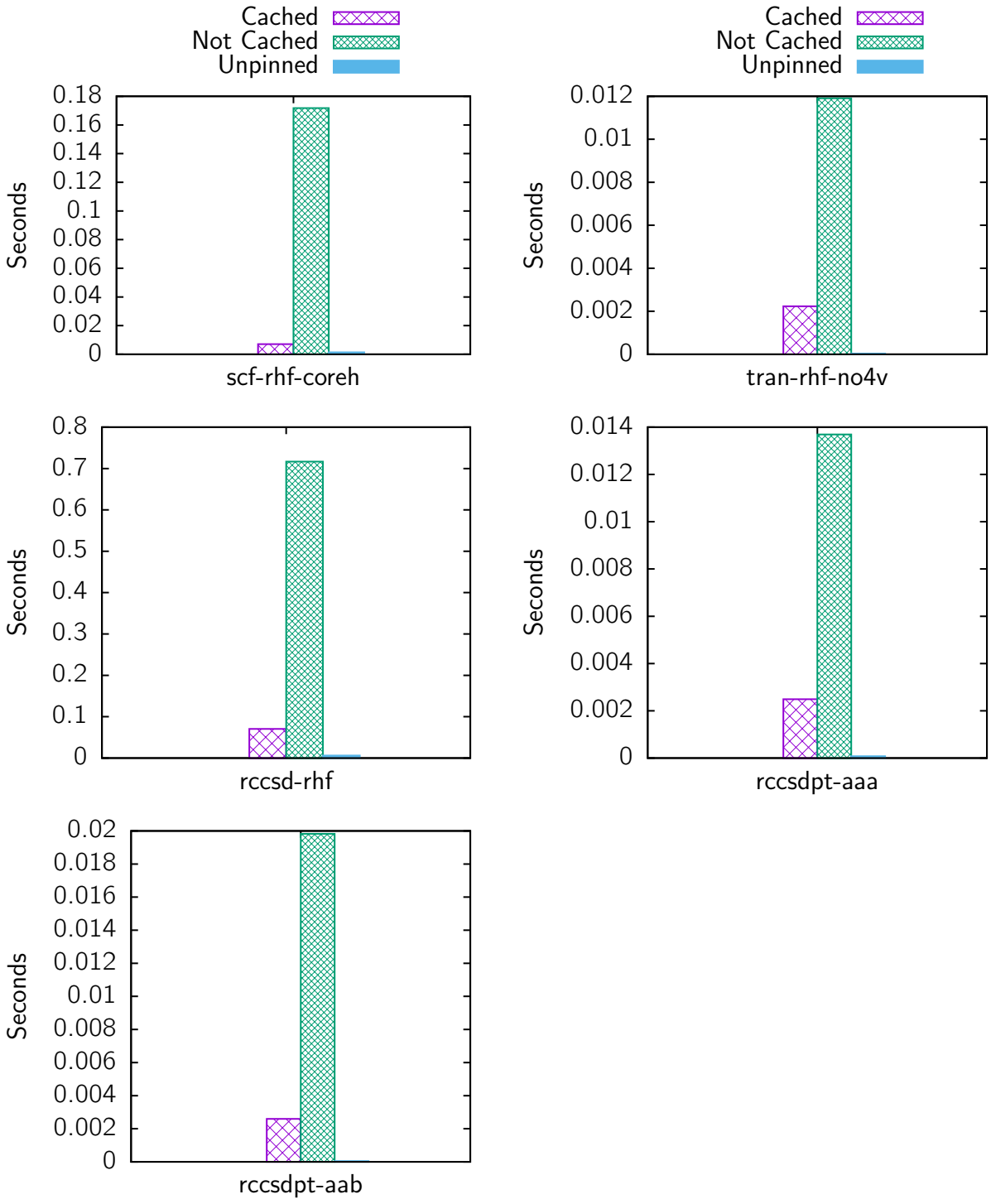

Figure 5-15. Page locked cached vs page locked uncached vs non-page locked allocation and deallocation times

