NEAR-MEMORY PRIMITIVE SUPPORT AND INFRASTRUCTURE FOR SPARSE ALGORITHM

A Dissertation
Presented to
The Academic Faculty

By

Kartikay Garg

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2017

NEAR-MEMORY PRIMITIVE SUPPORT AND INFRASTRUCTURE FOR SPARSE ALGORITHM

Approved by:

Dr. Sudhakar Yalamanchili,
Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Jeffrey Young, Co-advisor
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: April 28, 2017

ACKNOWLEDGEMENTS

I wish to express my gratitude to my advisor Dr. Sudhakar Yalamanchili, for his support and motivation through the course of my thesis. His guidance was invaluable to my research. I would also like to express my utmost respect for my co-advisor, Dr. Jeffrey Young. His unwavering faith in my capabilities and potential helped me make great strides in my work and beyond. I look up to him immensely. He patiently answered all my queries and provided extremely valuable feedback.

I am also very grateful to Piyush Sao, for providing me with his invaluable time and assistance. His efforts to guide me through his base code, greatly assisted me to develop critical components for my work.

I am greatly indebted to Dr. Tushar Krishna, who not only taught me my Computer Architecture course, but also provided me numerous opportunities to expand my knowledge and work on research projects. His support has been an invaluable component of my grad life.

I would also like to thank my fellow lab mates, Burhan Mudassar, Ramyad Hadidi, Blaise Tine, Chad Kersey, and Karthik Rao for helping me out greatly during the course of my program. They were always welcoming and helped me a great deal through different aspects of my graduate studies.

I thank the committee members for their precious time and consent to serve on my committee. I thank them for their insightful comments.

Finally, a special thanks to my family for their moral support and encouragement. They are the pillar of my strength. I thank my parents and my sisters, for their loving support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Current trends in multi-core processors and heterogeneous architectures have aimed at improving and performance in terms of raw GFLOPS/sec . This has had fundamental implications on interaction between compute cores and the memory system. Particularly for the HPC (High Performance Computing) community, the conventional DRAM memory subsystem poses a major bottleneck in achieving peak acceleration speedups due to bandwidth and fetch latency limitations. Developers invest time and resources to select and design the optimal offload accelerators to avoid paying these latencies directly, and they couple this hardware design with algorithms to maximize compute utilization while hiding the maximum memory latencies [1].

This thesis introduces an approach to solving the problem of memory latency performance penalties with traditional accelerators. By introducing simple near-data-processing (NDP) accelerators for primitives such as SpMV (Sparse Matrix Multiplication of Vectors) and DGEMM (Double Precision Dense Matrix Multiplication) kernels, applications can achieve a considerable performance boost. NDP can be combined with new 3D-stacked architectures to provide high internal bandwidth and data parallelism. Additionally, the vertical TSV (Through Silicon Via) links in 3D-stacked memories can help reduce average access times for memory requests and accelerate atomic-type operations like Read-Modify-Write.

We evaluate these technologies using a common HPC algorithm, "LU decomposition for large order, sparse matrices". This algorithm is included in one of the most commonly used solvers for the HPC community, SUPERLU [2], and has been accelerated on GPU and Xeon Phi. We take the existing state of the art solver implementations of the SUPERLU suite as a baseline implementation to study and analyze memory access patterns with DRAM and stacked DRAM and to make meaningful inferences about the opportunities for acceleration. The work includes a preliminary analysis of extensions to the SUPERLU algorithm like HALO (Highly Asynchronous Lazy Offload) [1] on the CPU and GPU with a near-term path

to Field Programmable Gate Array (FPGA) accelerator platforms as well. We study the effect of block and grid size decisions and other optimization parameters on the performance of the application. Finally choosing a baseline implementation, we discuss the techniques to extract a memory trace, representative of an accelerated, bulk synchronous parallel (BSP) application behavior. This includes exploration of binary instrumentation techniques and simulation infrastructures as potential candidates.

Using the Pico Computing FPGA board with an on-board interfaced HMC (Hybrid Memory Cube) chip, we build upon the primitive framework provided by the vendor for application performance estimation and studying memory subsystem metrics. This enables us to study the behavior of the HMC for any given memory access pattern/trace. We study the effect of varying the command issue queue size and the number of contending ports (hardware queues) on the access latency and available peak bandwidth. Experiments include the effect of queuing parallel memory requests in a similar fashion to GPUs across multiple requesting ports contending for a single access link to the HMC. This is compared to an in-order issue strategy across multiple requesting ports. These experiments help us decide an optimum batch size for the target application on the hypothetical custom offload HMC accelerator.

From our experiments, the optimum batch size is also dependent upon the number of active ports contending for access to the HMC. We demonstrate that the performance of the HALO LU decomposition kernel is platform dependent and that block size should match the throughput of the CPU, GPU, or FPGA accelerator. We conclude by saying that a strong correlation exits between the command FIFO depth in the HMC hardware and request batch size in the kernel on host and that future algorithm design should take these factors in to account.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Large scientific codes, such as simulations of experimental fusion systems like NIMROD [3], rely in part on solving large systems of linear equations and solver libraries like Lawrence Berkeley's SUPERLU library [2]. When these real world applications are modeled as HPC problems, they tend to be represented as large sparse matrix based calculations, and a major chunk of HPC applications use sparse kernels like Sparse Matrix-Vector multiplication (SpMV) as part of their core simulation kernels. These simulation kernels can thus be accelerated by linear algebra operations, such as LU decomposition, a computational technique that is widely incorporated in the aforementioned SUPERLU library. Accelerating HPC compute primitives such as SpMV or LU decomposition can directly lead to a performance boost for the applications employing them, including more than 15 well-known DoE scientific codes. Previous approaches to accelerate these types of primitives, such as those in [4, 1], map these computations onto distributed and accelerated systems like large CPU or GPU-based clusters. However, these systems are ultimately limited by the slowest component, which has recently been the memory subsystem.

Secondly, with the end of Dennard Scaling, HPC compute clusters face a tremendous fundamental problem of thermal budgeting. For data-intensive applications with limited cache locality, the data transfer cost between the memory and the compute core constitutes a major chunk of the energy expenses. Thus, energy constraint considerations from the perspective of data movement need to be given utmost priority. One of the best ways to optimize energy usage for data intensive applications is to place processing as close as possible to the data in main memory [5]. Advances in 3D integration, specifically with

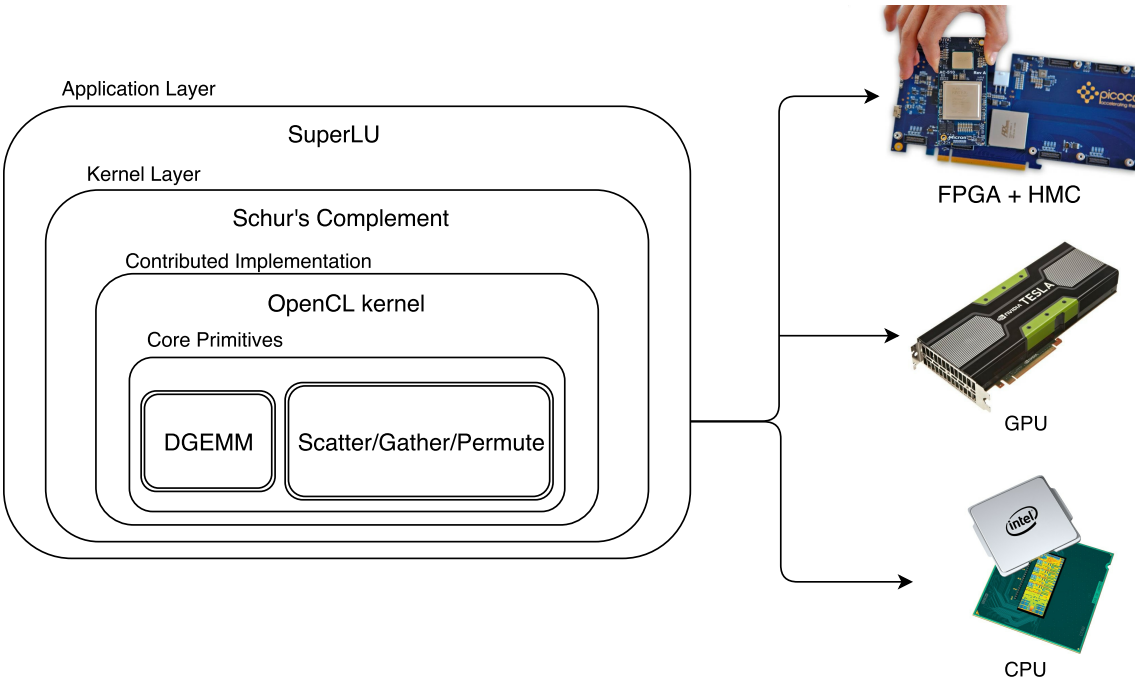


Figure 1.1: Scope of Work

High-Bandwidth Memory (High Bandwidth Memory (HBM)) and Hybrid Memory Cube (HMC), provide an opportunity to implement near-data processing (NDP) and processing in memory (PIM) without the technology problems that similar efforts had in the past.

These NDP and PIM architectures can reduce latency penalties of deep caches but can also offer extensive data level parallelism opportunities. This directly translates to reduced memory access latency and higher internal bandwidth for applications. This reduced latency helps support faster context switches and in the case of PIMs, open up opportunities for high-bandwidth Single Instruction Multiple Thread (SIMT) and Single Instruction Multiple Data (SIMD) architecture accelerator designs that can sit on the logic layer beneath the DRAM stacks.

At the same time, Field Programmable Gate Arrays (FPGAs) combined with 3D memories provide a power efficient platform for NDP designs due to their customization capabilities for specific applications. Additionally, new high-level synthesis techniques for programming FPGAs using OpenCL have introduced new avenues for accelerating HPC applications on re-configurable fabric. Together, 3D stacked memories and FPGA fabrics

can potentially enable developers to program a customized yet high-performance algorithm that uses less power than an equivalent GPU implementation.

Following the compelling reasons stated above, this work will look at such HPC primitives that can better support sparse algorithms (specifically those included in the SUPERLU package) with a new class of high-performance, 3D-stacked memories and also help to design and implement infrastructure to use algorithm components on new FPGA-based systems with 3D-stacked memories.

Thesis Statement: Reevaluating core primitives such as DGEMM, SCATTER, and GATHER for 3D-stacked PIM architectures that incorporate re-configurable fabrics can deliver multi-fold performance improvements for SUPERLU and other sparse algorithms.

Figure 1.1 visualizes the scope of our work and Figure 1.2 details the thesis contributions. This work analyzes the existing LU decomposition methods in SUPERLU and the recently proposed state of the art extensions for reducing data movement [4]. We complement our proposal with an analysis of dense matrices of large dimensionality to estimate the potential benefits of accelerating computation on the typical sparse matrices that are computed with SUPERLU. The results are bench-marked against the performance of the same SUPERLU primitives on GPU, which is the currently most widely adopted platforms for hardware acceleration of similar HPC workloads

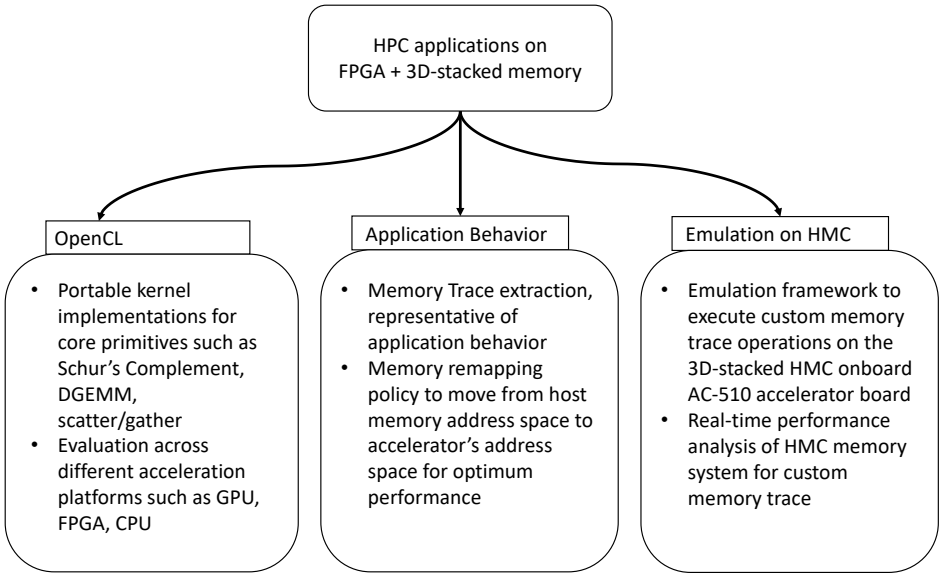


Figure 1.2: Thesis Contributions

1.2 Introduction to SUPERLU

SUPERLU is a general purpose library which has implementations of solvers for efficiently handling large sets of linear equations of the form $AX = B$ [6]. Non-linear systems tend to be expressed in form of a sparse matrix of high dimension value ranges. This calls for highly specialized direct solver algorithmic implementations to avoid redundant or unnecessary computation. The SUPERLU library is employed in a variety of computational science and engineering applications.

