ENHANCEMENTS TO LOOPING CONSTRUCTS BY PREFETCHING AND EXPLOITING GPU

By

ANURAG PESHNE

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

UNIVERSITY OF FLORIDA

2018

© 2018 Anurag Peshne

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 entire course of my thesis work. She provided new ideas whenever I got stuck and reminded me of finer points which I missed. And most importantly she introduced me to a whole new area of computation which I wouldn't have known to exist if I didn't get a chance to work on this project.

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

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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

ENHANCEMENTS TO LOOPING CONSTRUCTS BY PREFETCHING AND EXPLOITING GPU

By

Anurag Peshne

August 2018

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

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

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

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

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

CHAPTER 1
INTRODUCTION TO THESIS

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

1.1 Introduction to Issues with Working with GPUs

In ACESIII, the previous version of ACES, programmer had to explicitly deal with managing the memory transfer between GPU and CPU and marking regions well suited for execution on GPU. In ACES4, managing memory transfer is done automatically by the runtime. This is implemented by tracking block changes using version numbers.

Still there is need for SIAL programmer to decide which portion of the SIAL code is well suited for execution on GPU. Transferring data between GPU and CPU is expensive and can dominate the total time spent in computation. Hence this is not a trivial decision to make since the programmer has to minimize time spent in total memory transfers. This in turn depends on various factors such as size of the block and number of instructions in the code block supported on GPU. Judging based on size of block is itself difficult because it depends on the type of GPU available at runtime. Larger blocks need more time to transfer between GPU and CPU, but at same time the difference in time taken to operate on blocks by GPU and CPU grows exponentially with the size of the block. And lastly, same programs are used to calculate different results by supplying different data files. A portion of code which executed faster on GPU for certain block size may in fact execute slower than on CPU for smaller block size if the gain in time by executing on GPU cannot compensate for the transfer time.

1.2 Introduction to Issue of Data Transfer from Server

In SIAL, looping constructs are the only way to work with individual blocks. Typical work flow of SIAL includes requesting a block of larger array from server, processing the block, compute resulting block and send it back to the server. Though the operation of requesting block from server is non blocking, subsequent operations in the loop block until Message Passing Interface (MPI) transfer is completed. To amortize the cost of network transfer, prefetching has been implemented which does the transfer concurrently with the block processing. By prefetching the anticipated blocks that would be needed in next iteration of loop, the wait time for block can be reduced and in some cases completely eliminated.

1.3 Organization of the Thesis

Chapter 2 presents related literature regarding efforts made to exploit GPU in SIA as well as other programming models and the technique of prefetching to hide latency in accessing data. Then an introduction to background of this thesis, including architecture and implementation of SIA and ACES4, is presented in chapter 3. Chapter 4 describes the problem and solution of optimizing use of GPU to execute SIAL code; the design and implementation of eager-pushing of blocks and dynamic determination of suitable code block for execution on GPU is presented in chapter 5. Chapter 6 states the results of the benchmarks and experiments. And finally, chapter 7 presents conclusion of this thesis and future works.

CHAPTER 2
RELATED WORK

We discuss work done by others in using GPU for scientific calculations using several methods such as directive based GPU programming models, using CUDA in higher level programming language and use of GPU in previous versions of ACES and work done using in prefetching to hide latency in accessing data. We also look at work done in general idea of prefetching such as non blocking fetch operation, techniques used to determine optimal prefetching parameters such as number of blocks to prefetch, prefetch cache size.

2.1 GPGPU

This section discuss previous work done on General Purpose computing on Graphics Processing Units.

2.1.1 Directive Based GPU programming

There are several attempts in area of directive based programming models. These attempts include OpenACC (1), OpenMP for accelerator (2), and less updated attempts including OpenMPC (3) and hiCuda (4). The approach taken by these models is to make a implementation GPU ready by placing directives which make execution of loops parallel. In contrast, SIA exploits coarse grained concurrency using super instructions and super numbers or blocks by executing super instruction on GPU. These super instructions in turn exploit fine grained concurrency and can make use of above models to make an existing CPU implementation GPU ready by inserting suitable directives or have a completely rewritten low level CUDA implementation. Thus these models work *with* the previously mentioned models rather than against them.

2.1.2 GPU in Computational Chemistry

There are implementations of coupled cluster methods in computational chemistry on GPUs reported (5)(6)(7). 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 implementation of contraction operators on a GPU by Ma et al. (8) generates CUDA code to directly implement the contractions by optimizing for the particular order of indices in the contractions. This, as compared to using permutations and then two dimensional matrix-matrix multiplication as implemented in SIA, should achieve better performance for a contraction. These specialized CUDA contraction implementations were added to NWChem, a computational chemistry package, and considerable speedups are reported for CCSD(T) calculation on CPU/GPU hybrid systems. Such optimized operators can be incorporated into SIA and be provided as built in block super instruction for contraction.

2.1.3 GPU Programming in Other High Level Programming Languages

Several attempts for support for GPU in general purpose high level programming languages have been made. A few examples include PyCUDA (9), jCUDA (10) and Chapel (11). These toolkits gives an interface to GPU from high level programming languages such as Python and Java but programmer still needs to deal with low level GPU book keeping such as memory management and defining kernels. Whereas in SIA, once the superinstructions are defined, runtime can jump between GPU and CPU implementation transparently to the domain programmer, the runtime takes care of memory management and GPU implementations of common operations on blocks are build into the runtime.

2.1.4 GPU in Previous Versions of ACES

There has been work done previously to exploit GPU in Super Instruction Architecture by Jindal et al (12) in ACESIII. This implementation required programmer to mark a block of code to be executed on GPU using directives `gpu_on` and `gpu_off` and manually manage memory transfer to and from GPU. Executing operations on small blocks on GPU can actually be slower than executing on CPU due to time spent in memory transfer and insignificant gain in speed. Thus it is better to execute small blocks with 2 indices on CPU and larger block with

4 and up indices on GPU. It was left to the SIAL programmer to take this call on which block is large enough so that benefit of executing it on GPU becomes significant.

2.2 Prefetching

Prefetching is an old technique (13)(14)(15) used to get data up in memory hierarchy before they are actually needed by processor with the aim to hide the data access latency due to difference in speed to access data. A variety of approaches, including hardware prefetching and caching, compiler techniques, pre-execution and runtime execution, have been implemented in high performance computing.

Hardware prefetching techniques including techniques such as transferring separate cache blocks (16) were implemented much before software techniques were implemented. Porterfield (17) introduced 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.

Prefetching is also exploited in area of disk IO apart from feeding processor cache from memory. Patterson et al (18) implemented informed prefetching mechanism for IO intensive application to exploit highly parallel IO systems. The mechanism depends on disclosure of future access dynamically.

Dahlgren and Stenstrom (19) developed an algorithm to determine the number of memory blocks to prefetch. They proposed *adaptive sequential prefetching* policy, which allows the number of blocks to prefetch, K to vary during runtime of the program to reflect spatial locality shown by the program. K is varied by calculating *prefetch efficiency* metric which is defined as the ratio of useful prefetched blocks to the total number of prefetched blocks. If the prefetch efficiency exceeds a threshold then K is incremented and decremented if it drops below a lower threshold. The value of K can reach 0, effectively disabling prefetching.

There had been work done by Bhatia et al (20) to determine size of 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 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 miss on both of caches then the algorithm leaves the size unchanged. To determine if the cache size is too large, the eviction cache is observed and if eviction cache receives no hits then the size of cache is decremented.

In SIA, blocks which are needed to be prefetched can be predicted with more accuracy than memory blocks. Further exact sequence in which blocks are needed is also known. This makes prefetching in SIA free of few of the problems faced in general memory block prefetching discussed in above section. Similar to the problems faced in general memory prefetching, it is difficult to predict how many blocks to prefetch. But once block is prefetched and computed on, the block can be safely evicted from memory. The problem of how many blocks to prefetch is related to how much memory should be allocated for prefetching.

CHAPTER 3
BACKGROUND: SUPER INSTRUCTION ARCHITECTURE

This chapter introduces the SIA, ACES4, an implementation of SIA, block based programming, the design of worker and server, SIAL, parallel looping constructs and design of GPU implementation.

3.1 SIA

The Super Instruction Architecture is a special purpose, domain agnostic, parallel programming framework which is engineered for solving very large computation expressed in terms of multi-dimensional arrays and super instructions to operate on them. Since these multi-dimensional arrays can be too large to hold in physical memory of a single processor, they are broken into blocks or super numbers. These super numbers are input to special instructions written to operate on blocks instead of floating point numbers and hence called as super instructions.

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

3.2 Architecture of SIA

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

SIA supports arrays of size greater than combined memory of all processors involved in computation by providing 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 — workers are responsible for actual execution of instructions on the blocks, while servers make sure blocks are served to and from workers. The division of number of servers versus number of workers is chosen by SIA but can be overridden by passing command line arguments.

3.3 SIAL

SIAL is a programming language provided for expressing problems of target domain. The idea behind SIAL is to keep the domain problem separate from platform and computing problem. SIAL programs are written by the domain experts whereas the intricacies involved in execution of SIAL programs, like distribution of data, parallel execution, memory management, runtime optimizations, are handled by computing experts.

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

As mentioned above, domain experts can write their own domain specific instructions which take in single or multiple blocks and output a resulting block. These instructions can be written in C, C++ or Fortran. Since these languages are much more closer to hardware, these super instructions can be written in a very optimized way. Further, this facility can be exploited

to port the super instructions to other computing devices such as GPU by writing these super instructions using Nvidia CUDA.

3.3.1 SIAL Interpreter

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

3.4 Block Structure

As described above, instead of processing data as a single floating point number, SIA processes blocks of floating point numbers. These blocks are chunks of even larger multidimensional arrays. This is represented inside SIAL interpreter using `Block` class. Since SIA supports heterogeneous computing using other computing devices such as GPU and GPU has their own device memory which is separate from main processor memory, there is a facility in `Block` class to represent the block memory in other computing device. Along with member attributes which represent block metadata and member functions which act on the block, the Block class has pointers to memory location on each computation unit: CPU, GPU and support for more computing device such as Intel Xeon Phi.