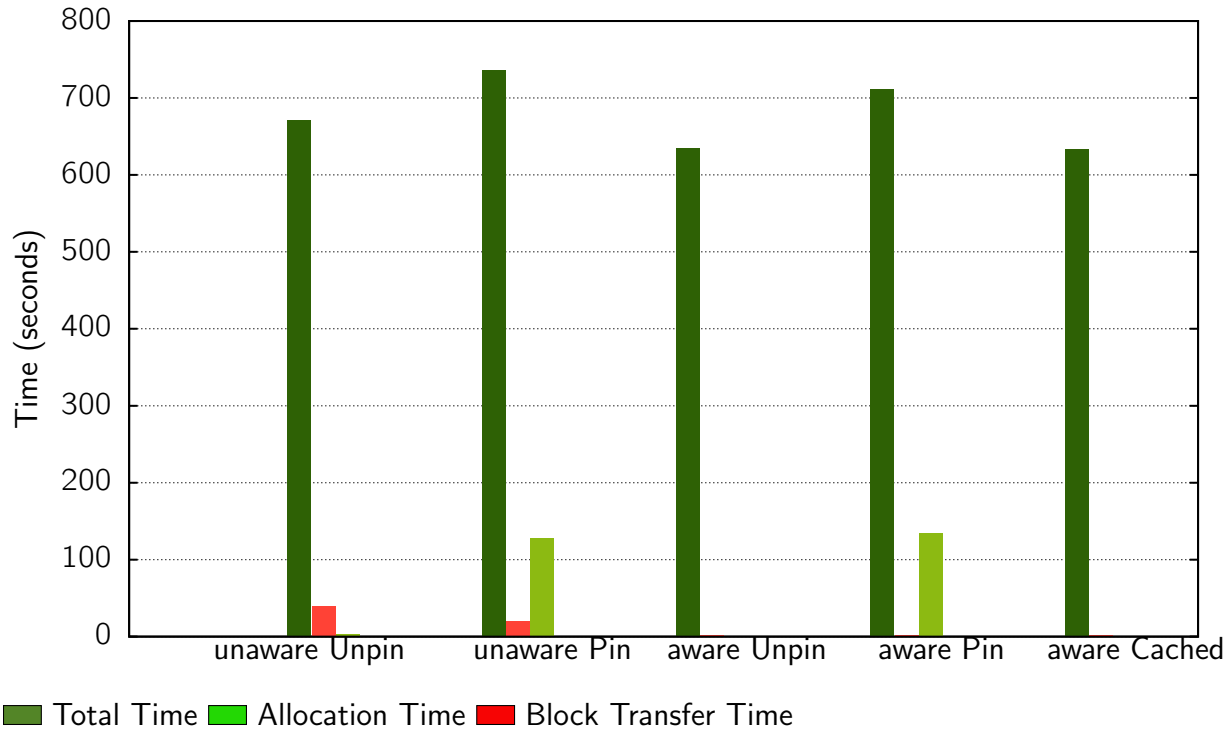

Figure 5-16. Overall improvement in runtime in CCSD calculation on $C_{12}H_{10}$ molecule in a CC-PVDZ basis

CHAPTER 6
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 technique of data block prefetching to SIA. The 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 4.3.1, can be used as a metric to evaluate the performance of prefetching. As seen in section 4.3.5.1, prefetching helps in reducing overall `wait_time` when most of the `wait_time` is due to blocking for data block transfer over the network. Also, the total number of workers participating in the computation should be taken in consideration. If the number of workers is very high then the index range will be divided among high number of workers and each individual worker will not get sufficient index range for prefetching.

It was found that the optimal number of blocks to prefetch in most cases is one. This result is not sensitive to the block size.