The library implementations offers single and double precision implementations. The properties of the matrices which act as raw input data structures for such libraries, have a major influence on the system call implementation. The rank, symmetric nature, fractal representations, etc. have a direct effect on the technique adopted. These different input requirements have led to optimized implementations of the functionality in various flavors such as sequential, parallel and distributed.

SUPERLU helps to solve a given sparse matrix A with LU decomposition, which factors

it to a product format represented as $A = L.U$, where L is a unit lower triangular matrix and U is an upper triangular matrix. This problem (sometimes referred to as sparse LU) is usually the most expensive step computationally, in a sparse direct solver like SUPERLU. It typically has a large memory footprint, thereby benefiting from the use of a distributed memory system. Given the increased utilization of some form of GPU-like acceleration for such systems, we need an efficient way to exploit all forms of available parallelism, whether in distributed memory or shared memory. This is the prime motivation for selecting "sparse LU decomposition" as the target application.

With this in mind, we aim at matching the available high degree of parallelism in the HMC with the respective flavor of the SUPERLU library to leverage maximum benefit. For this reason, we pick the SUPERLU_DIST package which has a high degree of parallel calls to map fractals of the matrix for partial solving to independent distributed clusters, many of which have accelerators like GPUs.

This implementation provides good motivation for an optimized offload accelerator in the HMC logic layer for data parallel independent HPC primitives similar to those scheduled on GPU SIMD lanes. An improvement in the primitive's memory request access latency and internal bandwidth owing to the HMC's 3D-stacked architecture could translate to direct boost in performance metrics for application run time. Since we cannot currently test acceleration in the HMC stack we instead pursue NDP implementations with FPGA for this work.

Stacked memory bandwidth and latency benefits can be complemented by the reduction in the number of multiple local copies needed to be created in accelerator global memories such as those of GPU clusters. For GPUs to achieve peak memory access performance, each independent workgroup/thread-block needs to be assigned local memory copies for asynchronized accesses by threads within the group/block. Later, different copies need to be merged in a synchronizing code section to create a contiguous copy (as in the case of array representations) or one big block of merged data elements stored in a sparse matrix

format. This can be handled much more elegantly in an NDP or PIM HMC implementation. Independent data fractals of the same sparse block representation can potentially be mapped to independent compute entities on the logic layer by carefully mapping the data store format in the HMC. This can be controlled by adjusting the granularity of assignment of banks, vaults or quads to independent data fractals. This can help bypass the need for multiple copies of the data blocks. This is because before finally shipping off the result in a sparsely formatted data to the host, a simple address stream for accesses to independent banks/vaults/quads can be created in a reorder buffer which delivers the desired data blocks in the respective order to the host. This may potentially help to overcome data copying overheads.

1.3 Introduction to 3D-stacked memory architectures

With advancements in packaging technology, we are now able to integrate multiple DRAM dies in a three dimensional structure. This architecture offers a high internal link bandwidth between stacks, and offer low energy consumption benefits. The energy savings stem mainly from the opportunity to be able to move computation much closer to the memory, on the logic layer integrated in the 3D stack, as shown in figure 1.3. This helps save energy otherwise wasted in shipping data from the memory store to the compute core, and to transfer the updated data back to the memory for permanent storage. Such was the paradigm in previous processor-centric architectures, where compute cores would request data from the shared DRAM memory store, over a communication channel.

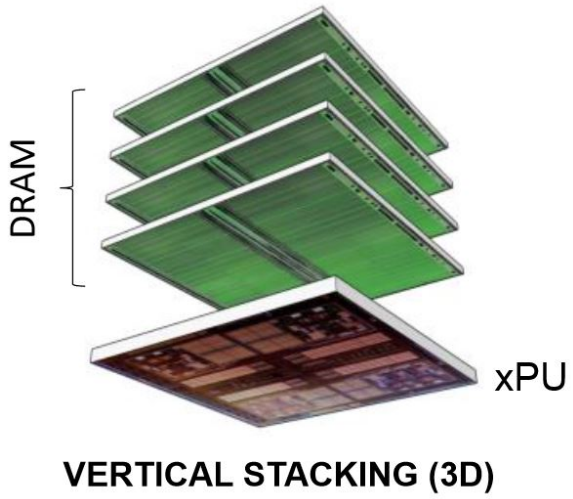


Figure 1.3: 3D stacking for Near Data Processing (NDP) [7]

Thus now basic tasks such as Read-Update-Write can be associated with memory and may be performed very close to the memory die.

1.3.1 Hybrid Memory Cube Architecture

Hybrid Memory Cube (HMC) is a 3D-stacked memory architecture proposed by one of the industry leaders, Micron. This architecture proposes the integration of a network and logic layer under multiple DRAM dies in a 3D-stacked structure. The dies are vertically connected by means of Through Silicon Vias (TSV), or vertical communication channels. The TSVs provide high internal communication bandwidth between memory and the controller when compared to 2D structures for similar memory densities.

HMC introduces significant improvements in access bandwidth, lower chip area footprint and lower power consumption when compared to DRAM. Although in order to achieve these benefits, memory requests cannot be naively sent to the device. For example, access patterns and read-write request ratios and communication overhead between the HMC and host play a crucial role in determining the performance benefits that may be achieved for a target application.

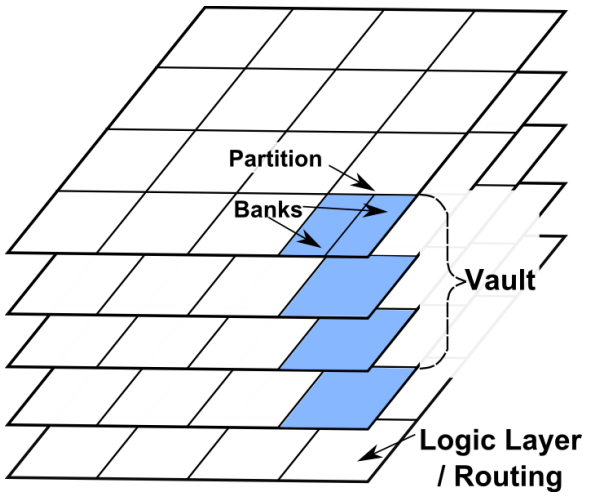


Figure 1.4: Memory organization in Hybrid Memory Cube (HMC).

(a) Memory organization

A HMC memory may be understood by breaking it down into constituent building blocks. The chip on board our evaluation platform (See Section 5) accelerator board is HMC Consortium 1.1 compliant. As shown in figure 1.4, it consists of 8 DRAM dies with one logic layer. The 3D structure is divided into 16 vertical sections, known as vaults. Each vault has a dedicated memory controller in the logic layer to handle vertical requests to any of the DRAM stacks. Four such vaults in the same physical vicinity are logically grouped together, and are known as a quadrant. The four vaults of a quadrant usually share a common external link. Each layer within a vault consists of 2 banks. This means that our HMC device has 256 banks of DRAM.

$$\begin{aligned} Banks_{(spec1.1)} &= 8 Memory stacks \times 16 vault/stack \times 2 banks/partition \\ &= 256 banks \end{aligned}$$

In each subsequent iteration of the HMC specification, the memory partitions are made more dense.

(b) Interface protocol

HMC works over a serial link, employing a packet based communication protocol. Thus in contrast to conventional parallel DRAM interfaces, HMC utilizes serialization and de-serialization (SerDes) modules. Thus it enables us to achieve higher bandwidth over HMC links at the cost of higher power usage for SerDes. A packet in this communication protocol is usually of a 16B or higher granularity. These packets are called ”Flits”. Special header and tail flits carry header information to ensure packet integrity and flow control for communication.

A HMC is usually connected to a host via two or four external links, as shown in figure 1.5. Each link is an independent link to a quadrant in the HMC. The quadrants within a HMC are internally connected via a crossbar, which serves as a router between the external links and the distributed vaults. Routing latency is lower for packets with vault destinations in the same quadrant, and higher for packets with vault destinations in other quadrants.

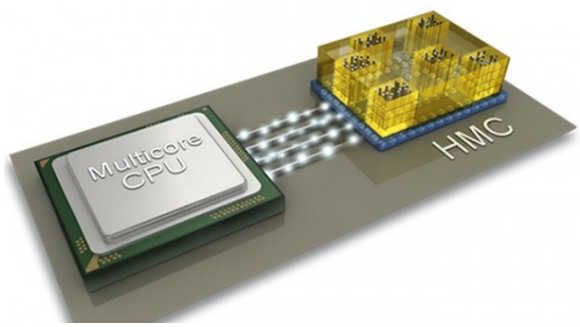


Figure 1.5: Serial links between CPU and HMC [8]

CHAPTER 2
BACKGROUND STUDY

An extensive study into the existing widely accepted state of the art implementation for our target application, LU decomposition for large order sparse matrices with SUPERLU, is necessary to understand possible optimization. This chapter discusses a deeper analysis of the LU decomposition algorithm in the SUPERLU_DIST package to better understand the opportunities for parallelism on an offload accelerator or GPU platform. This is also followed by a literature survey of prior recent work with similar approaches to mitigate the memory bottleneck by hardware improvements.

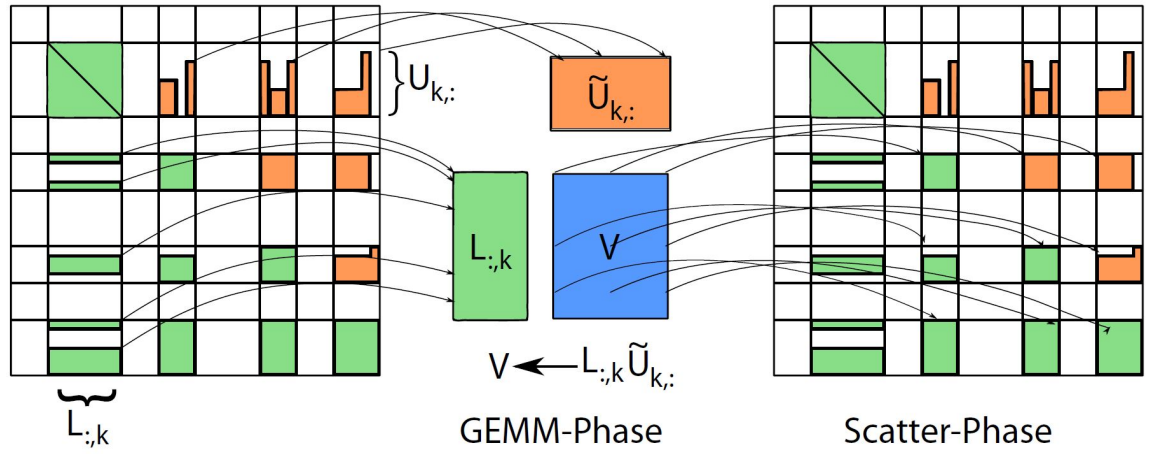
2.1 Related Work

For the target application, LU decomposition, numerous efforts in the past have been made in order to increase the performance of relevant HPC primitives such as matrix-multiply and related SpMV kernels. Some focus on improvement in the algorithmic implementations to increase the occupancy of the compute units, reduce memory transfer invocations, and/or hide memory access latency to the maximum possible extent. Other approaches emphasize hardware improvements to reduce the absolute memory access costs and the peak achievable bandwidth.