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

3.5 Executing Super Instructions on GPU

There are two ways in which GPU are exploited in SIA to obtain high concurrency. First the super instructions can be written in close to device hardware Cuda programming language. These instructions can make use of low level hardware features and domain knowledge to fine tune the implementation. Secondly some of the general purpose block operations such as matrix multiplication, addition, scaling, tensor contraction can be implemented for GPU in the interpreter itself. These operations can be included 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 size of input and output.

3.6 Overview of the ACES

ACES4 is an implementation of SIA for chemical computation. It has been executed on a variety of architectures, but is especially optimized to enable calculations on leadership class supercomputers. The chemical computation done using ACES4 uses data of high dimension. A typical calculation in this domain takes as input the geometry of a molecular system and a choice of single particle orbitals as basis to expand the many-electron quantum-mechanical wave function. The complex algorithms which produce properties of the molecular system can easily require arrays of double precision floating point of size several hundred Gigabytes. Of these arrays, at least three need rapid access and are usually stored in RAM, the rest that are used less frequently can be stored on disk.

CHAPTER 4
BLOCK PREFETCHING

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

4.1 Background

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

Algorithm 4.1. *Processing large array*

1: **loop** ▷ *SIAL do/pardo loop*
2: $GET\ A[i,j]$ ▷ *Request data from server asynchronously*
3: $GET\ B[j,k]$ ▷ *Network bound*
4: $t\_result[j,k] \leftarrow A[i,j] \times B[j,k]$ ▷ *Compute bound*
5: $CALL\ compute\_fun(t\_result[j,k])$ ▷ *Compute bound*
6: $PUT\ AB[i,k] \leftarrow t\_result[i,k]$ ▷ *Network bound*
7: **end loop**

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

4.2 Implementation of Prefetching

4.2.1 pardo Loop Implementation

While the `do` loop which iterates over the indices one by one in SIAL is simple, there are multiple implementation of `pardo` loop, which differ in the distribution of indices and thus 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 sub classes: SIAL supports large sparse arrays. To loop over them efficiently SIAL has various implementations of fragmented pardo loops. These loops have knowledge of internal structure of the sparse arrays and thus can skip over rows and columns having non 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 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 set of indices and populates interpreter state. This state is used by interpreter to calculate blocks using array and index values.

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

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

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

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

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

- `peek_indices` returns set of indices and internally takes care of maintaining and creating 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 list by 1 if the set of indices requested for is last one in the list and there are more indices in the loop.
  Algorithm 4.4. *peek_indices(IndexList::iterator) → [index], IndexList::iterator*

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

- `prefetch_indices` remembers the last set of indices returned to each `GET` statement and returns the next set of indices on call. It remembers by mapping the position of indices in list to `line numbers` of each `GET`. This makes varying the number of prefetched blocks for each `GET` possible.
  Algorithm 4.5. *prefetch_indices() → [index]*

```
1: function PREFETCH_INDICES
```

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

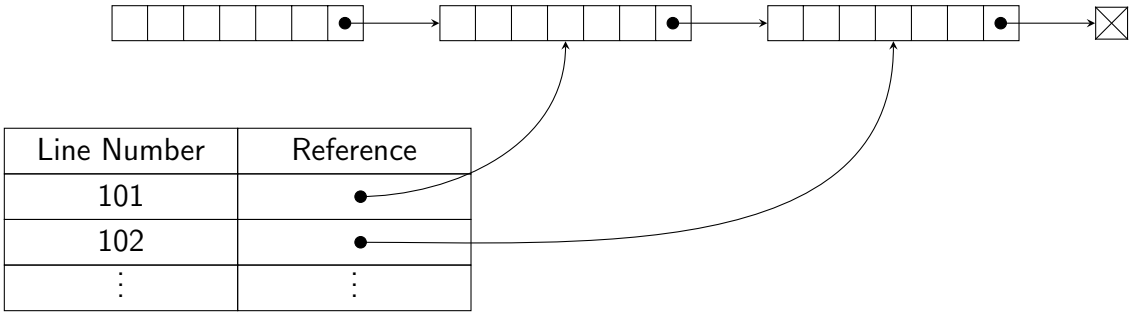

Figure 4-1. `prefetch_indices` saves mapping of line number to position in list

- `update` is now changed to simply pop the first set of indices from the list and update interpreter state so that other modules can find the value of current indices. This decreases the length of list by 1.
  Algorithm 4.6. *update_indices() → bool*

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

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

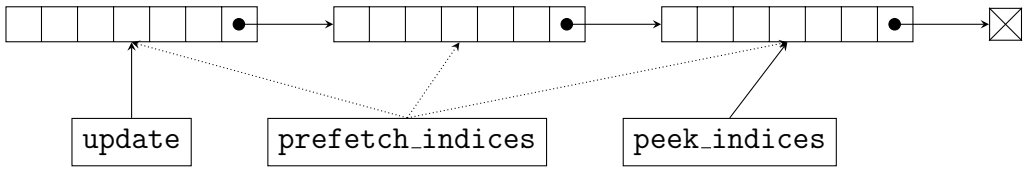

Figure 4-2. `update` consumes, `peek_indices` produces if needed, `prefetch_indices` is free to move along list

CHAPTER 5
EXPLOITING GPU

5.1 Background

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

5.1.1 Attempts in ACESIII

There have been attempts(12) made in previous version of ACES to use GPU to speed up computation. In this work the SIAL programmer had to deal with a lot of low level GPU memory operations such as allocating memory on GPU, copying blocks to and from GPU and deallocating memory on GPU. Since not all calculations are suitable to be executed on GPU, the SIAL programmer had to mark regions of SIAL code suitable to be executed on GPU.

Listing 5.1. Code Fragment from ACESIII for CCSD calculation

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

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

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

5.2 Runtime Memory Management

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

Due to this added layer, supporting multiple compute devices is now possible. The meta data includes whether the block data was *dirty*, *valid* and a **version number** which is incremented each time the block is changed by the compute device. This helps in keeping the GPU and main memory (CPU) synchronized.

Listing 5.2. `Device_Info` Class structure

```
1 class DeviceInfo {
2   double* data_ptr_;
3   unsigned int data_version_;
4   bool onDevice; // is block data on current device
5   bool isDirty; // is block data modified/dirty
6   bool isAsync; // are there any pending operations on device
7 }
```

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

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

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

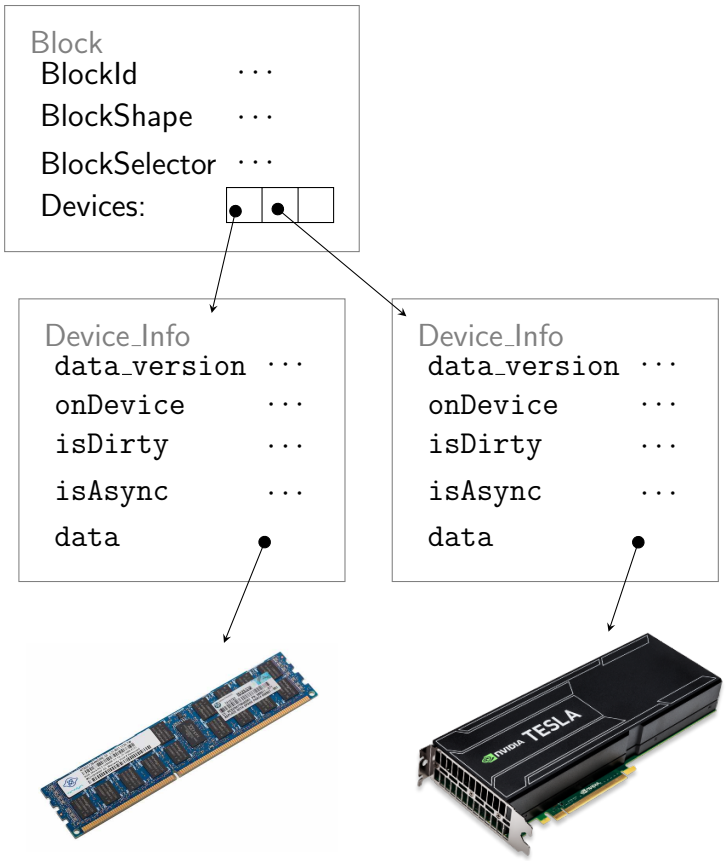

Figure 5-1. `Block` and `Device_Info` structure

```
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 pointer to memory location on device identified

by `deviceid`. The logic to always return 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:     end if
6:     return this.devices[deviceid].data
7: end function
```

5.3 Optimizing Block Copying

An enormous cost has to be paid (21)(22) 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 executing on CPU, it is necessary to minimize time spent on block transfers. This has been achieved by deploying multiple optimizations that are explained in this section.