Executing super instructions on GPU can speed up the computation when block sizes are large enough. Transferring blocks between a GPU and host memory is expensive but it can be avoided or reduced by directly transferring blocks from the GPU memory. Page locked memory can improve memory transfer speed by invoking DMA and bypassing CPU. But allocating 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.

Several further improvements to GPU utilization in ACES4 can be envisioned. First, more super instructions on GPUs should be implemented along with finding a way to easily port user defined super instructions to GPUs. Another optimization would be to exploit the asynchronous memory synchronization between the GPU memory and the host memory by

looking ahead for instructions which need the memory to be transferred and initiating the asynchronous transfer as soon as data is ready. Feasibility of having access to GPUs on servers and implementing RDMA to transfer memory blocks between workers and servers can also be explored. Lastly, implementing a more advanced caching mechanism for page locked memory blocks in order to have a more efficient caching mechanism.

APPENDIX
INPUT FILES AND SIAL PROGRAMS USED FOR BECHMARKING

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

Table A-1. HiperGator 2 spec sheet

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

Table A-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

Table A-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

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

Table A-4. HiperGator 2 node interconnect specification

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

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

```
1  sial put_test
2    predefined int norb
3    aoindex i = 1:norb
4    aoindex j = 1:norb
5    distributed a[i,j]
6    distributed b[i,j]
7    temp t[i,j]
8
9    scalar x
10
11   print "starting loop"
12   pardo i
13     do j
14       t[i,j] = (scalar)((i-1)*norb + (j-1)) + 1.0
15       put a[i,j] = t[i,j]
16     enddo j
17   endpardo i
18
19   sip_barrier
20
21   pardo i
22     do j
23     get a[i,j]
24     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
```

Listing A.3 presents the ZMAT file used to generate input data file for CCSD calculation on $C_{12}H_{10}$ molecule in CC-PVDZ basis.

Listing A.3. ZMAT file for CCSD calculation on $C_{12}H_{10}$ molecule in CC-PVDZ basis.

```
C12H10(BP)
C       0.000000     0.000000     1.485300
C       0.000000    -1.305500     0.831600
H       0.000000    -2.201200     1.348700
C       0.000000     1.305500     0.831600
H       0.000000     2.201200     1.348700
C       0.000000    -1.392100    -0.725900
H       0.000000    -2.287800    -1.243000
C       0.000000     1.392100    -0.725900
H       0.000000     2.287800    -1.243000
C       0.000000     0.000000    -1.524200
H       0.000000     0.000000    -2.263900
H       0.000000     0.000000     6.563900
C       0.000000     0.000000     5.824200
```

```
H        0.000000     2.287800     5.543000
C        0.000000     1.392100     5.025900
H        0.000000    -2.287800     5.543000
C        0.000000    -1.392100     5.025900
H        0.000000     2.201200     2.951300
C        0.000000     1.305500     3.468400
H        0.000000    -2.201200     2.951300
C        0.000000    -1.305500     3.468400
C        0.000000     0.000000     2.814700

*ACES2(CALC=CCSD,BASIS=CC-PVDZ,
#ref=uhf,mult=2,charge=-1
cc_maxcyc=3
sym=none,,guess=core,damp_type=davidson)

*SIP
MAXMEM=2500
SIAL_PROGRAM=scf_rhf.siox
SIAL_PROGRAM=tran_uhf_no4v.siox
SIAL_PROGRAM=rccsd_rhf.siox
```

Listing A.4 presents the ZMAT file used to generate input data file for CCSD calculation on C_{20} molecule in AUG-CC-PVDZ basis.