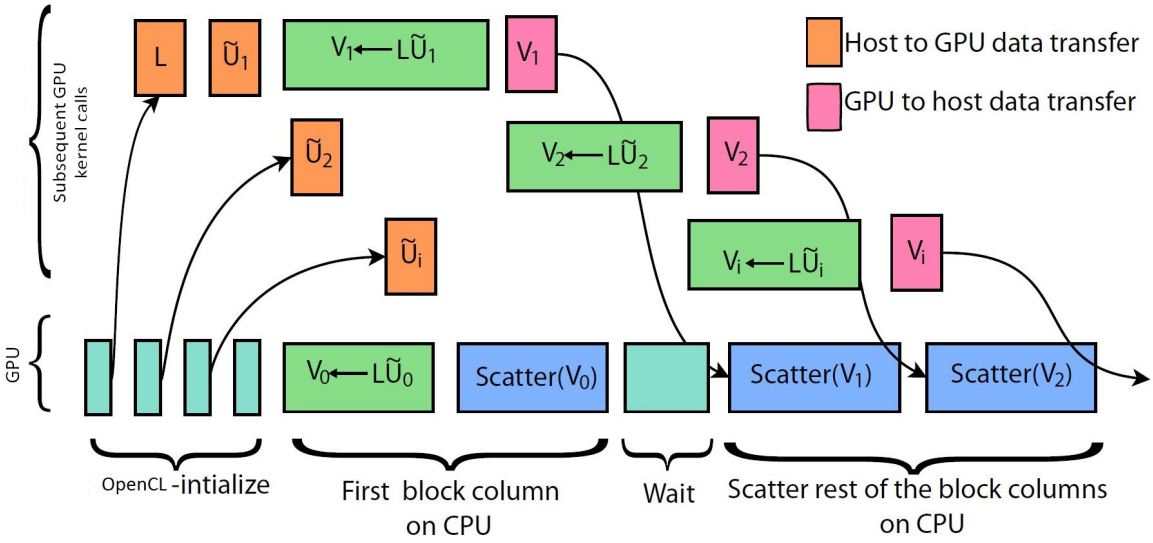
2.1.1 Algorithmic Optimization of SuperLU

Most implementations usually invoke a phase to perform matrix row and column manipulations to create favorable fractal formations without changing the properties of the matrix. This is followed by multiple DGEMM (Double-precision Matrix Multiply) API calls mapped to a GPU or other accelerator for operations on dense data fractals of the otherwise sparse matrix. This phase, sometimes referred to as the Schur Complement, is the most common

choice for optimization algorithms to improve the efficiency of the algorithm from a compute efficiency point of view. We tackle this phase from the memory system's perspective by employing a combined FPGA and HMC architecture to exploit potential benefits.



(a) Scatter DGEMM operation to smaller dimension dense matrix block DGEMM calls



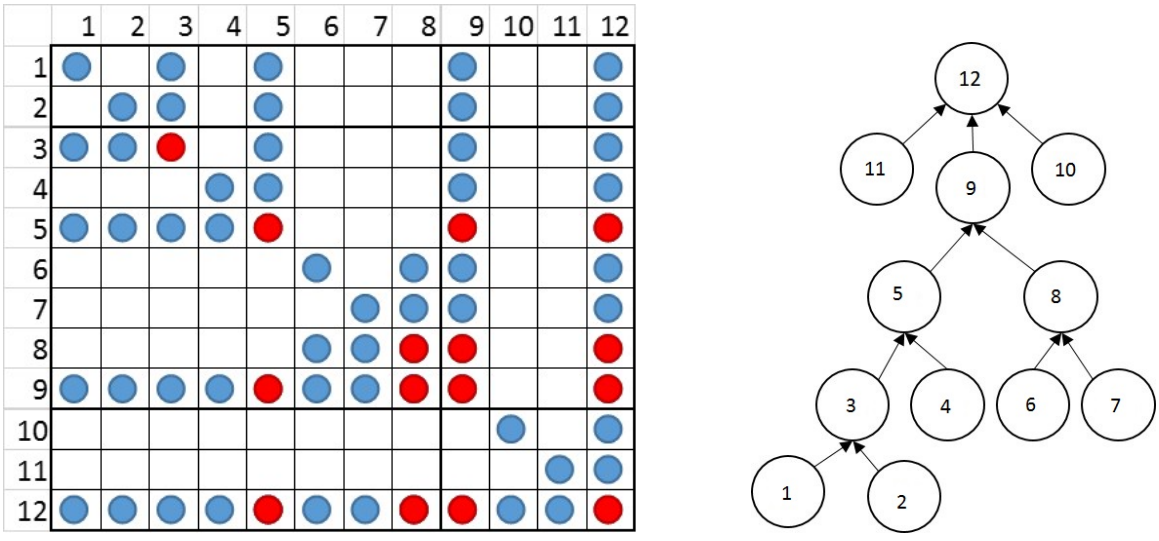
(b) Using Data Transfer idle time on host for DGEMM compute

Figure 2.1: Asynchronous data transfer and compute time sharing. Adopted from P.Sao, R. Vuduc, X. S. Li, Euro-Prar, 2014 [4]

Efforts like those of Sao, et. al [4] aim to optimize this phase in order to speed up the overall computation. One of the approaches to do so is by distributing the workload across multiple compute nodes or GPU lanes or threads to overlap the embarrassingly parallel computations, like those of Schur Complement updates of matrix blocks. The authors of

this work do so by presenting the first hybrid MPI+OpenMP+CUDA implementation of a distributed memory, right-looking, asymmetric sparse direct solver for sparse LU using static pivoting.

Another state of the art approach is one which reduces the wall clock time of the previous approach considerably by overlapping the data transfer overheads with SC (Schur Complement) update computations. This is the approach taken in the Highly Asynchronous Lazy Offload (HALO) optimization that was added an implementation of SUPERLU_DIST [4]. The HALO approach is described in figure 2.1. The updates to matrix blocks in memory are scheduled in parallel to manipulation of other independent blocks prepared in a prior step in the form of a tree data structure. The nodes of the tree, as shown in figure 2.2, represent an independent sub-block ready for a SC update. The parent of a node represents a dependency relationship on the result of its child's SC update before the SC update can occur for a parent sub-block. Thus the partial results of the computation are stored on the respective offload accelerator/GPU unless deemed absolutely necessary as a dependency for the SC update of the parent sub-block node.



(a) Sparse matrix

(b) Dependence tree

Figure 2.2: Dependent sub-blocks are coded in red, and independent sub-blocks that may be scheduled for Schur Complement update in the same iteration stage are coded in blue. The dependence tree shows independent blocks that may be processed for Schur Complement in the same iteration stage.

Another crucial contribution of the work in [1] is in minimizing size of such data transfers between the CPU and the GPU local memories. This is made possible by sharing the computation overhead between the CPU (host) and the GPU (accelerator) in a proportion (chosen dynamically) such that the overall time taken by the two to process their chunks is the same. Thus only the required sub-group of the U-panels and the L-panel, as shown in figure 2.3 is shipped off to the GPU, instead of the entire matrix or the entire U-panel as in case of naive sparse LU direct solver implementations. Naive accelerator implementations offload all the BLAS calls for operations such as DGEMM to the GPU (accelerator) leaving the CPU (host) idle while it waits to receive the final results. This is further aggravated by the need to ship entire U and L block panels to the accelerator, increasing data copy overheads.

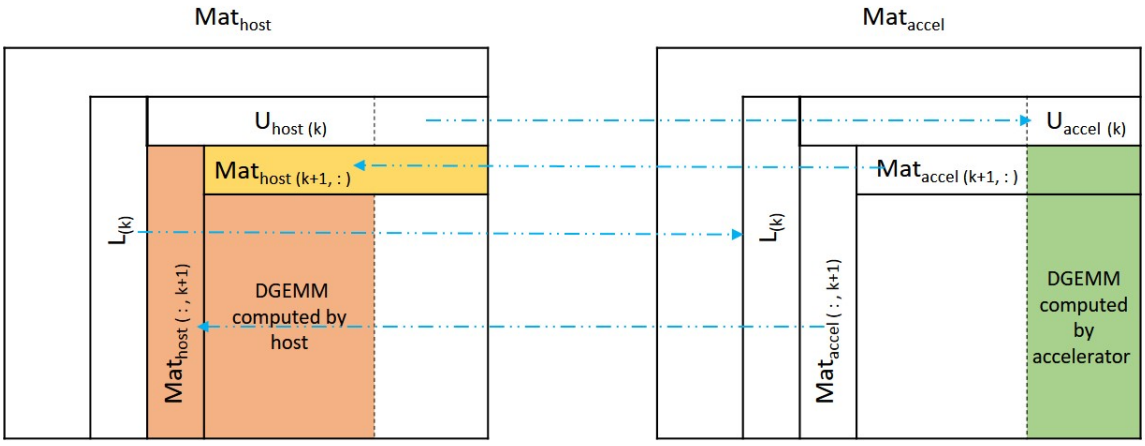


Figure 2.3: Schur Complement update in k-iteration. The L(k) and U(k) panels, calculated in k-th panel-factorization on the CPU, are sent to the GPU. The GPU sends (k + 1) matrix-panels to the CPU. The CPU and GPU update parts of the k-th Schur-complement. The DGEMM result for areas marked in orange is computed by CPU and the one in green is computed by GPU. The CPU merges the received GPUs (k + 1) matrix-panels with its own (k +1) matrix-panels, before the (k +1) iteration starts. Adopted from P.Sao, R. Vuduc, X. S. Li, Euro-Prar, 2014 [4]

2.1.2 Hardware Approaches for Data Movement

Another approach taken by [9] suggests augmenting the hardware to reap benefits. They propose using the logic layer of the 3D-stacked memory architecture to implement a data

restructuring engine for irregular access patterns, that would ultimately reduce the volume of data to be transferred to the compute core on the memory bus. The proposal discusses a hardware engine that acts as a hardware offload unit for a CPU requesting data reorganization for a data structure with large strides or a statically determinable irregular access pattern, to a cache friendly layout buffer. This hardware engine takes over the responsibility for address translation and coherence housekeeping, while moving data between DRAM banks local view buffer which is ultimately streamed to the CPU. The high package internal bandwidth of the stacked memory helps achieve lower latency for data reorganization compared to other prior similar approaches on the host.

Although the approach seems to offer many benefits for sparse access streams with fixed access patterns, the limiting factor of the proposal is its dependence on scatter/gather and DMA (Direct Memory Access) kernels to deliver a static irregular access stream to the hardware engine. This may limit opportunities to optimize for dynamic access streams that are dependent on previously accessed data values. It does however reinforce the need for data movement and access primitives like scatter and gather on these types of architectures.

CHAPTER 3
IMPLEMENTING THE TARGET APPLICATION FOR NDP

This chapter talks about the design of the OpenCL implementation of the target proxy application that is suitable to run on CPU, GPU, and FPGA. OpenCL is selected as the target language due to its versatility across platforms and new compilers that support FPGA hardware compilation via High-Level Synthesis (HLS). We discuss the assumptions that are used to implement the matrix data structure representation in memory while implementing the proxy SUPERLU or LU decomposition kernel. We go into more detail about the different computation phases of the algorithm and the parallelism opportunities within. Finally we highlight the libraries used for efficient BLAS (Basic Linear Algebra Subroutines) calls on the CPU and the GPU platforms.

3.1 SUPERLU with HALO

For the purpose of studying the maximum achievable benefit from the HMC near-memory SUPERLU implementation, we use a dense matrix as our initial input kernel. This is an assumption made in order to maximize the number of memory accesses requests made to access the data elements, as compared to the house keeping overhead information such as data indexes that are used with sparse matrix representations. This worst case scenario model for the HMC memory model provides a counterpoint to the competing baseline model of a GPU accelerator. Normally a GPU would have a large data parallelism potential and the opportunity to hide memory access latency with an abundance of compute scheduling. Thus an improvement for the worst case scenario of dense matrix input would make the best argument further exploring memory models supporting NDP architectures for HPC, particularly for sparse LU implementations.

This proxy HALO implementation in OpenCL, is then used to obtain memory traces

representative of the application behavior which are then used for simulation using the GUPS HMC framework to make application performance estimates (Ch. 6).

The final SUPERLU implementation for sparse matrices and HALO data movement implemented in OpenCL is kept as a final piece of the work for testing on the real FPGA hardware interfaced with HMC memory chip. While this thesis does not examine this in detail, this implementation will be used to study real achievable numbers of performance improvements for sparse matrices. A respective vendor toolchain to convert the OpenCL solver implementation for sparseLU to an optimum verilog design for prototyping on the FPGA is later discussed in Ch. 5.