5.3.1 Reducing Block Transfers

The SIAL runtime has the information about intent of each block request, whether the block is being requested to be read, updated or written. Using this information the number of synchronizations between devices can be reduced. The blocks which are going to be written need not be synchronized, however the 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:     end if
```

10: **return** *this*
11: **end function**

Now, 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)` as in case of block read or update. If no synchronization is needed, as in case of block being written, runtime can simply get the pointer on device memory by calling `block->get_data(device)`. This change in accessing and updating block data along with the information about intent of the request can reduce several unnecessary synchronizations.

5.3.2 Memory Pinning

The memory allocated on CPU or main memory is pageable by default. The GPU cannot access data on such pageable host memory (23) (24). For this reason when a memory copy operation is requested from host to the GPU, the CUDA driver first allocates a temporary **page locked**, or *pinned* memory on host (main memory) and copies the host memory to the temporary pinned memory and then finally transfers the memory from pinned memory to GPU device memory.

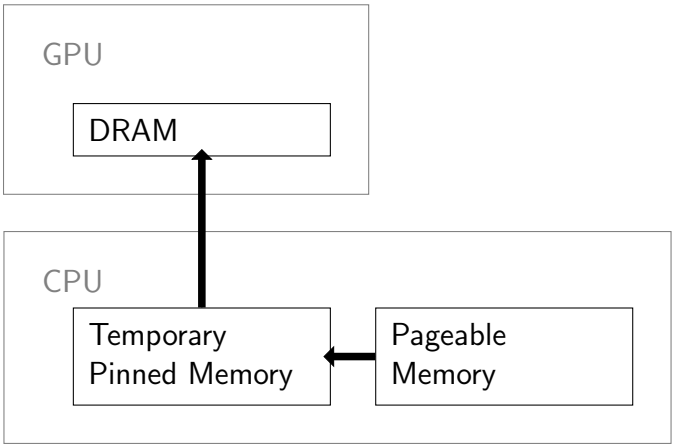

Figure 5-2. `memcpy` w/o memory pinning

Due to this copying of data to first a temporary page locked memory, extra time is spent in redundant copy operation. To overcome this, the host memory can directly be allocated in pinned memory. For this CUDA gives API `cudaMallocHost()`, `cudaHostAlloc()` to allocate

memory and `cudaFreeHost()` to deallocate the memory. Using this technique the data flow presented in 5-2 is modified to flow presented in 5-3.

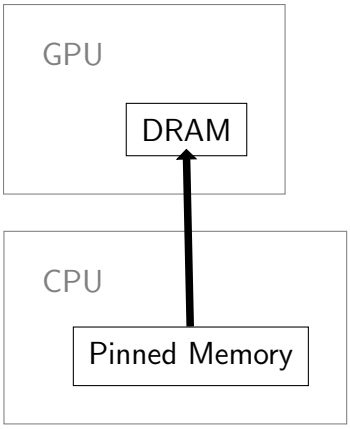

Figure 5-3. `memcpy` with memory pinning

5.3.2.1 Page Locked Memory Bandwidth

Apart from saving a redundant copy operation, memory page locking also helps in reducing time to copy because when memory is page locked, the GPU can invoke Direct Memory Access (DMA) controller (25)(26)(27)(28) to transfer memory bypassing the CPU. This results in around twice (23) the bandwidth between host and GPU.

5.3.2.2 Asynchronous `memcpy`

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

5.3.3 Memory Pinning Overhead

It was observed that while pinned memory made memory transfer faster, it carried a large overhead as compared to allocation of non pinned memory. Similar observations by Boyer et al (29)(30)(31) have been reported. They reported memory allocation call `cudaMalloc` to be 30-40% more expensive than `malloc` and `cudaHostMalloc` to be slower than `malloc` by a

factor of 100. To overcome this, a caching scheme was developed to reuse the already page locked memory pages.

5.3.3.1 Reusing Page Locked Memory Pages

The problem of caching page locked memory pages is similar to problem of dynamic allocation of memory which is handled by Operating System. Extensive work (32) is done in solving the problems involved in efficiently allocating memory dynamically such as external and internal fragmentation along with minimizing the time taken to find suitable block of memory. Caching and serving page locked memory blocks is similar to dynamic memory allocation in the way that it needs to find suitable size memory block such that least space in form of internal fragmentation is wasted but it is different from dynamic allocation in the way that the memory blocks cannot be split or coalesced.

5.3.3.2 Implementation of Caching Mechanism

When a page locked block of memory is deleted, instead of un-pinning it and calling `delete[]` on it, it is saved in a list. Two maps are used for book keeping. One of it, `free_list`, presented in figure 5-4, maps the block size to a list of pointers to `double`. When a request for block of memory is received this list is searched for block of size equal or greater than size of block requested. If no such block exists 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 un-pinned and returned to OS and then a new block is requested. If there are no blocks in `free_list` and block size is greater than free memory then system is out of memory and exception is thrown.

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

Algorithm 5.4. *Page Locked Block Allocation*

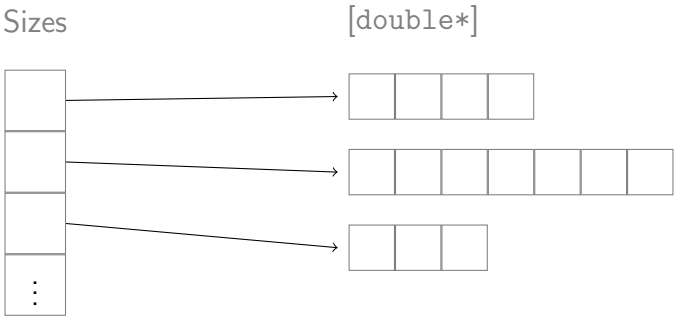

Figure 5-4. Structure of `free_list` map used for mapping block size to list of free blocks.

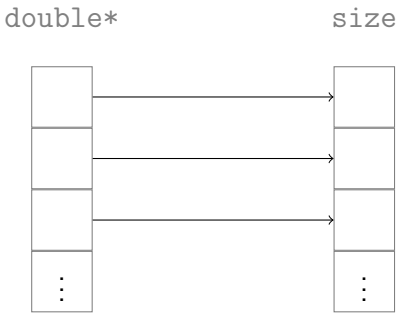

Figure 5-5. Structure of `reverse_lookup` map used for mapping back blocks to actual size of block.

```
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
```

```
    else
        cached_block ← cached_block_list.pop()
        return cached_block
    end if
end function