Listing A.4. ZMAT file for CCSD calculation on C_{20} molecule in AUG-CC-PVDZ basis.

```
C20
C      -0.52468348   -1.08750645    0.41975795
C      -1.19641536    0.16294590    0.41975742
C      -0.21474209    1.18821243    0.41975906
```

```
C      1.06369792    0.57140919    0.41975731
C      0.87214374   -0.83506149    0.41975636
C     -2.52396375    0.34375174    0.01480088
C     -1.10687386   -2.29420637    0.01479373
C      1.83987722   -1.76164863    0.01479410
C      1.16971329   -2.98963502   -0.21728013
C      3.03760663   -1.03879605   -0.21727732
C     -2.48185146   -2.03631099   -0.21727647
C     -0.04928196   -3.20994040   -0.21727976
C     -2.70358159    1.73112574   -0.21727430
C     -3.06806465   -0.94505601   -0.21727526
C     -0.45302044    2.50665606    0.01479516
C      2.24398142    1.20544799    0.01479749
C      3.20477352    0.18861528   -0.21727658
C      1.92662556    2.56792881   -0.21727743
C     -1.84688649    2.62586429   -0.21727441
C      0.81094572    3.10620594   -0.21727674

*ACES2
BASIS=AUG-CC-PVDZ
#GUESS=ATOMIC
GUESS=CORE
calc=ccsd
damp_type=davidson
CALC=CCSD,SPHERICAL=on

*SIP
MAXMEM=2500
SIAL_PROGRAM=scf_rhf.siox
```

```
34 SIAL_PROGRAM=tran_rhf_no4v.siox
35 SIAL_PROGRAM=rccsd_rhf.siox
36 SIAL_PROGRAM=rlambda_rhf.siox
```

Git commit hashes for versions of source code used for conducting experiments:

Enhancement	Hash
Prefetching	`caaf15eca1c966c0a38696edbc8d5b959ed66d08`
GPU Support	`bd901e29baf3344fbe261f9f4c2c8712283cabb9`

REFERENCES

[1] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton, "A block-oriented language and runtime system for tensor algebra with very large arrays," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.3

[2] S. J. F., G. Jrgen, W. J. D., L. W. J., and B. R. J., "The aces ii program system," *International Journal of Quantum Chemistry*, vol. 44, no. 26, pp. 879–894. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.560440876

[3] D. Erik, L. V. F., P. Ajith, P. M. J., S. B. A., and B. R. J., "Software design of aces iii with the super instruction architecture," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 1, no. 6, pp. 895–901. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.77

[4] N. Jindal, V. Lotrich, E. Deumens, and B. A. Sanders, "Exploiting gpus with the super instruction architecture," *Int. J. Parallel Program.*, vol. 44, no. 2, pp. 309–324, Apr. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0319-4

[5] NvidiaStaff. (2012) How to optimize data transfer in cuda c/c++ — nvidia developer blog. [Online]. Available: https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

[6] OpenACC-standard.org. (2018) Openacc. [Online]. Available: https://www.openacc.org

[7] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "Openmp for accelerators," in *Proceedings of the 7th International Conference on OpenMP in the Petascale Era*, ser. IWOMP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 108–121. [Online]. Available: http://dl.acm.org/citation.cfm?id=2023025.2023037

[8] S. Lee and R. Eigenmann, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.36

[9] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 78–90, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2010.62

[10] K. Bhaskaran-Nair, W. Ma, S. Krishnamoorthy, O. Villa, H. J. J. van Dam, E. Apr, and K. Kowalski, "Noniterative multireference coupled cluster methods on heterogeneous cpugpu systems," *Journal of Chemical Theory and Computation*, vol. 9, no. 4, pp. 1949–1957, 2013, pMID: 26583545. [Online]. Available: https://doi.org/10.1021/ct301130u

[11] A. E. DePrince and J. R. Hammond, "Coupled cluster theory on graphics processing units i. the coupled cluster doubles method," *Journal of Chemical Theory and Computation*, vol. 7, no. 5, pp. 1287–1295, 2011, pMID: 26610123. [Online]. Available: https://doi.org/10.1021/ct100584w