3.1.1 OpenCL SUPERLU Implementation

SUPERLU with HALO can be summarized as multiple progressive iterations of 3 phases, namely panel-factorization, DGEMM (Double Precision Matrix Multiplication) and SC (Schur Complement) update. A detailed explanation of these phases can be found in the following paper [1].

Since our argument is for a PIM or NDP architecture acheived by moving the accelerator to the logic layer of a 3D-stacked memory chip, we are only concerned with the memory access stream of the BLAS calls offloaded to the accelerator. Thus for our discussion of the SUPERLU_DIST package [4], the distributed memory model of a cluster of hosts interacting with MPI calls is not of our utmost concern. Thus, for simplicity, in our proxy application, we consider interaction between a single CPU (host) and a GPU (offload accelerator). For this reason, the phase that interest us is the DGEMM phase, which is offloaded to the GPU.

To discuss the DGEMM phase, we must understand how the data is prepared for the GPU and the asynchronous lazy offload calls, which help reduce the data transfer overhead costs in HALO. During the initialization phase, the GPU kernel is allocated a section of the memory (A^{ϕ}) initialized to all zeros. This memory serves as an analogous representation of the input matrix (A), where the computed partial DGEMM products are superimposed.

To understand this, consider a matrix $A_{host}(i,j)$ and its analogous copy on the accelerator (GPU), $A_{GPU}(i,j)$. $A^0_{host}(i,j)$ denotes the initial value of $A_{host}(i,j)$. As descibed above, we initialize $A_{GPU}(i,j)$ with zero values. In an iteraion $k < min(i,j)$, HALO updates either $A_{GPU}(i,j) \leftarrow A_{GPU}(i,j) - L(i,k).U(k,j)$ on the GPU or $A_{CPU} \leftarrow A_{CPU} - L(i,k).U(k,j)$ on the CPU. Let α_1 denote the set of iterations in which $A_{GPU}(i,j)$ is updated on the GPU, and α_2 denote the iterations when $A_{CPU}(i,j)$ is updated on the CPU. Then, the images of $A_{CPU}(i,j)$ and $A_{GPU}(i,j)$ can be represented by:

$$A_{GPU}(i,j) \leftarrow -\sum_{k \in \alpha_1} L(i,k)U(k,j) \tag{3.1}$$

$$A_{CPU}(i,j) \leftarrow A^0_{host}(i,j) - \sum_{k \in \alpha_2} L(i,k)U(k,j) \tag{3.2}$$

If we added $A_{GPU}(i,j)$ to $A_{CPU}(i,j)$, it would lead to the same result as updating $A_{CPU}(i,j)$ each α_1 iterations. i.e.,

$$\begin{aligned} A_{CPU}(i,j) &\leftarrow A_{CPU}(i,j) + A_{GPU}(i,j) \\ &= A^0_{host}(i,j) - \sum_{k \in (\alpha_1 \cup \alpha_2)} L(i,k)U(k,j) \end{aligned}$$

Thus before the $k = min(i,j)^{th}$ iteration begins, we can query for the $A_{GPU}(i,j)$ block and add it to $A_{CPU}(i,j)$. The resultant $A_{CPU}(i,j)$ block contains updates from all $\alpha_1 \cup \alpha_2$ iterations. Thus, when in the $k = min(i,j)^{th}$ panel-factorization stage, the images of the $A_{CPU}(i,j)$ block in the GPU offload case and the non-offloaded case are the same. This analysis holds true for all the blocks in the k^{th} panel-factorization stage. Hence, the factored $L(k)$ and $U(k)$ panels in the GPU offloaded case are the same as they would have been in the vanilla non-offloaded case.

Post panel-factorization phase, the burden of the SC update call is shared by the host

and the GPU. The HALO algorithm schedules the matrix multiply operation of the lower triangular matrix panel (L-panel) and the upper triangular matrix panel (U-panel), required for the SC update step, over the host and the GPU in an efficient way. Thus one portion of the result matrix is obtained as the result of a BLAS library call on the CPU while the remaining is scheduled on the GPU. The GPU computes the result of the DGEMM operation and superimposes it over the the local GPU memory representing the analogous section of the input matrix. This is critical to the data transfer savings promised by HALO. This avoids the need for transferring the entire computed result to the host for each GPU call. Thus the GPU keeps accumulating the results of the partial L, U panel multiplications for the subsequently scheduled DGEMM operations. This results in the GPU local memory (A^{ϕ}) serving as a temporary storage for the compounded summation of the partial L, U panel products. A data transfer request call to the GPU is thus triggered by the host only when it is absolutely necessary to apply the sum of the computed partial products of the L, U panels to the respective block in the host memory store. This is done in order to complete the SC update of a block on the host before scheduling it for panel factorization on the host. Thus the data transfer overhead costs are reduced significantly.

In order to achieve the maximum GPU occupancy, HALO uses the elimination tree as described in the paper to identify the blocks which can be processed in the Schur Complement (SC) update stage in parallel in k^{th} iteration. Thus at any point of time, multiple blocks panels could be issued for computing DGEMM products if they are independent nodes on non-overlapping branches of the elimination tree. This helps overlap DGEMM computation of k^{th} iteration with data transfer calls invoked for completing SC update to a block, to be issued for panel factorization in $k + 1^{th}$ iteration. Thus hiding memory access latency for fetching the computed result from GPU. It should be noted that this in no way helps the GPU to hide the local GPU memory access latency. That is strictly dependent on the number of blocks scheduled at the moment on the GPU, for DGEMM. Thus our argument for a HMC memory model contends for a quicker result computation on the GPU.

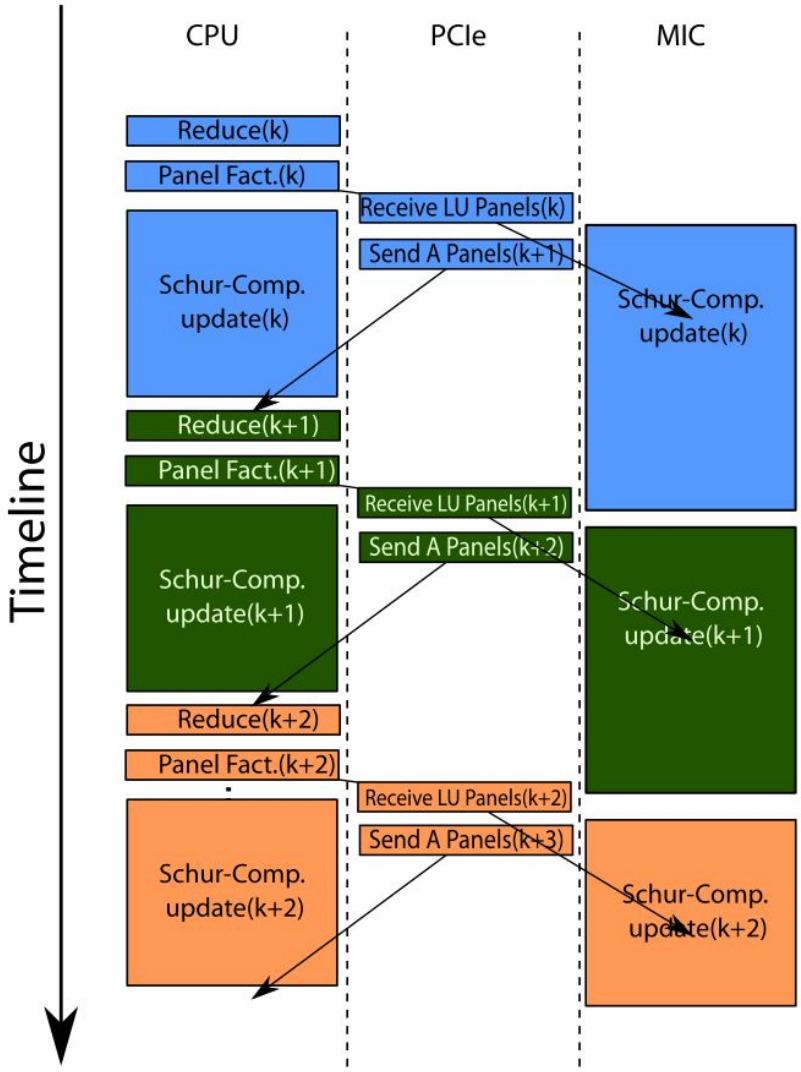


Figure 3.1: Concurrent execution of the remainder of Schur complement update for blocks $A(k)$ on the GPU after transfer of required panels $A(k+1)$ to CPU. Allows for concurrent factorization of subsequent independent block $A(k+1)$ on CPU. In general, the Schur complement update is much longer than both other steps and data transfer. Adopted from P.Sao, R. Vuduc, X. S. Li, Euro-Par, 2014 [4]

Another important aspect is that this implementation is for a strict comparison of a portable application. This means that it is compatible with both the hardware platforms, one with a GPU using a DDR3 memory subsystem, and another with a GPU/accelerator using a HMC chip. Thus the benefits of dynamic HMC address mapping (as discussed in the section 1.1) by rewriting the accelerator (GPU) kernel cannot be exploited. We cannot

avoid the need for multiple local partial copies of the product on different local memories of SMXs (Symmetric Multi-streaming processors).

3.2 BLAS libraries

Our implementation includes standard BLAS libraries in order to perform linear algebraic manipulations on matrices in an efficient manner. We invoke CBLAS library calls on the host using the OpenBLAS runtime package bound dynamically to the executable. Similarly we include the clBLAS runtime library package to support efficient BLAS implementations in OpenCL kernels running on an accelerator (CPU, GPU, etc.).

3.2.1 clBLAS

This library [10] houses efficient OpenCL kernel implementations of BLAS level 1, 2 and 3 routines. The library supports running on CPU devices to facilitate debugging and multicore programming. Thus dedicated implementation for efficient OpenCL BLAS execution on CPU is assured. Importing this library helps us implement an efficient proxy application for the accelerator capable of representing a tuned accelerator kernel. This is irrespective of the actual hardware platform chosen for the accelerator. Thus assumptions may be taken in the thus obtained memory trace to expect maximum data parallelism.

The PICO vendor toolchain discussed for FPGA compilation in Ch. 5 has only limited support for BLAS call translations to efficient Verilog design implementations. Thus for final execution of the sparseLU application on FPGA interfaced with a HMC memory, we may need to implement a custom OpenCL implementation for DGEMM.

3.2.2 OpenBLAS

This library houses efficient multi-thread, multi-core implementations of BLAS level 1, 2 and 3 routines. This ensures efficient manipulation of the matrix data structures during BLAS calls on host. Its significance is that while generating the memory trace (Ch. 4, we must be careful of these contending threads executing BLAS calls because they may make requests to the same blocks of memory simultaneously.

CHAPTER 4

EVALUATING MEMORY BEHAVIOR WITH MEMORY TRACES

Since our FPGA and HMC Near Data Processing platform currently has limited support for running BLAS operations that are key to the proxy application, we use this chapter to discuss how memory traces are generated and used with our FPGA framework to test the memory access characteristics of our kernel. A memory trace refers to a record of attributes of all the memory access requests made by the hardware, including the size of transaction, access type (read/write), address, etc. The possible approaches to get the trace include binary instrumentation or executing the binary through a simulation framework with a custom memory model. The span of memory requests which are of interest to us, is too wide over the lifetime of the executable in our case. This is so because the DGEMM call invocations are distributed across multiple iterations of the LU factorization algorithm. Thus a simulation framework modeling the host and the accelerator pipelines through memory stages would simply take too long to simulate the entire program to generate the memory trace. Thus the alternate approach for binary instrumentation is adopted.

4.1 Hardware platform

At the time of this work, there was a lack of a stable compatible binary instrumentation tool to record memory trace for a binary running on a GPU hardware. Most tool chains exist for recording memory trace for an instruction trace running on a simulator frameworks for a GPU platform. Only a few dynamic instrumentation tools, like ”lynx” [11] exist for GPU platforms. But these tools support instrumentation for only a small subset of NVIDIA GPU architectures of the ages past. Recent tools such as GT-Pin [12] and GPUPROF lack the necessary documentation and support for experimentation and reliable usage on more recent GPU architectures. They are also often vendor restricted i.e. they work for only select

vendor platforms; GT-pin is restricted for only Intel GPU platforms and currently exists only in beta format.