function FREE(block)
    actual_size ← reverse_lookup[block]
    free_list[actual_size].append(block)
end function
```

5.3.4 GPU Streams

A *stream* in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host (33). Streams can be considered as pipes of operations in which the operations get evaluated in First In First Out (FIFO) fashion. Operations in different streams can be interleaved and CUDA driver might execute them concurrently.

5.3.4.1 Non Blocking Device Synchronization

GPU streams were used to implement non blocking SIA block synchronization. Using multiple CUDA streams and page locked memory, DMA controllers can be invoked to transfer memory from host to GPU and vice-versa asynchronously to both CPU and GPU kernel execution engine. Current generation of GPU have two DMA controllers (27)(28) 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 implement the non blocking synchronization, multiple streams are created and stored in the module `gpu_super_instructions` 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 same time a bit `isAsync` is set to denote that the `Block` object has pending asynchronous operations. Before de-referencing the memory pointer, runtime should wait for the operation to finish by using CUDA call `cudaStreamSynchronize`.

Algorithm 5.5. *Asynchronous Block Synchronization*

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

5.4 GPU Buffers in Internode Communication

SIAL makes it possible to work on extremely large arrays by dividing the array into multiple blocks and then server distributing the blocks to workers. Workers and Server nodes communicate using highly optimized MPI library. Since GPU have their own memory, using GPU for computation in a multi node environment can add an additional layer of communication between GPU and network cards. In this section optimizations for this communication are presented.

5.4.1 Background

GPU cards have their own memory on which the GPU can compute. In Section 5.3.2 the techniques used for optimizing memory transfer between host memory (main memory) and GPU memory are discussed. Additionally when MPI is used for internode communication, the data buffers to send are staged to MPI buffers automatically by MPI. To send a buffer allocated on GPU using MPI, it first has to be copied to main memory so that its pointer can be passed to MPI functions like `MPI_Send`/`MPI_Recv`. Thus when GPU are used along with MPI to send a unpinned memory buffer, in all there will 3 memory copy operations and 6 in total if we consider memory operations on the destination node. These operations are represented in Figure 5-6 by black arrows.

5.4.2 MPI Transfers Using DMA

After page locking a memory buffer, the CUDA driver and network fabric driver can share it. Thus two set of copy operations can be saved by requesting page locked buffers. This transfer is represented in Figure 5-6 by blue arrows. It is worth noting that though this saves couple of copy operations, the buffers still are managed through CPU memory and thus are limited by the CPU memory bandwidth. SIA will make use of this when the buffers are page locked and software libraries & hardware facilities are not present for Remote Direct Memory Access (RDMA) which is presented next.