[12] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski, "Gpu-based implementations of the noniterative regularized-ccsd(t) corrections: Applications to strongly correlated systems," *Journal of Chemical Theory and Computation*, vol. 7, no. 5, pp. 1316–1327, 2011, pMID: 26610126. [Online]. Available: https://doi.org/10.1021/ct1007247

[13] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, "Optimizing tensor contraction expressions for hybrid cpu-gpu execution," *Cluster Computing*, vol. 16, no. 1, pp. 131–155, Mar 2013. [Online]. Available: https://doi.org/10.1007/s10586-011-0179-2

[14] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Comput.*, vol. 38, no. 3, pp. 157–174, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.09.001

[15] Y. Yan, M. Grossman, and V. Sarkar, "Jcuda: A programmer-friendly interface for accelerating java programs with cuda," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 887–899. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03869-3_82

[16] A. Sidelnik, M. Garzarn, D. Padua, and B. L Chamberlain, "Using the high productivity language chapel to target gpgpu architectures," University of Illinois, Tech. Rep., 01 2011.

[17] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 46–51. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513901

[18] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, ser. DaMoN '12. New York, NY, USA: ACM, 2012, pp. 55–62. [Online]. Available: http://doi.acm.org/10.1145/2236584.2236592

[19] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of mellanox/nvidia gpudirect over infiniband—a new model for gpu to gpu communications," *Computer Science - Research and Development*, vol. 26, no. 3, pp. 267–273, Jun 2011. [Online]. Available: https://doi.org/10.1007/s00450-011-0157-1

[20] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "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*, ser. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 80–89. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2013.17

[21] W. Anacker and C. Ping Wang, "Performance evaluation of computing systems with memory hierarchies," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 764 – 773, 01 1968.

[22] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982. [Online]. Available: http://doi.acm.org/10.1145/356887.356892

[23] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/358923.358939

[24] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978. [Online]. Available: http://dx.doi.org/10.1109/C-M.1978.218016

[25] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Ph.D. dissertation, Rice University, Houston, TX, USA, 1989, aAI9012855.

[26] R. H. Patterson and G. A. Gibson, "Exposing i/o concurrency with informed prefetching," in *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems*, ser. PDIS '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 7–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=381992.383614

[27] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing - Volume 01*, ser. ICPP '93. Washington, DC, USA: IEEE Computer Society, 1993, pp. 56–63. [Online]. Available: http://dx.doi.org/10.1109/ICPP.1993.92

[28] S. Bhatia, E. Varki, and A. Merchant, "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*, ser. MASCOTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 89–98. [Online]. Available: http://dx.doi.org/10.1109/MASCOTS.2010.18

[29] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 94–103. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735706

[30] M. Boyer. (2018) Memory transfer overhead. [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html

[31] NvidiaStaff. (2018) Programming guide cuda. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[32] ——. (2018) Standard dma transfer: Gpudirect rdma: Cuda toolkit documentation. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html#standard-dma-transfer-example-sequence

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

[34] NvidiaStaff. (2013) Nvidia tesla gpu accelerators datasheet. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/Tesla-KSeries-Overview-LR.pdf

[35] ——. (2013) Nvidia tesla k series gpu accelerators datasheet. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf

[36] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron, "Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5160984

[37] M. Boyer. (2018) Memory management overhead. [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_management_overhead.html

[38] ——. (2018) Choosing between pinned and non-pinned memory. [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html

[39] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proceedings of the International Workshop on Memory Management*, ser. IWMM '95. London, UK, UK: Springer-Verlag, 1995, pp. 1–116. [Online]. Available: http://dl.acm.org/citation.cfm?id=645647.664690

[40] NvidiaStaff. (2012) How to overlap data transfers in cuda c/c++. [Online]. Available: https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

[41] ——. (2013) An introduction to cuda-aware mpi. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[42] openmpi.org. (2018) Faq: Running cuda-aware open-mpi. [Online]. Available: https://www.open-mpi.org/faq/?category=runcuda#mpi-apis-cuda

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.