For this reason, we choose a stable CPU platform capable of executing OpenCL kernels coupled with the PIN instrumentation tool [13]. The supported runtime packages on Intel CPUs provide efficient multi-threaded implementations for multi-core processor platforms.

We use the Intel(R) Core(TM) i7-4790K CPU processor clocked at 4.00GHz for our experiments to generate the memory traces. The generation of a memory trace for the proxy application with a square matrix of dimension 1000 takes ˜25 GB of storage space and 4 hours instrumentation runtime.

4.2 PIN tool

Pin [13, 14] is a dynamic binary instrumentation framework by Intel for their IA-32, x86-64 and MIC instruction-set architectures. The tool enables creation of dynamic program analysis tools or ”pintools” for user space applications. Instrumentation is performed at run time on the compiled binary files. Thus, it does not require recompiling of source code and can support instrumenting programs that dynamically generate code. It allows context information such as register contents to be passed to the injected code as parameters.

The tool provides a rich API library which allows for specialized filtering of information to be monitored for the executing binary. It enables us to dynamically modify the target binary application on the fly through the static definition of procedure calls that are inserted dynamically into the instruction stream based on instruction type. This is depicted in figure 4.1.

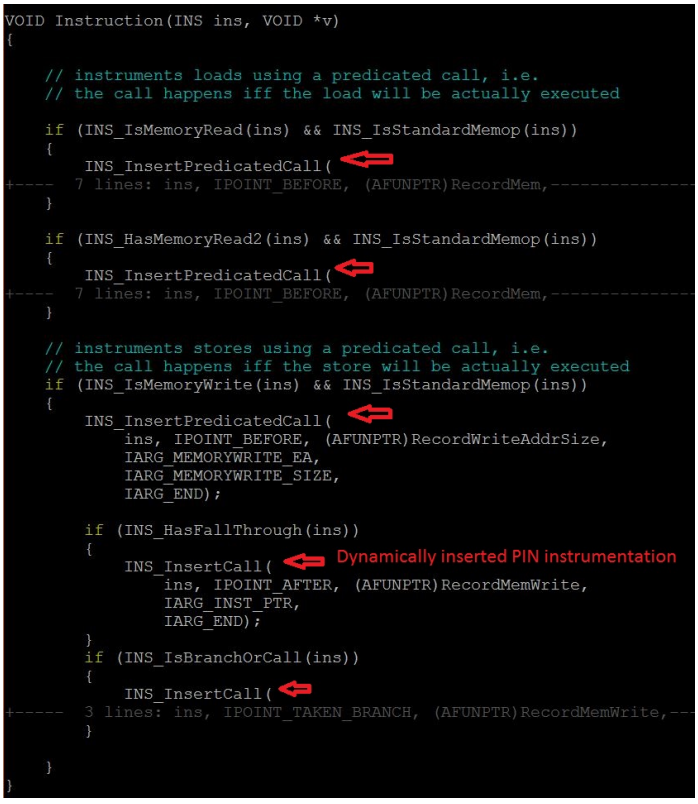


Figure 4.1: Dynamic binary instrumentation by probing instructions and inserting predicated calls.

The tool fidelity encompasses:

1. The granularity of instrumentation: instruction, basic blocks, trace
2. Information to extract: memory references, instruction pointer information, routine name, instruction image name
3. Multi-thread primitives, etc...

Our focus is to obtain the memory trace for the offloaded DGEMM routine invoked from the clBLAS library. Thus we need to focus on the OpenCL kernel execution on the Intel CPU. Hence, we concentrate our efforts to isolate the memory accesses initiated from the instruction stream that belong to the dynamically linked Intel OpenCL runtime package images. Similarly, we trigger the instruction level instrumentation only after invocation of the proxy application routine. And we end the instrumentation, as soon as we return with

the factored LU result. This is shown in figure 4.3. The entry and exit point of the proxy are identified using the dis-assembly dump of the binary as shown in figure 4.2.

4.2.1 Memory Trace Pintool Filter

Pintool refers to the filter applications developed using Pin to focus the scope of instrumentation on the instruction stream which is of deep interest. In order to isolate memory access requests to generate a trace, the technique involves two logical steps. These include identifying the instruction type to pinpoint memory operations that interest us, and then isolating the required information attributes from the instruction header.

For standard memory manipulation instructions which perform simple ”loads” and ”stores” on memory operands, predicated calls for instrumentation are inserted. These trigger a routine which records the the details if the memory operation is indeed performed. The attributed recorded include the instruction pointer, memory address, request size, type of operation (read/write), and the actual probed data.

For other standard memory access instructions initiating write requests on its operands or for conditional codes making control decisions, a respective instrumentation call for tracking the write update to the AFUNPTR pointer is inserted. This helps the profiler track the path taken by the branches for subsequent calls.

4.2.2 Custom Pintool Filters

In order to maintain a good degree of performance for the instrumentation phase, and to restrict the amount of pre-processing required for the memory trace to extract the OpenCL kernel memory references, we limit the scope of the instrumentation with a filter.

We generate a dis-assembly of the compiled binary to identify the memory address of the instruction invoking the proxy application routine. Similarly the memory address for the return instruction for the target is noted. These instruction addresses are hard coded in a custom pintool which triggers instruction level instrumentation only when the routine for

the target application is executed. The instrumentation is disabled once a return from the routine is acknowledged by the instruction pointer. This also triggers other house keeping actions such as closing the trace file pointer and exiting the application. This is shown in figure 4.3.

```
0000000000403926 <_Z8dense_LUPdRP7_cl_memiiRP11_cl_contextRP17_cl_command_queue>:
  403926:   55                      push   %rbp
  403927:   48 89 e5                mov    %rsp,%rbp
  40392a:   41 57                   push   %r15
  40392c:   41 56                   push   %r14
  40392e:   41 55                   push   %r13
  403930:   41 54                   push   %r12
  403932:   53                      push   %rbx
  403933:   48 81 ec a8 08 00 00    sub    $0x8a8,%rsp
  40393a:   48 89 bd 58 f7 ff ff    mov    %rdi,-0x8a8(%rbp)
  403941:   48 89 b5 50 f7 ff ff    mov    %rsi,-0x8b0(%rbp)
            :
            :
  406070:   48 89 d8                mov    %rbx,%rax
  406073:   48 89 c7                mov    %rax,%rdi
  406076:   e8 75 c3 ff ff          callq  4023f0 <_Unwind_Resume@plt>
  40607b:   e8 30 c2 ff ff          callq  4022b0 <__stack_chk_fail@plt>
  406080:   48 8d 65 d8             lea    -0x28(%rbp),%rsp
  406084:   5b                      pop    %rbx
  406085:   41 5c                   pop    %r12
  406087:   41 5d                   pop    %r13
  406089:   41 5e                   pop    %r14
  40608b:   41 5f                   pop    %r15
  40608d:   5d                      pop    %rbp
  40608e:   c3                      retq
```

Figure 4.2: Disassembly of the proxy target application. Helps identify the memory address for entry and return instructions for the OpenCL kernel subroutine.

```
VOID Trace(TRACE trace, VOID * val)
{
    if (!filter.SelectTrace(trace))
        return;

    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins))
        {
            if((INS_Address(ins) == 0x403926)){   Entry point to proxy application
                PIN_MutexLock(&thread_lock);
                  flag_inst = true;
                PIN_MutexUnlock(&thread_lock);
            }
            PIN_MutexLock(&thread_lock);
              out << hex << setw(8) << INS_Address(ins) << " ";
            PIN_MutexUnlock(&thread_lock);
            RTN rtn = TRACE_Rtn(trace);
            if (RTN_Valid(rtn))
            {
                IMG img = SEC_Img(RTN_Sec(rtn));   probing for OpenCL kernel calls
                if (IMG_Valid(img)) {
                  if((strstr(IMG_Name(img).c_str(), "intel") != NULL) && (flag_inst ==
                    out << IMG_Name(img) << ":" << RTN_Name(rtn) << " " ;
                    Instruction(ins, NULL);
                  }
                }
            }
           PIN_MutexLock(&thread_lock);
             out << INS_Disassemble(ins) << endl;
           PIN_MutexUnlock(&thread_lock);

             if(INS_Address(ins) == 0x40608e){   Exit point from proxy application
               PIN_MutexLock(&thread_lock);
                 flag_inst = false;
               PIN_MutexUnlock(&thread_lock);
                 cout<<"Instrumentation ended by PIN TOOL!\n";
                 TraceFile<<"Instrumentation ENDS!!!\n";
                 out<<"\n----------------------------\n";
                 out<<"Instrumentation ENDS!!!\n";
                 TraceFile.flush();
                 // return;
                 exit(0);
             }

        }
    }
}
```

Figure 4.3: Custom pin tool to generate memory trace recording requests issued by target application OpenCL kernel

Once instrumentation begins, instruction traces from the runtime package images that are linked dynamically to the binary are forwarded to the pintool for analysis. As the application progresses, basic blocks from the images are analyzed by executing instructions one by one in the pintool environment. Only the images for OpenCL runtime packages are probed further, to isolate memory requests. This ensures a memory trace record for memory transactions generated by only the offloaded OpenCL kernels.

An important design consideration accounts for the multi-threaded nature of the executing binary. Thus a careful use of mutex locks provided by ”pin” is necessary to avoid obfuscated trace records where multiple competing threads contend for access to the file pointer to flush their stream contents. This in turn also negatively impacts the performance of the probed application on the hardware because a lock contention for the trace file pointer is generated by each thread for every memory operation. But this is a necessary cost we must absorb for meaningful results.

4.3 Limitations

The chosen hardware platform (CPU) immediately poses restrictions on the meaningful information that can be extracted out of the memory traces due to its limited number of hardware threads. The support for multi-thread runtime packages for OpenCL kernel execution allows for multiple independent threads to be issued on the CPU, which may be seen as emulating concurrent OpenCL workitems. Similarly, multi-core runtime support for OpenCL kernel execution provides a heavy-weight concurrent execution similar to that at the SMX or threadblock level on the GPU. This is another reason for choosing clBLAS as the BLAS runtime library for accelerator kernel implementation. This enables us to emulate a platform tuned OpenCL kernel over a wide range of accelerator hardware platforms such as Nvidia GPUs and Intel CPUs. But ***because of the inherent serial scheduling nature of a thread on a CPU, the information about concurrent parallel execution of workitems within a wavefront is lost.*** Thus any possible information about concurrency in the issued

memory requests is lost.

No information about the mapping of independent workitems (belonging to the same wavefront) to their respective counterparts as CPU-schedule thread IDs is passed on through the OpenCL runtime packages. Thus the ***stream of memory requests loses any information regarding restrictions in thread scheduling due to inter-workitem data dependencies***. Therefore, we cannot easily map our memory access patterns to a GPU-like implementation. This may not be a huge loss for traditional OpenCL implementations on FPGA which use heavy-weight pipelines similar to CPU threads.

However, ***intra-workitem data dependencies are preserved*** in the memory address stream. These dependencies within a thread instruction trace may be identified by matching the CPU thread IDs of the suspected latter stream command with the CPU thread id of the immediate previous stream command with the same memory address. But there may be false positives as well, because of the arbitration of lock acquisition as discussed in the subsection 4.2.2. For example, contending CPU threads could delay the recording time point of an earlier transaction in the trace, leading to an inverted dependence.

Similarly, ***no consistency guarantees are preserved in the memory trace*** because of the lack of a tool (and possibly an associated cache model) to be able to represent concurrent memory requests being emulated on the CPU platform. This lack of timing information between subsequent and concurrent accesses as in case of a true GPU execution model, validates a wide range of assumptions that can be made over this obtained memory trace. A sample is shown in figure 4.4.