5.4.3 MPI Transfers Using RDMA

The intermediate copying operations to host memory can be avoided further by taking advantage of RDMA. RDMA is a technique using which the GPU can send buffers from GPU memory to network adapter without staging through host memory. OpenMPI supports RDMA (34) 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, the servers in SIA are not allocated GPU since servers are not responsible for any heavy calculation on blocks. They just manage blocks from workers and swap *inactive* blocks to disk. Hence a more accurate representation of use of RDMA in SIA is presented in Figure 5-7.

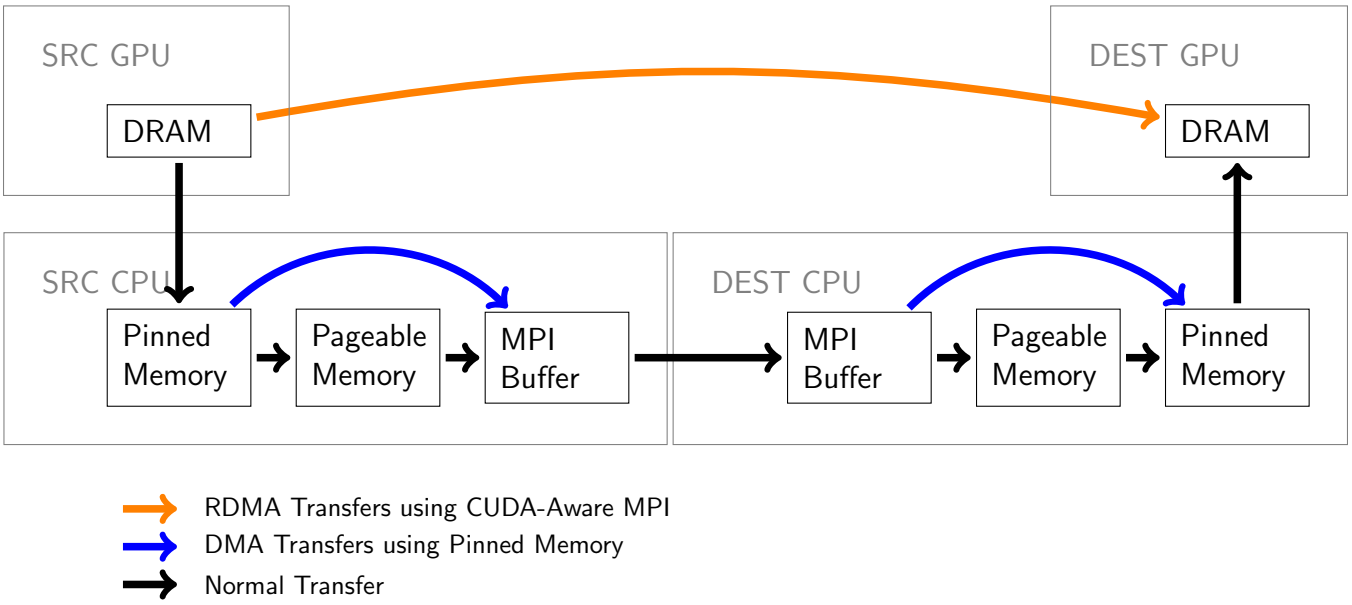

Figure 5-6. RDMA, DMA and normal transmission between two nodes with GPU

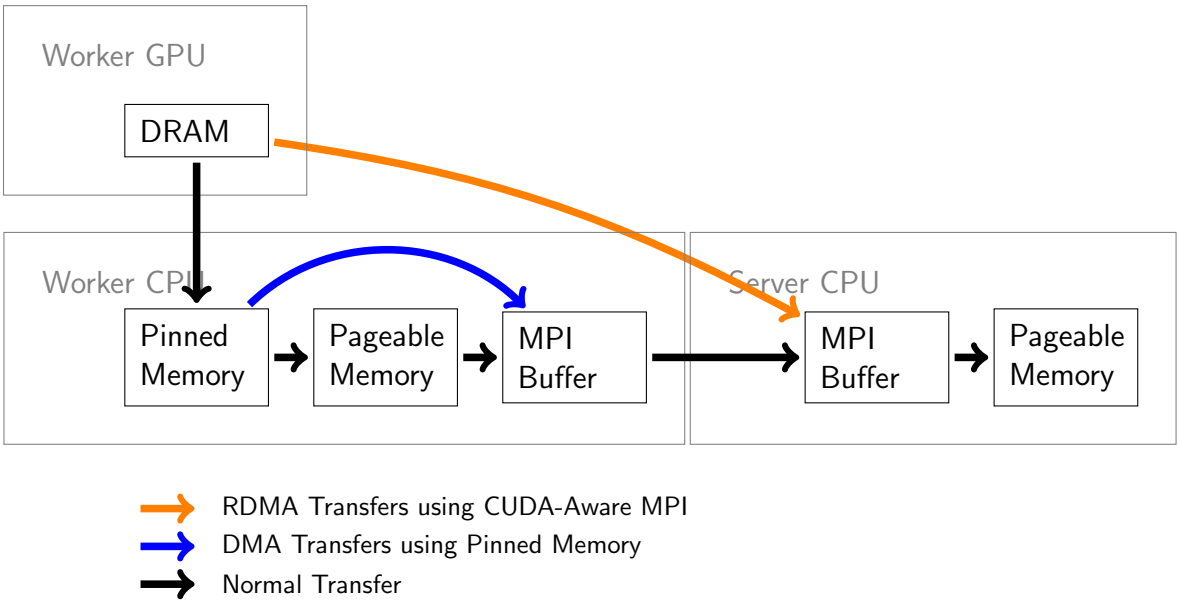

Figure 5-7. RDMA, DMA and normal transmission in SIA

CHAPTER 6
EXPERIMENTS AND RESULTS

This thesis present a few of the possible performance optimizations on looping constructs for SIA. These optimizations primarily aimed to reduce the network operation cost and computing cost. This section describes series of experiments conducted by varying input parameters such as block size as well as the optimization parameters such as number of blocks to prefetch and its effects on the performance of the system.

6.1 Environment

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

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

Table 6-1. HiperGator 2 Spec Sheet

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

6.2 Prefetching

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

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

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

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

Table 6-4. HiperGator 2 Node interconnect specification

6.2.1 `hit_ratio`

To understand the performance of the prefetching mechanism a new metric is introduced. Prefetch `hit_ratio` is defined as the ratio of the number of times the SIA runtime did not have to block for a certain data block to be ready and total number of times the data block is 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 where appropriate to measure effectiveness of parameters in prefetching.

6.2.2 Index Length

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

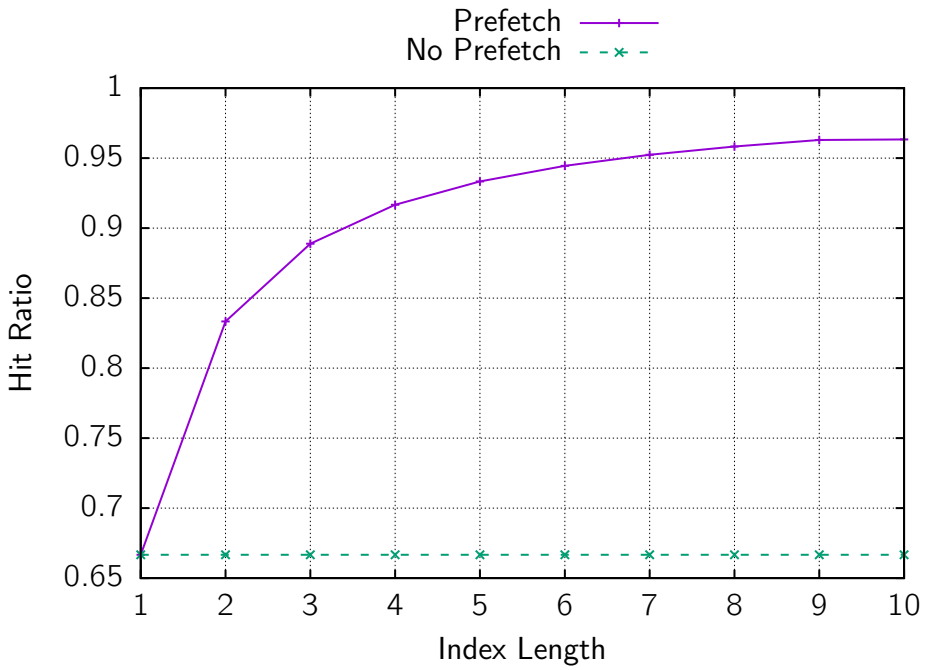

Figure 6-1. Index Range Length v/s `hit_ratio`

Note that the runtime has to block for data only 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 a index spans only 1 then there is no scope for the runtime to do prefetching. This is evident from the plot when index length is 1, the `hit_ratio` with prefetching is equal to with no prefetching. As range of index length increased, prefetching gets working. This can be easily observed from exponential growth in `hit_ratio`. And eventually the curve for `hit_ratio` flattens out after 6 since no significant improvement is achieved by increasing the index range length.

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