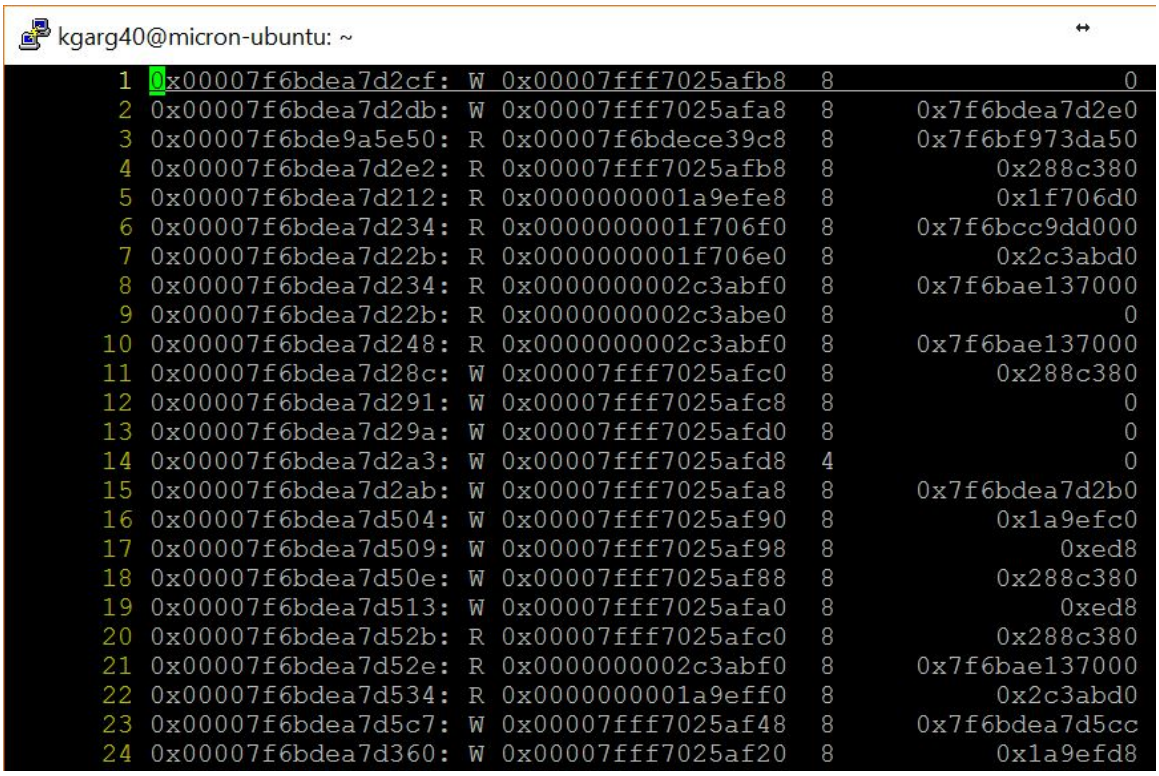


Figure 4.4: A snippet of the memory trace generated by the pin tool. Format: `<InstrAddr>: R/W <memAddr> <size> <data>`

4.3.1 Extrapolations for memory trace for representing GPU trace

Despite the above limitations, these OpenCL-based traces offer an opportunity to explore the performance implication for PIM on HMC by making meaningful extrapolations to the trace data. The serial memory reference stream is a good way to get comparison figures for the overall contribution of the memory system model to the application run time. This provides us with a minimum limit barrier for exploring the parallelism opportunities in the computation which enable us to successfully hide the effect of memory access latency. The lower this scheduling barrier is, the more probable and easier it is to find an independent task to schedule in parallel. This ensures that the result is ready before it is deemed necessary by HALO to transfer to the host, preventing any idle time on the host compute cores. Lower latency for single memory requests and a higher degree of data parallelism in the memory model will also help lower this barrier.

Thus before feeding the trace to our FPGA and HMC-based simulation model, we make the following assumptions.

- Memory requests are issued in a round robin policy to each port represented by a queue, contending for access to a common shared serial link to the HMC. This is representative of multiple compute units (lanes) in an accelerator (GPU) making simultaneous independent requests to the memory subsystem.

- A group of requests (of fixed size) may be issued sequentially in a batch, as independent requests. This is to represent the compute capability of an accelerator allowing it to hide memory access latency while servicing other workgroups, meanwhile queuing new memory requests.

- Serial execution within batches needs to be ensured, representing serial intra-workitem dependencies. That is, this represents one workitem/lane issuing memory requests sequentially.

- Multiple queues sharing the common serial link may be serviced out-of-order in any fashion to achieve maximum link occupancy and average transfer bit rate.

Thus we use the trace limitations as a benefit instead of letting them be a roadblock to our analysis. With these assumptions we make an argument for the 3D memory architecture models, to understand the potential of benefits that may be exploited from them. While at the same time we ensure that the correct functionality and data quality as presented by the memory trace in terms of memory state, is reproduced by our simulations on the HMC by honoring the consistency and data dependencies which are represented by the memory trace.

CHAPTER 5

EVALUATION MODEL AND HARDWARE INFRASTRUCTURE

After we successfully generate a memory trace representative of the application behavior, we switch gears to the performance study and evaluation of the trace on a 3D memory architecture model. We use the AC-510 accelerator board by Pico Computing (now, a part of Micron) that combines a Xilinx FPGA with a HMC (Hybrid Memory Cube), for our evaluations. The platform features a HMC Consortium Specification 1.1 interface between the FPGA and the on-board HMC and a vendor implemented HMC controller IP. This "'Pico Framework"' also allows us to leverage vendor-provided communication via PCI Express and software-readable registers on the FPGA and to express a custom tuned IP for establishing a stream based interface firmware with instrumentation support.

Figure 5.1: The vendor implemented Pico framework software stack gives easy to use APIs to tweak hardware design parameters on the FPGA logic. The stream framework may be used to interact directly with the HMC. Adopted from [15]

5.1 Evaluation Platform

The evaluation platform consists of a host CPU communicating with a backplane card over a x16 PCIe Gen3 (32GB/s) full width link. This card provides a physical medium for the modules to communicate with the rest of the system over PCIe. It can house up to 6 AC-510 accelerator boards, each communicating over a x8 PCIe Gen3 (16GB/s) half width upstream link. The accelerator board, shown in figure 5.2 consists of a Xilinx Ultrascale FPGA (XCVU060) interfaced with Micron's high bandwidth HMC over 2 half width x8 PCIe full duplex lanes supporting 15Gbps transfer rates. Each FPGA module has its own connection to the PCIe network, through which it can communicate with the host CPU. The host CPU must manage dataflow to and from each of the FPGAs and control their processing.

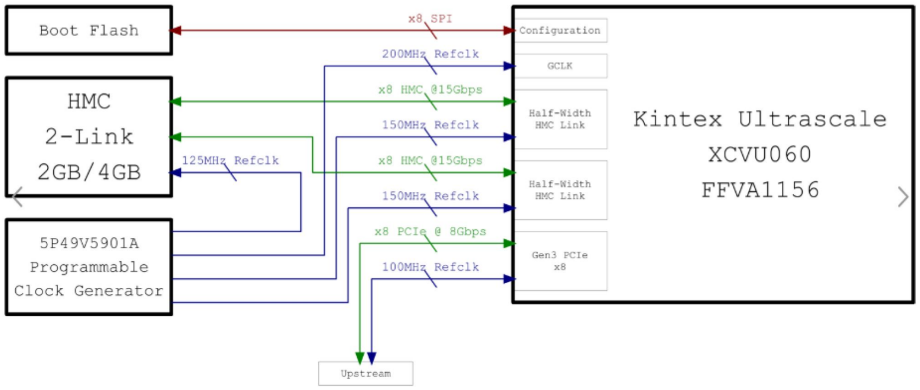


Figure 5.2: FPGA-HMC link architecture. Adopted from [16]

To better understand the system, we split the behavior into 2 aspects: software and firmware.

- Software is comprised of the host CPU code, which consists of our application reading a memory trace and scheduling memory accesses on the FPGA over HMC port queues. This also includes the Pico device driver which provides a ready to use solution to interact with the vendor HMC controller IP on the FPGA.
- Firmware comprises of the Pico HMC controller IP implementation accessible by on-board logic modules using the standard AXI (Advanced Extensible Interface)

bus interface. This is augmented by custom Verilog code to support instrumentation registers and initialization registers to reset performance probing counters. This provides easy to control switches in software for experimentation and to probe metrics such as latency and bandwidth by reading cycle counts and the number of requests issued to user ports.

The knobs provided by vendor and custom firmware can be accessed by means of the Pico Framework API. The vendor framework provides a stream based communication model to a multi port memory interface.

5.1.1 PICO stream framework

A stream is a unidirectional channel for point to point communication that carries sequences of data with flow control information. The Pico API provides a simple to use model to know when the stream is available for reading or writing, and when data can be sent or received using a DMA-like mechanism. The stream firmware on the FPGA is essentially a FIFO interface. This ensures an in-order delivery of command requests to the HMC controller. This enables us to emulate sequential consistency on a user port (emulating a GPU lane/accelerator processing element), modeled by the memory trace. The stream based interface also helps cut down on software overheads incurred while scheduling packets one by one, for transmission over the HMC user ports. This provides a low latency model compared to serial bus models.

As shown in figure 5.3, we set up two streams for each HMC user port. One to forward the command requests to the HMC user port, another to accept responses returned by the HMC controller to the user port. These streams provide the basic functionality to interact with the user ports via software APIs.

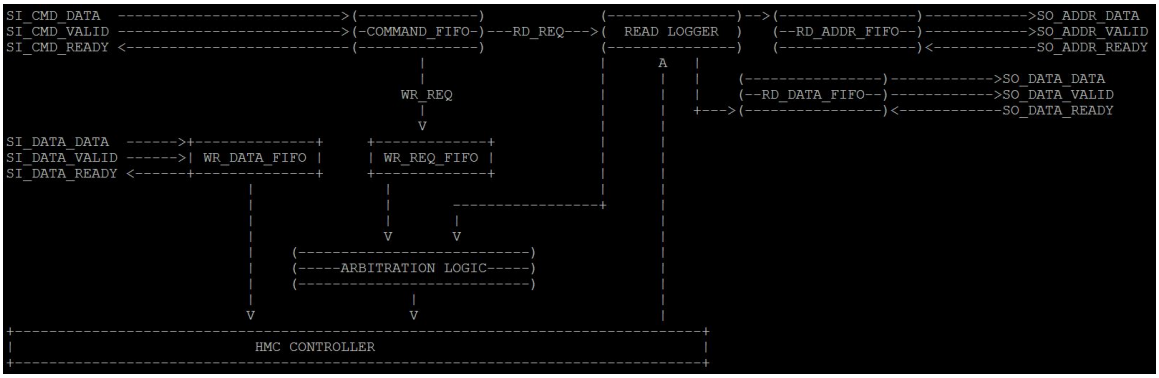


Figure 5.3: Read and Write channels for Command and Data streams using Pico stream framework. Adopted from Pico code documentation.

5.2 HMC addressing

The HMC supports a 34 bit address space, as shown in figure 5.4. This allows for a total of 16 GB of memory on one HMC chip. The current stacked memory on the AC-510 supports only 4GB, so the higher order 2 bits are ignored. Although each memory location refers to a byte of data, the access granularity differs. The lowest granularity of access is 16 bytes. Thus the 4 LSB (Least Significant Byte) bits are masked out during access. For a denser data packet, the granularity of access may be increased to 32, 64 or 128 bytes.

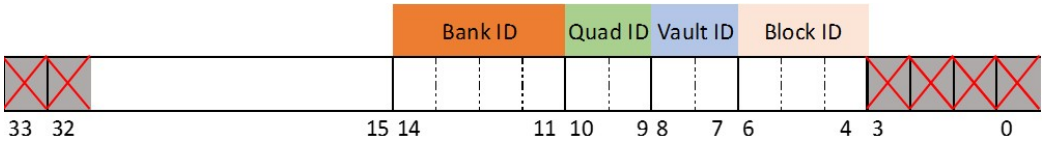


Figure 5.4: The components of the 34-bit address space of HMC.

The bit fields represent the following:

- [3] : 4 LSB bits ignored during access
- [6:4] : address bits masked as per block size
- [8:7] : 2 bits for Vault ID

- [10:9] : 2 bits for Quadrant ID
- [14:11] : 4 bits for Bank ID
- [31:15] : Block Address
- [33:32] : masked for 4GB HMC chip onboard AC-510

The default memory addressing ensures that sequential address memory requests are distributed across vaults first, then to banks, and finally spreading across quadrants. This provides for a high degree of parallel access bandwidth for a sequential address stream. Using this information, the memory allocation in the HMC may be carefully organized in order to optimally utilize the available link bandwidth.

5.3 GUPS framework

The vendor provides a useful application ”GUPS” for the accelerator board platform. The application helps us measure memory system performance, by measuring number of operations or giga-updates performed per second. Hence the name of the application ”GUPS” (Giga Updates Per Second). GUPS can be used to help us make first order approximations on the performance that an application shall observe. The vendor framework generates patterns of random and/or linear memory address requests from the FPGA logic, in a sequence as programmed. This enables them to make a performance estimation for the memory system for an application, using the memory request pattern.

We modify this framework to study the realtime performance of the memory system for a custom memory trace, generated as described in Ch. 4. Using this custom framework, we are able to make accurate predictions of the expected latency and bandwidth for our proxy application by using the previously generated memory trace. We provide support in our GUPS application to remap memory accesses to the HMC's address space in a configurable manner. In addition, we modify the firmware to take it addresses and data from

software-based Pico streams rather than generating them in the hardware as with the normal GUPS design. This allows us to extend our study to understand potentials for performance improvement for both naive as well as custom tweaked implementation of the application on a 3D memory architecture.

For 1st order analysis, we use a random mapping policy to move from the host's 64 bit memory address space (of the memory trace file) to the HMC's 34 bit address space (to be forwarded to the HMC controller IP). We may easily extend these to much more sophisticated mapping schemes in order to localize accesses from a user port to a single vault in order to optimally exploit the data parallelism within a vault, and make use of high internal bandwidth in a HMC. This may also be used to distribute accesses from different user ports to different vault controllers in order to minimize the network flow traffic and congestion on the logic layer of the HMC. This evaluation is currently underway but is not included in this work.

Our tweaked GUPS application is a multi-threaded application emulating the behavior of multiple processing elements (lanes in a GPU warp) issuing independent memory requests simultaneously to a common memory link. This is shown in fig. 5.5. The memory controller in such an accelerator platform coalesces the memory requests received and forwards them off-chip to the memory die. The memory access latency is hidden by servicing compute needs of other workgroups, ready for execution. The new memory requests from these workgroups are then issued in a cascaded manner to the previously pending requests. We emulate a similar behavior by maintaining independent queues of commands ready to be issued to the HMC, for each user port. Independent execution threads represent and control independent user ports on the FPGA. Each thread schedules memory requests over its respective user port, in parallel to the other threads. Thus multiple user ports contend for access to the shared HMC link. While the user port waits to receive a response for the scheduled request, it issues another memory request from the batch queue, with a different tag id. This helps us to emulate the cascading nature of memory requests issued by the

controller while waiting on responses for pending requests.

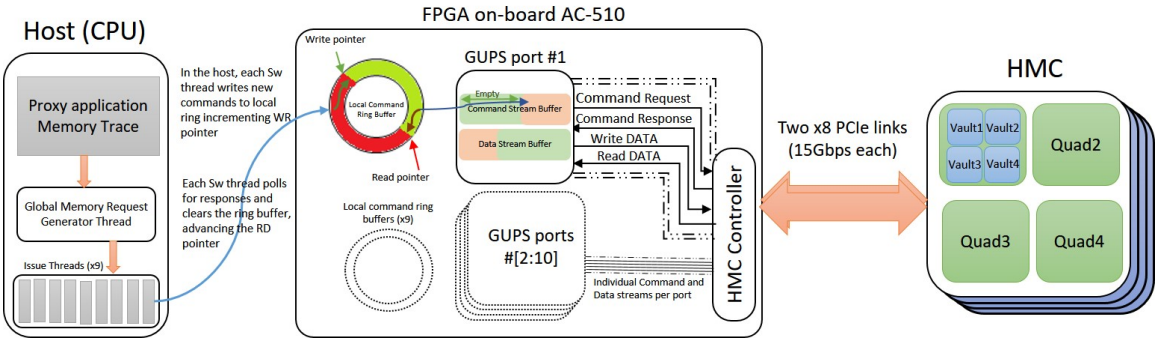


Figure 5.5: The figure depicts the software and firmware modifications used to measure multi-port performance of the HMC. Each software thread controls access to the command and data streams of respective GUPS user ports. Each thread issues batches of commands from the circular command queue to the port, as per depth of empty FPGA port buffer flits.

This batch scheduling of memory requests also helps minimize and overcome the overhead of reading a memory transaction from the trace file and to prepare a respective command request packet for the HMC controller. Apart from these, the overheads in software to schedule a request and probe for response are significant and are much bigger in terms of absolute time than the hardware latency incurred by the FPGA to receive a response from the HMC. Thus precise measurements are made when multiple requests are issued to the HMC. Since the time taken to receive responses for all of them exceeds the polling time in software to probe for responses.

A producer thread spawned from the main thread in the initialization phase, runs in the background to replenish the circular queue buffers of the individual user ports. This thread parses the memory trace file and issues requests sequentially in a round robin fashion to user port buffers, advancing their write pointers. Once the port thread issues a batch of commands to the HMC, it advances the read pointer of the buffer. The producer thread exits only when it has scheduled all the operations from the trace file, setting a global flag on exit. The individual port threads exit only when the local queue's read pointer catches up to the local write pointer and the global flag is read to be true. Thus the program inculcates

a degree of randomness to the individual number of requests issued over each user port, enforcing a robust performance analysis framework.

5.4 Measurements and metric instrumentation

The Pico framework provides a way for real-time instrumentation of performance metrics of the user ports. This is facilitated by means of hardware mapped registers which hold the state of counters implemented in the FPGA logic. Separate 32 bit counters for each user port respectively count the number of read and write requests issued on each port. Other wider 48 bit registers keep track of other timing metrics such as **cumulative cycle count** to service all issued read requests since initialization.

These memory mapped hardware registers are probed by using the PicoBus communication framework. Using the Pico API, we read the value of these registers before spawning port threads, and after they finish execution. This helps us calculate performance metrics such as number of operations performed per second. This **throughput metric** is reported as **GOPS (Giga Operations per Second)**.

Absolute wall clock time taken to service issued requests on a user port is measured in software. We use the `clock_gettime()` API to measure the time taken precisely in nanoseconds. This help us compute **access latency numbers**, and **aggregate latency** to emulate the memory trace file requests on the HMC memory subsystem.

Bandwidth calculations are fairly easy as well. We can simply multiply the number of operations performed with the standard size of each operation. The measurements, shown in fig. 5.6 for our experiments are reported in the next chapter.

```
kgarg40@micron-ubuntu:~/mem_access_charac/software$ ./gups_parl gups_mem_trace --m 0x51
0 --p 2 --ro --batch 350 --f /data2/sample_addr.log
Finding an FPGA matching model = 0x510
Calling WriteRam on memory 0
Running gups_mem_trace
Batch size for issue queue is: 350
Enabled READ commands.
Disabled WRITE commands.
Opening Memory trace file: 0x7fff33153458 to read in the custom memory trace file.
--------------
Request for Issue SPAWN.
Signalled to main.
------------------------------
Opening streams to and from the FPGA on stream ids: 1, 2
Opening streams to and from the FPGA on stream ids: 3, 4
Issue thread id: 0 exiting.
Elapsed time for thread 0 , 3468539 commands is: 853587 usecs.
Issue thread id: 1 exiting.
Elapsed time for thread 1 , 3468538 commands is: 883548 usecs.

------Probed Counters----------------
For i:1
Read value for SUM_RD_Latency: Initial: 0 || Final: 35f7fe74
Read value for Request_READ: Initial: 0 || Final: 34ecfb
For i:2
Read value for SUM_RD_Latency: Initial: 0 || Final: 35cb56a1
Read value for Request_READ: Initial: 0 || Final: 34ecfa
------------------------------
Elapsed Time is: 946731.000000
------Computed Counters----------------
For i:1
READ_requests: 34ecfb
SUM_RD_LATENCY: 35f7fe74
For i:2
READ_requests: 34ecfa
SUM_RD_LATENCY: 35cb56a1
------------------------------
Sum request values across all ports: READ: 7fd0d17b39e0 || WRITE: 7fd0d14de72d
------------------------------
Module 1 GUPS READ: 0.004063
Module 2 GUPS READ: 0.003926
FPGA[0] Aggregate GUPS: 0.007327
All-links-aggregate GUPS: 0.007327
------Self Computed Metrics----------------
For i:1
Module 1 GUPS READ: 0.064472
For i:2
Module 2 GUPS READ: 0.064681
FPGA[0] Aggregate GUPS: 0.064576
------------------------------
kgarg40@micron-ubuntu:~/mem_access_charac/software$
```

Figure 5.6: Printing GUPS application statistics

5.5 Operations supported

The AC-510 module features a HMC Consortium Specification 1.1 compliant interface to the 3D-stack memory die. The specification supports a wide array of command requests. The protocol supports 16B granularity flits for communication over the HMC interface. Shown in figure 5.7 is the command header. But the GUPS application framework supports only simple READ and posted write requests. Posted write memory requests perform the same operation as normal write operations, except that no response is returned to the requester.

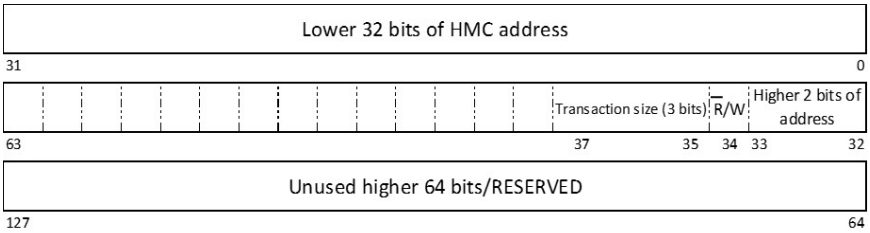


Figure 5.7: The constructs of a command packet header for the HMC

Since the granularity of operation of these basic READ/WRITE commands is 16B, the typical packet size for HMC requests is much larger than what our application dealing with double precision values, needs. For this reason, extra overhead in terms of operations performed on the HMC is incurred. While a simple READ request is unaffected by the high granularity of access, Writes become the bottleneck. A simple write request of size less than 16B needs to converted to a blocking READ operation followed by a modification and then a write request of the updated value to the memory address. For this reason, if we wish to reproduce the same memory state in the HMC as shall be observed in a byte accessible memory system for the application, the overhead is unjustified. For this reason, we adapt a different solution in order to keep our experiments meaningful and fair to the HMC's architecture. We remap all of our memory requests with size less than 16B to memory transactions in the HMC with size of 16B. By doing this, we wish to assure that we are able to study the same stress levels that a true byte accessible memory system shall face in the field. We are able to produce coherent results with this assumption because by the virtue of our proxy application, we can remap every request with size less than 16B, we can uniquely map the request address to a 16B block in the HMC.

CHAPTER 6

EXPERIMENTS AND RESULTS

This chapter presents results from experiments that are used to detail how SUPERLU Schur's Complement and SCATTER/GATHER memory accesses would perform on a GPU, CPU, and FPGA+HMC platform. Our proxy kernel for the Schur's Complement, and DGEMM, is evaluated on multiple platforms and memory access primitives are evaluated using memory traces from Pin on the combined FPGA and HMC hardware. We make inferences regarding the observed behavior and draw conclusions of how application behavior may be used to exploit potential parallelism of the HMC memory system.

As described in section 1.1, we evaluate our implementations on a GPU, FPGA, and a CPU. We chose the NVIDIA Tesla K40c as the GPU baseline. While the K40 has the best performance, this device's high Thermal Design Power (TDP) rating of 245W limits its power/performance efficiency. Similarly, the general purpose Haswell CPU based multicore platform with a TDP rating of 88W provides good performance but at a moderate power costs. FPGA based accelerators offer a lucrative design choice with low TDP ratings of 30W and with the right design for accelerated hardware, they can offer a higher performance per watt than GPU based accelerators.

6.1 Dense SuperLU proxy measurements

We start off with a stripped down implementation of the SUPERLU algorithm, first for the CPU (host) platform, extending it later with GPU (accelerator) offloading using OpenCL. We use the HALO algorithm [1] for absolute baseline comparison numbers for standalone execution time. A detailed analysis with multiple matrix and block dimensions, reveals that the overall execution time spent can be categorized into 2 parts. The time spent in computation on the logic cores and the time spent in accessing the required working data set

from the memory. The memory access time does not scale down linearly as we progress from dense to sparse matrices of large dimensions, owing to the data structure and meta-data overheads as in case of a Sparse matrix.