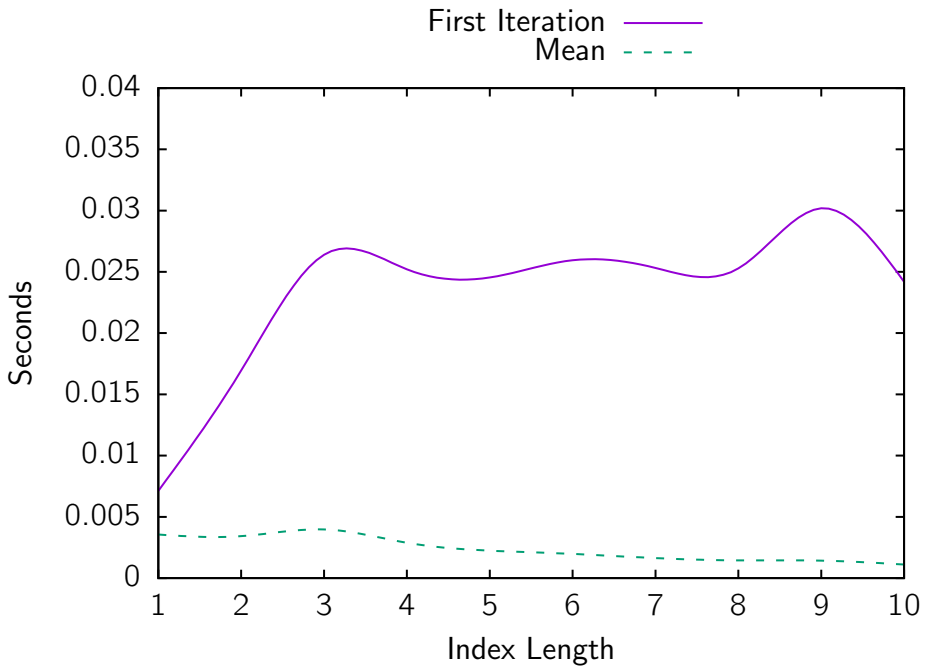

Figure 6-2. Index Range v/s `wait_time_` per iteration

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

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

6.2.3 Block Size

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

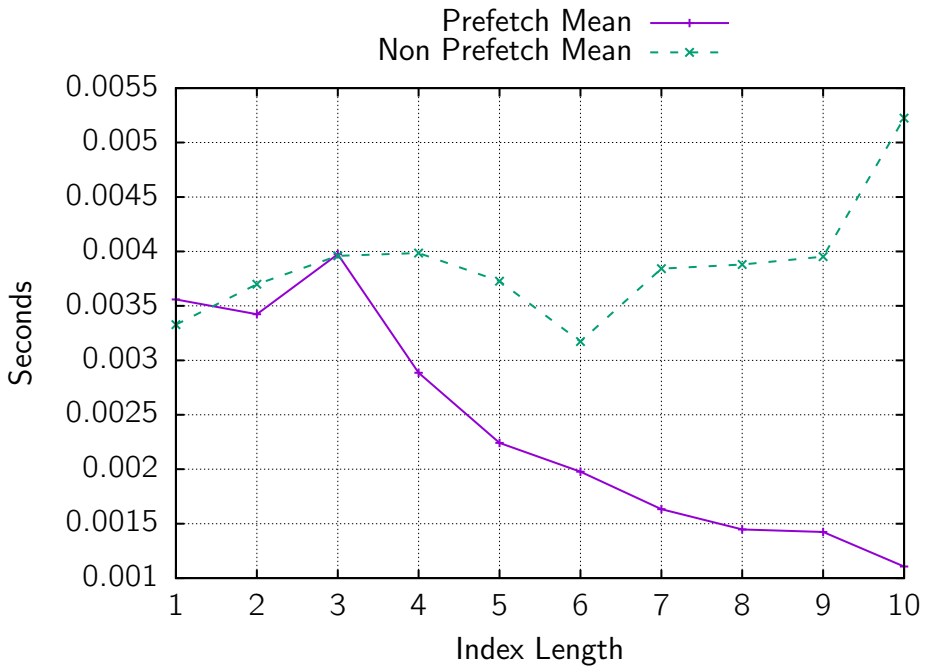

Figure 6-3. Index Range v/s `wait_time_` per iteration in Prefetched and no Prefetched Loop

As the block size increases, `wait_time_` for first iteration for loops with prefetch grows much faster than loops without prefetch. At block of size greater than 500 elements `wait_time_` for first iteration with prefetch is almost twice the corresponding `wait_time_` without prefetch.

But once the first request for blocks is made, subsequent iterations are not affected by the block size as compared to loops in which prefetching is not done, since the runtime need not block for subsequent blocks. This results in overall reduction in mean `wait_time_`. This trend is presented in figure 6-5.

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

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

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

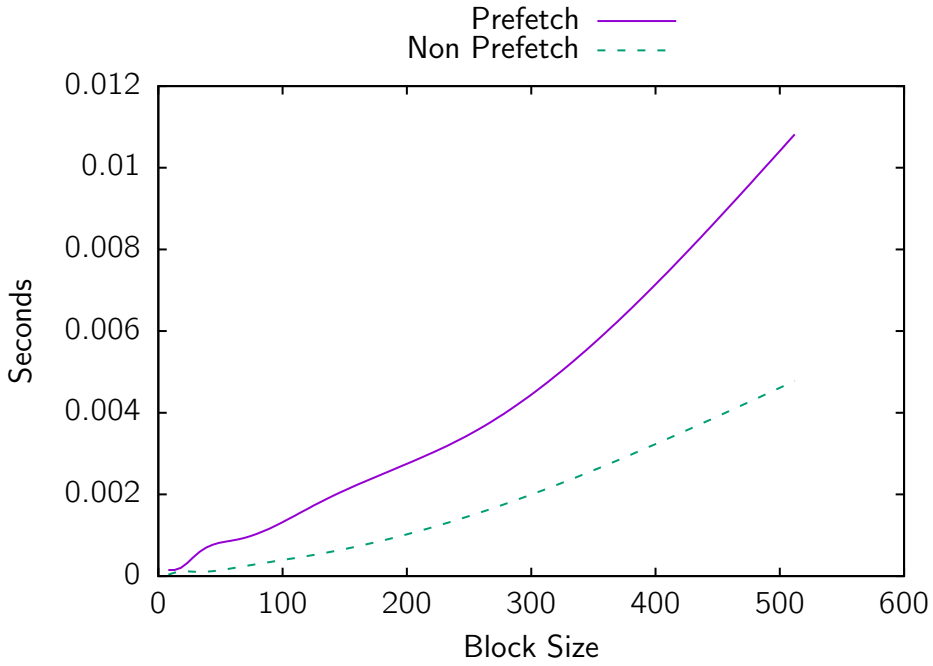

Figure 6-4. Block Size v/s `wait_time_` for first iteration

6.2.4 Number of Blocks to Prefetch

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

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

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

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

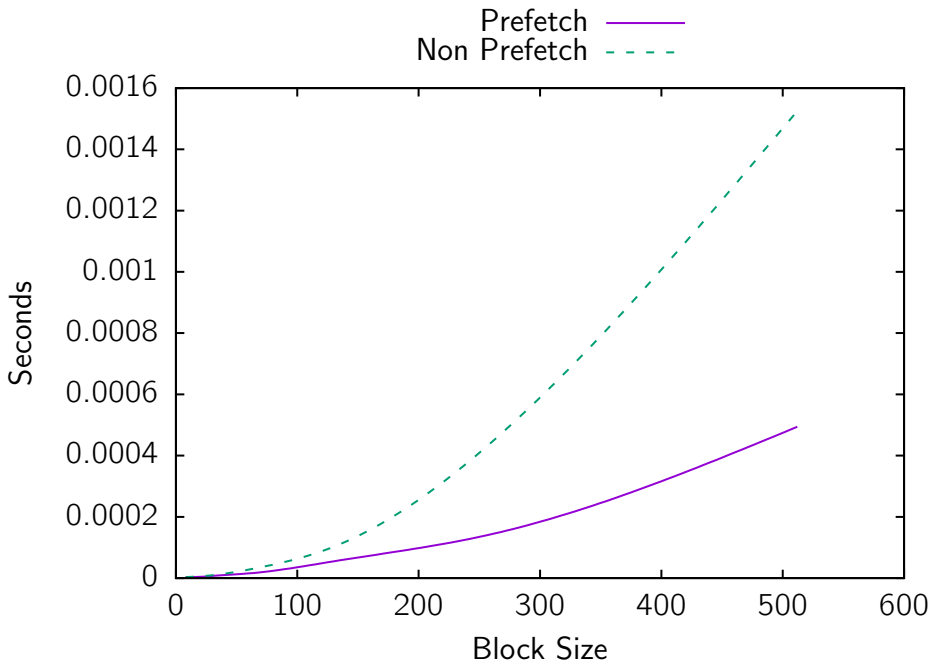

Figure 6-5. Block Size v/s Mean `wait_time_` for Prefetched and No Prefetch Loop

prefetch increases and then it grows again as increase in number of blocks to prefetch increases the `wait_time_` for first request to server.

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

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

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

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

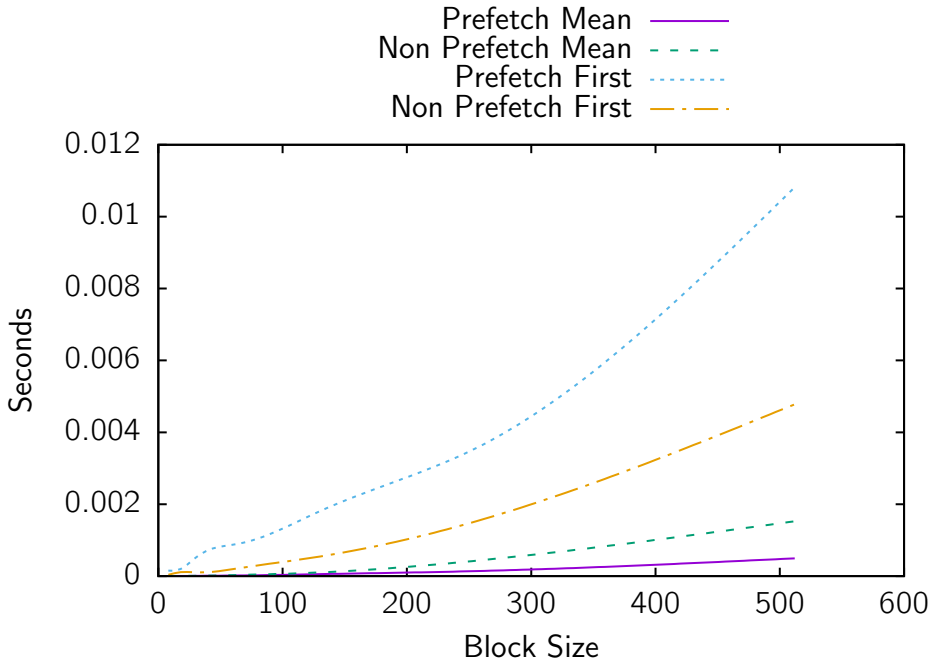

Figure 6-6. Block Size v/s Mean `wait_time_` for Prefetched and No Prefetch Loop

As number of server increase the overall run time for programs with prefetching decreases. This is clear from figure 6-10, as the difference in run time of programs without prefetch and prefetch increase. It is worth noticing that the only couple of programs show significant increase in difference while one of the programs shows no difference in prefetching and no prefetching. To study why this is the case, total `wait_time` of the programs is plotted in figure 6-11.

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

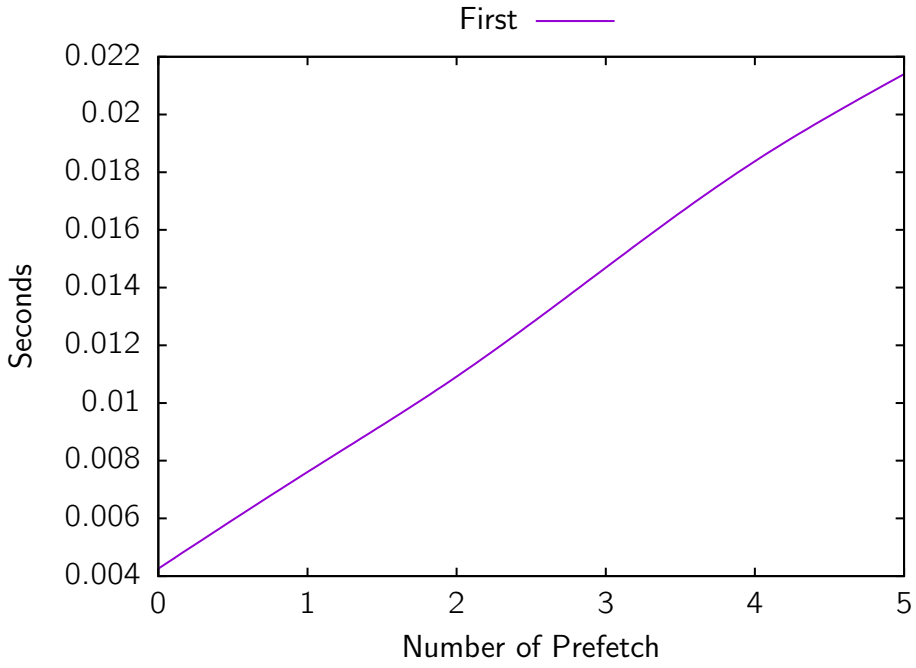

Figure 6-7. Number of Block Prefetched v/s `wait_time_` for first request

6.3 GPU

This section presents the experiments performed to benchmark the performance of using Graphic Processing Unit.

6.3.1 Memory Pinning

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

6.3.1.1 Copy Speed

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

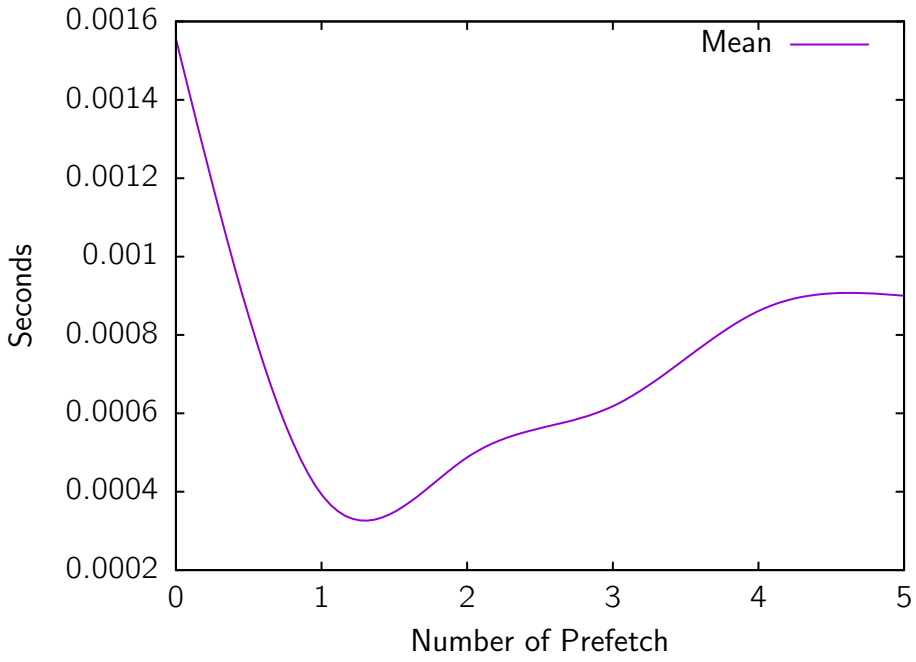

Figure 6-8. Number of Block Prefetched v/s mean `wait_time_`

It is evident from the plot and table 6-5 that explicit page locking only results in faster memory copy if the size of 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.

6.3.2 Optimized Transfer

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

6.3.3 Memory Pinning Overhead

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

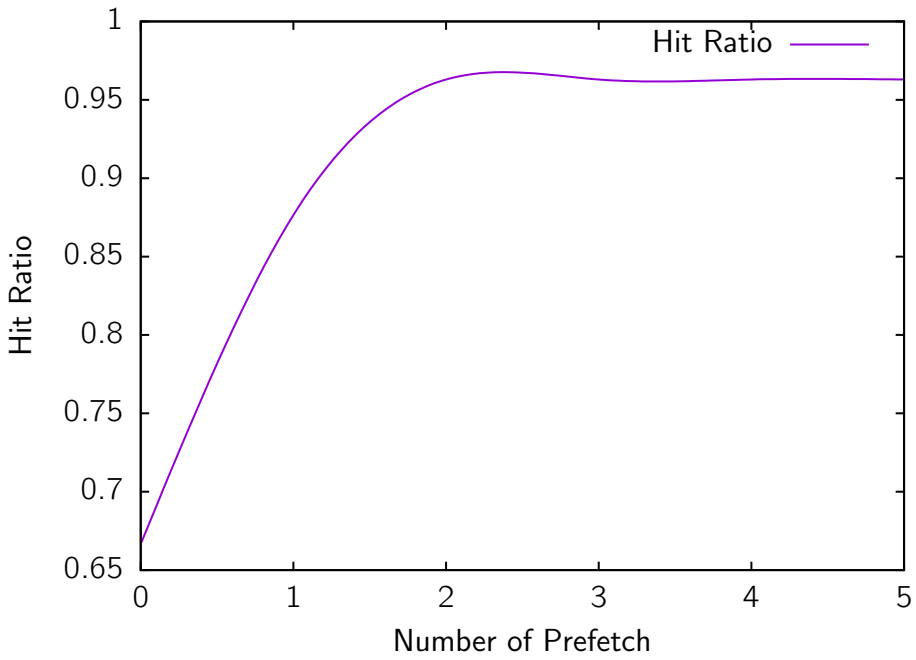

Figure 6-9. Number of Block Prefetched v/s Hit Ratio for first request

6.3.3.1 `alloc`

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

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

6.3.3.2 `free`

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

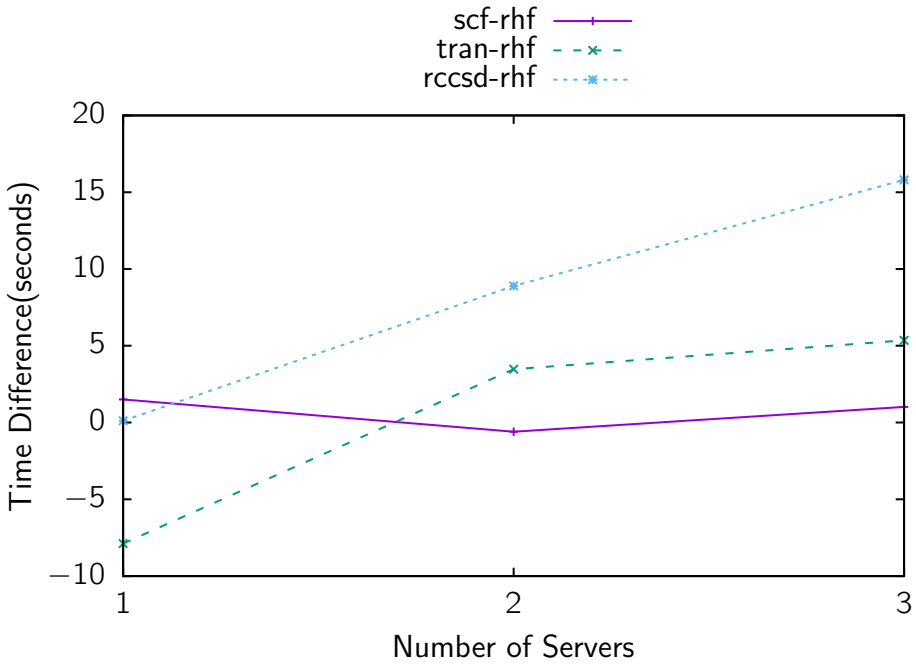

Figure 6-10. Effects of varying number of servers on $C_{12}H_{10}$ molecule.

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

6.3.4 RDMA

To benchmark RDMA, CUDA aware implementation of MPI was used. The size of block was varied against CPU and GPU buffer passed to MPI transfer functions. The results for `GET` operation are presented in figure 6-16.

6.3.4.1 `GET`

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

1. If CUDA aware MPI implementation is used then DMA is used for transferring data from network fabric to GPU buffers directly.
2. It seems that just by page locking main memory, even without using GPU, DMA is invoked by MPI implementation for transfer of data from network fabric to main memory.

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

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

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

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

6.3.4.2 PUT

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

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

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

6.3.4.3 Total Transfer

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

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

6.3.5 Caching Page Locked Blocks

Caching page locked memory blocks is important since allocating page locked memory is 100 times costlier than non page locked memory. To test caching two parameters have been selected as most appropriate: hit ratio and time spend 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.

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

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

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

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

Table 6-8. Page Locked Memory Blocks hit rate

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

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

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

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

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

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

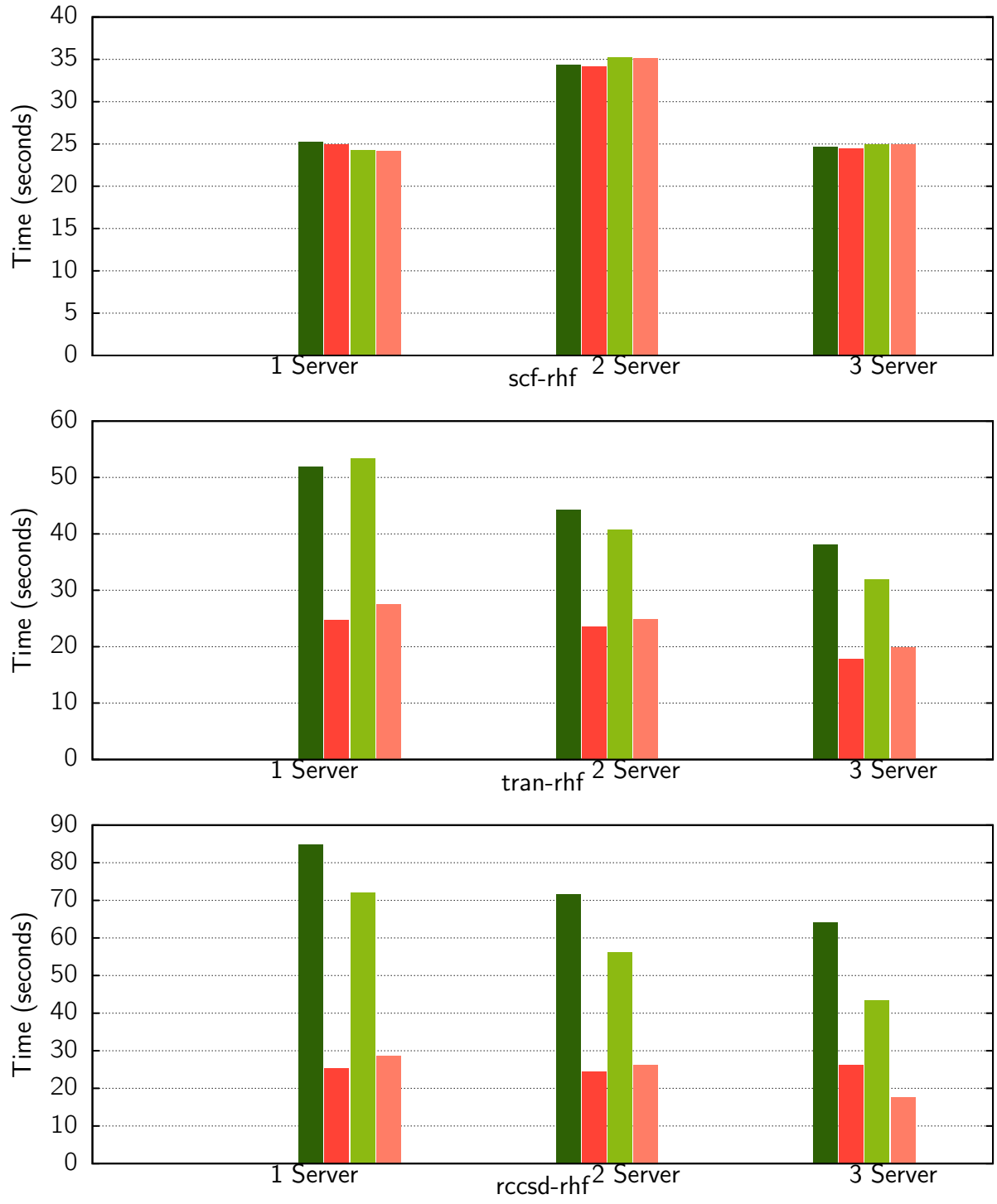

Figure 6-11. `wait_time` and barrier `wait_time` variation againsts number of servers.

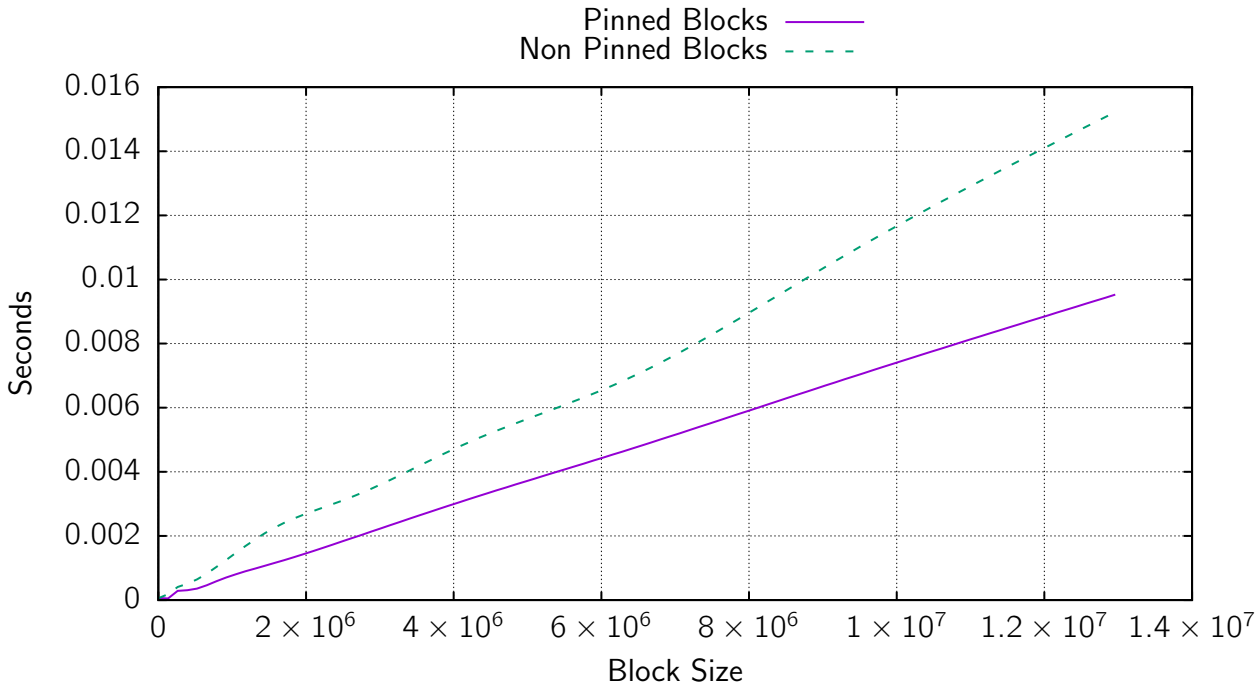

Figure 6-12. Time taken to transfer block to GPU for *pinned* and *non pinned* blocks

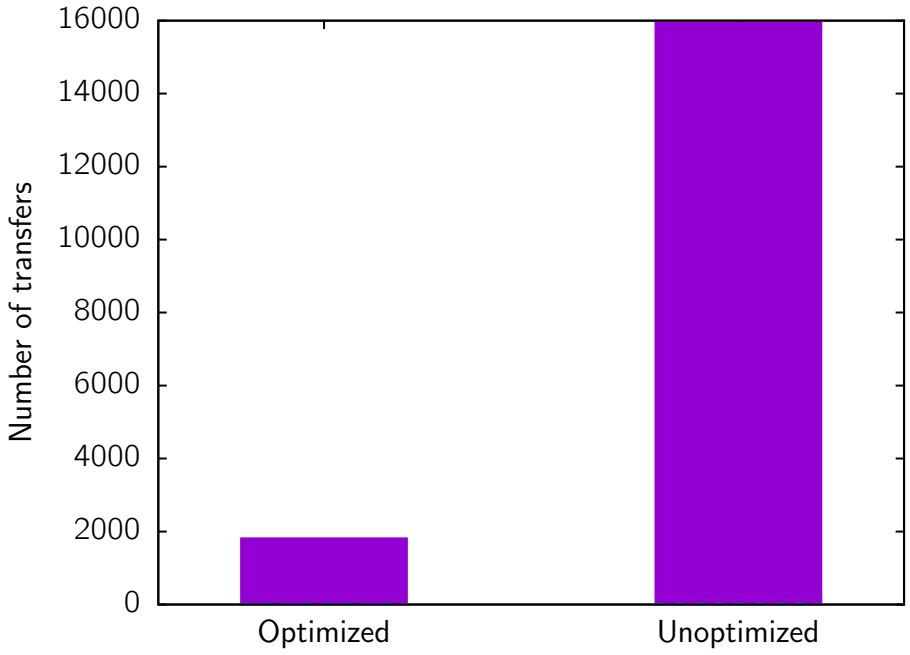

Figure 6-13. Optimized v/s Unoptimized Block Transfers for `rccsd_rhf.sialx`

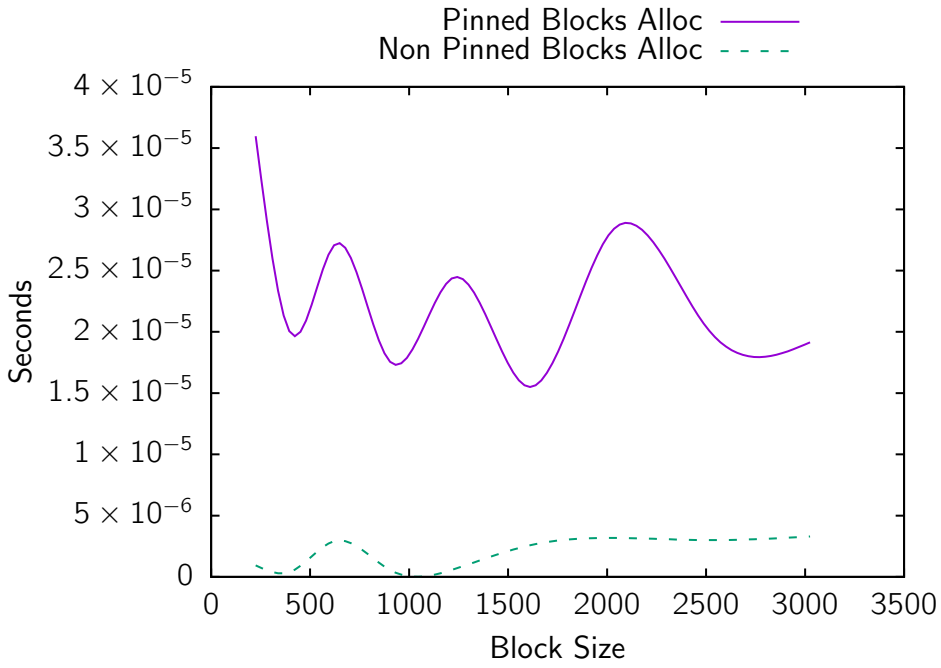

Figure 6-14. Pinned and non Pinned memory allocation

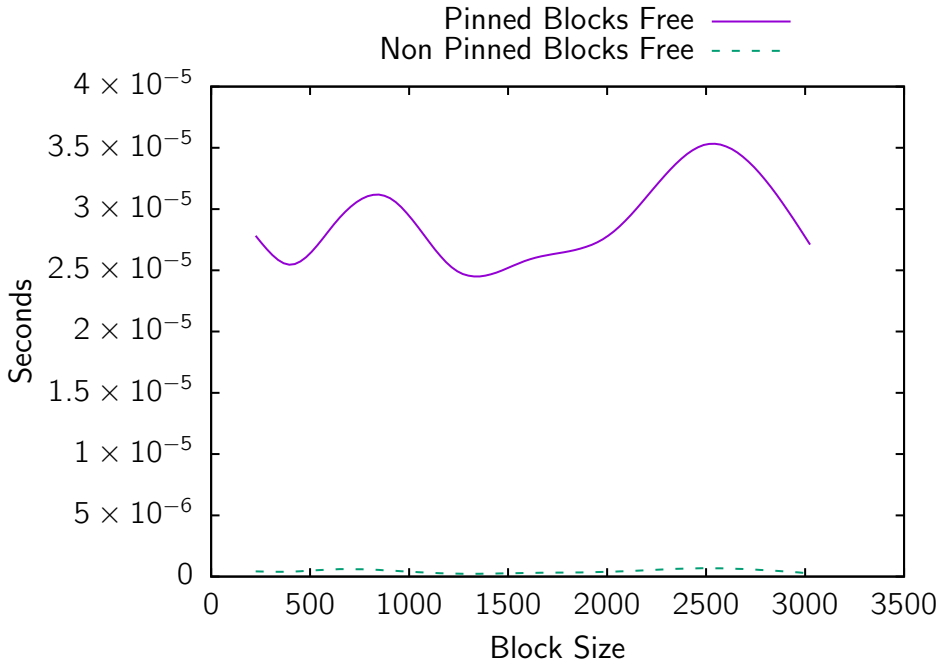

Figure 6-15. Pinned and non Pinned memory de-allocation

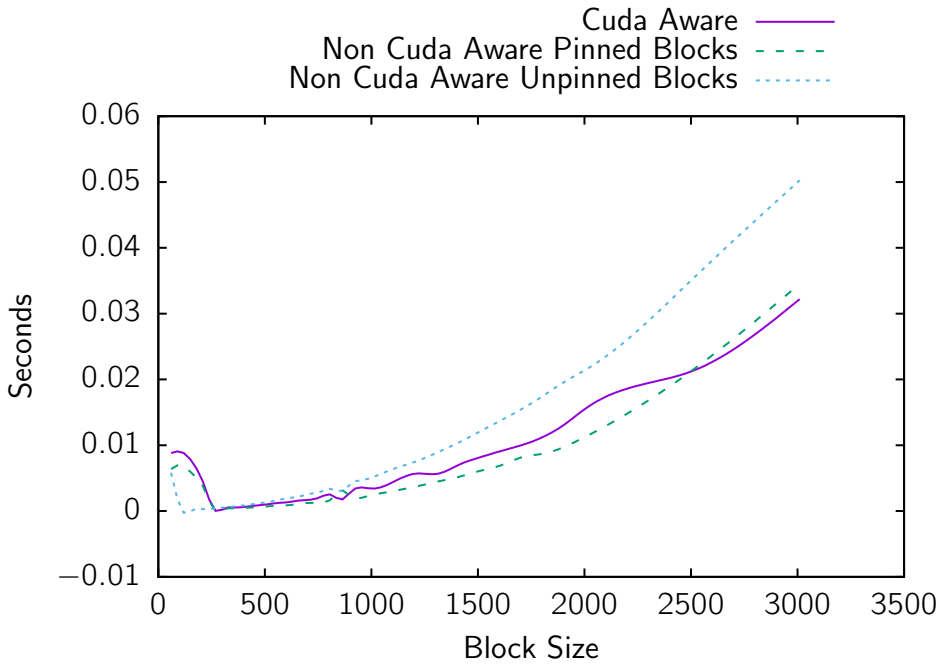

Figure 6-16. GPU and CPU buffer passed to `GET`

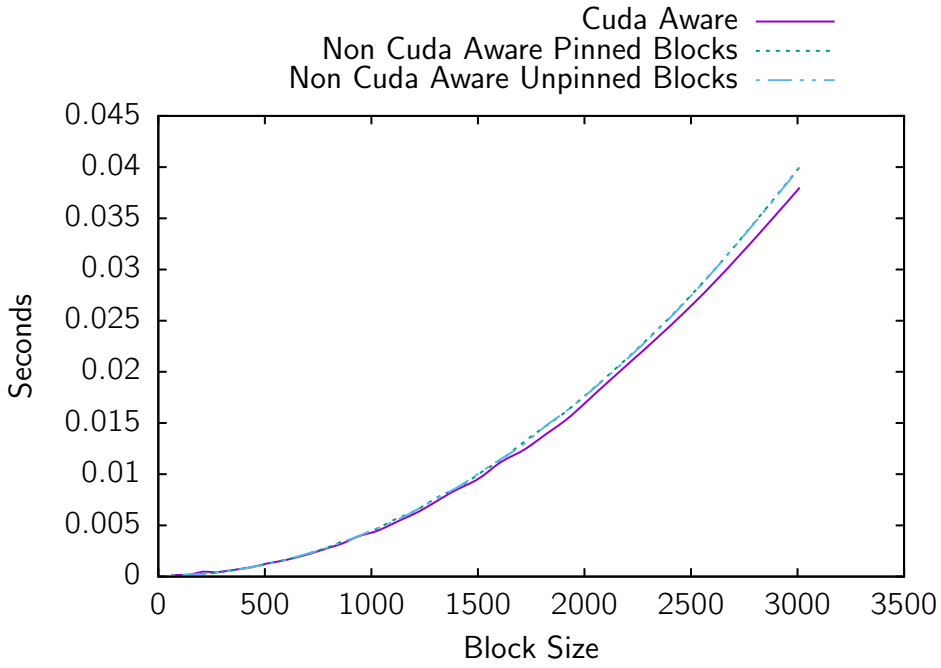

Figure 6-17. GPU and CPU buffer passed to `PUT`

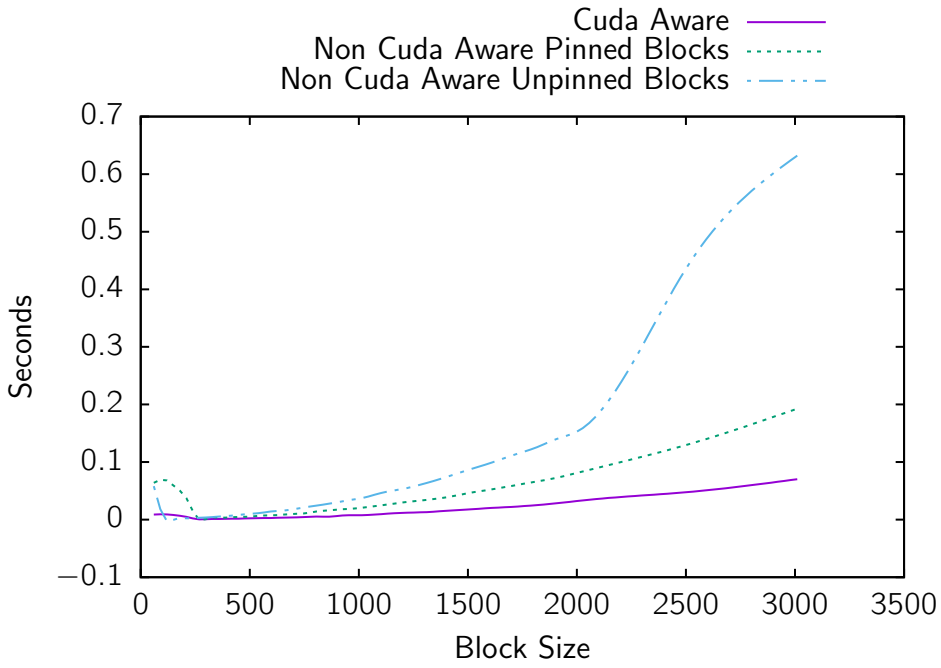

Figure 6-18. Total MPI transfer compared to CUDA Aware MPI transfer

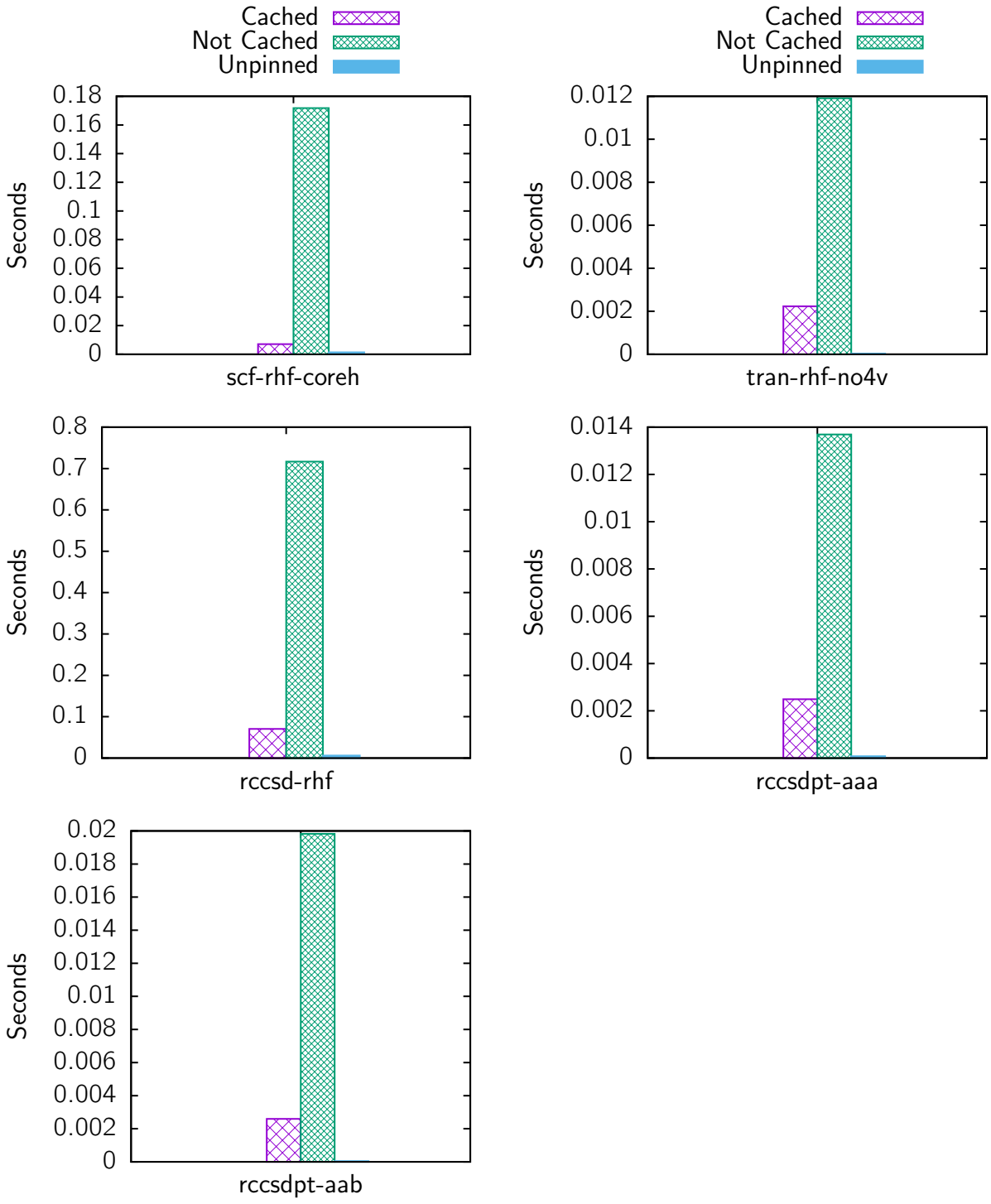

Figure 6-19. Page Locked Cached v/s Page Locked Uncached v/s Non Page Locked allocation and deallocation times

CHAPTER 7
CONCLUSION AND FUTURE WORK

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

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

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

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

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

to GPU on servers and implementing RDMA to transfer memory blocks between workers and servers. Lastly, implementing advanced caching mechanism for page locked memory blocks can be worked upon to have more memory efficient caching mechanism.

APPENDIX
SIALX PROGRAM USED FOR BECHMARKING

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

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

```
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
```

REFERENCES

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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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.

[12] 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

[13] 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.

[14] 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

[15] 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

[16] 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

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

[18] 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

[19] 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

[20] 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

[21] 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

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

[23] 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/

[24] ——. Programming guide cuda. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[25] ——. (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

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

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

[28] ——. (2013) Nvidia tesla k series gpu accelerators datasheet. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf

[29] 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

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

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

[32] 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

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

[34] 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.