Here we present our analysis of the performance of our proxy application on different evaluation platforms. In table 6.1, profiling data for one matrix dimension is depicted for experiments on GPU. For matrix dimension of 8000, as block size increases, we observe that the compute overlap time for CPU and GPU (in column 5) also increase. Thus an ideal block size is larger in this case, close to 200.

Table 6.1: Profiling code phase timing contributions on GPU

Matrix Dim	**Block Dim**	**Run time (sec)**	**SC update on host (sec)**	**Time for GPU GEMM to return, while host computes SC (sec)**	**Time on host to apply GEMM on local copy (sec)**
8000	40	17.0681	3.66E-02	2.19E-05	0.00140726
8000	80	10.0502	0.0486543	2.35E-05	0.0024981
8000	100	8.1636	0.0500713	2.41E-05	0.00257875
8000	160	4.7996	0.0538009	2.82E-05	0.00339264
8000	200	4.1368	0.0596777	3.08E-05	0.003977

Table 6.2: Profiling code phase timing contributions on CPU (OpenCL accelerator kernel)

Matrix Dim	**Block Dim**	**Run time (sec)**	**SC update on host (sec)**	**Time for GPU GEMM to return, while host computes SC (sec)**	**Time on host to apply GEMM on local copy (sec)**
8000	40	16.0652	0.06238	0.0388992	0.00133843
8000	80	13.4651	0.0674356	0.08429	0.00179011
8000	100	12.2684	0.0735	0.17232	0.00210493
8000	160	9.9743	0.104352	0.198438	0.00302897
8000	200	13.1479	0.124813	0.160466	0.00359104

In table 6.2, profiling data for matrix dimension 8000 is depicted for experiments running with the OpenCL kernel on the CPU. As block size increases, we observe the compute overlap time for host and accelerator (in column 5) to rise briefly and then fall. Thus for the

CPU platform the ideal batch size is 160 rather than 200. While most timing parameters increase with the increase in block size, we see that the maximum computation overlap time (0.19 seconds) occurs when the block size is 160, which gives us the best overall performance.

Following preliminary tests of our proxy application, we further investigate the effects of block size, matrix dimension, and the HALO optimization versus a baseline implementation (figure 6.1). We sweep a wide range of values for matrix dimension size (N), where N varies from 500 up to 10,000, in steps of 1,000. We sweep the block size (M) from 50 to 200, in steps of 20. A careful analysis confirms our understanding wherein data transfer overheads dominate lower matrix dimension and smaller block size test cases; this situation matches when GPU occupancy is low. As matrix dimensions are increased the GPU becomes better utilized and overall runtime is decreased. The non-HALO block size 80 test case does show up as an outlier in this test. While we don't have detailed profiles for the non-HALO case, it is likely that the dependence tree is at the optimal size when block size is 80 rather than 100. As block size increases, communication overheads increase and the dependence tree becomes denser and larger leading to larger data transfer overhead and possibly to decreased cache locality.

Next, we study the affect of changing block sizes on both the CPU and GPU platforms. As shown in figure 6.2, tweaking block size can have a major effect on the performance. As a generic trend from the figures, larger block sizes deliver higher performance. However, for the OpenCL kernel implementation running on the CPU, block size of 150 appears to be outperforming all other CPU kernels. Even then, the best CPU kernel is easily beaten by the GPU kernels, even at a small block size of 80 for the GPU kernel. Moreover, the sweet spot for optimum performance on the GPU is not the same as the CPU. For an optimum performance on the GPU, a block size of 200 is most efficient while the CPU performs best at block size of 150. This indicates that to balance our CPU and GPU computation we may need to find a "sweet spot" value that satisfies both CPU and GPU.

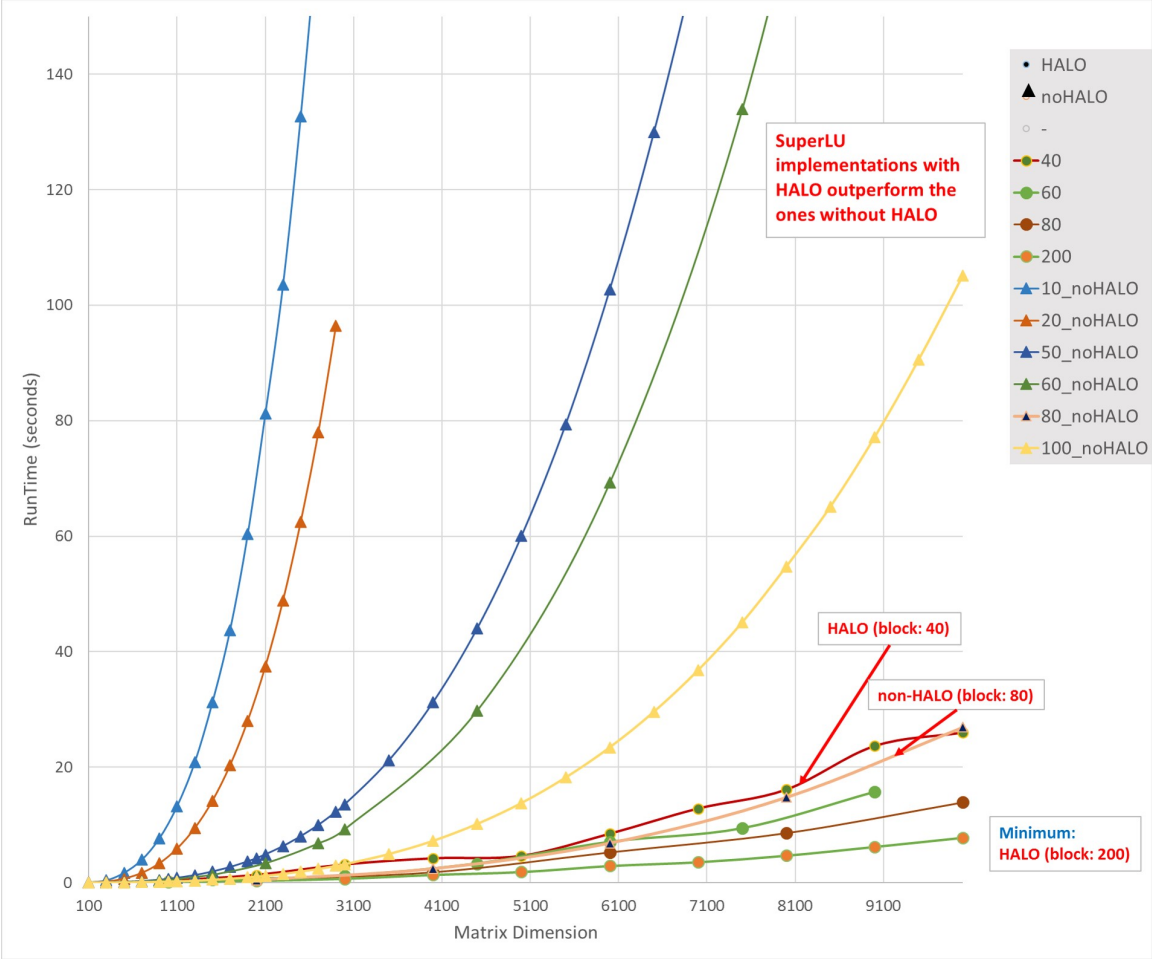


Figure 6.1: Performance comparison of SuperLU kernel with HALO vs. SuperLU without HALO on NVIDIA Tesla K40c GPU.

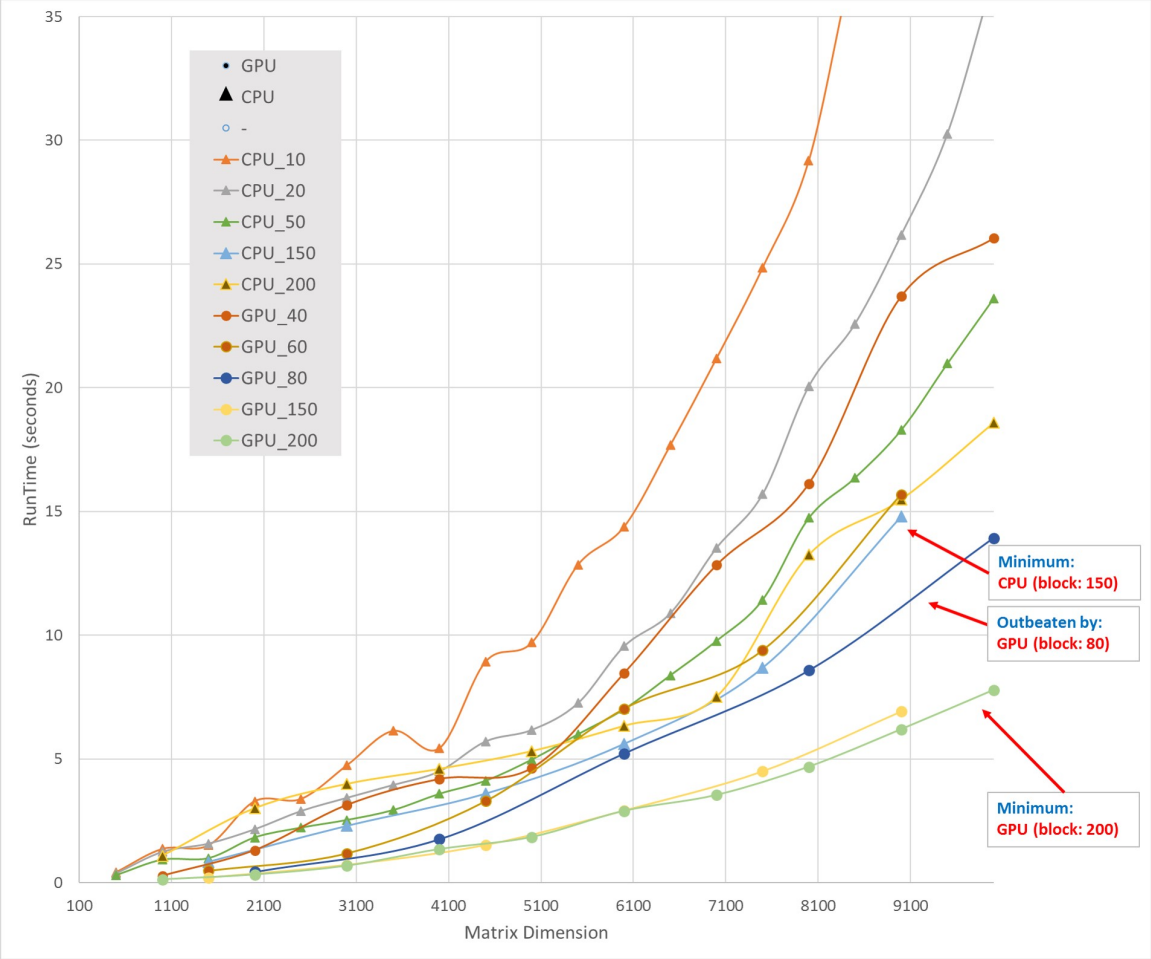


Figure 6.2: Performance comparison of HALO kernel on CPU (Intel 4790K) and GPU (Tesla K40c) platforms.

6.2 Memory trace experiments

We now present results obtained by running memory traces generated via techniques in Ch. 4 on the HMC evaluation platform. We vary multiple design attributes to study the affect of design decisions on the performance of the target application on the 3D-stacked memory system.

We vary attributes such as:

- Number of user ports being employed to issue commands to HMC
- Command batch size for issue

- Serial in-order issue or parallel issue across user ports

By varying the batch size, we are able to understand the optimum queue size for each lane/compute core contending for access to a common link. By reducing the batch size to 1, we can study implications of issuing requests in-order one by one to the memory system. This may potentially help us make predictions about expected performance on interfacing a CPU style processing core to the HMC, which makes blocking memory access calls. The Pico driver limits the batch size to 370, due to the Pico stream framework firmware implementation on the FPGA. For our analysis, we vary the batch size from 4 to 350 (for reliable experiment results).

By issuing commands serially on a single user port, software contention between threads for access to Pico board's object can be minimized. This helps us do an accurate analysis of performance metrics of a user port and how it varies as attributes of issued commands are varied. For example, we can investigate localization of read and write commands on separate ports or localization of memory access requests to a group of vaults on a particular port. Whereas, by issuing commands in parallel across multiple user ports, we are better able to understand link contention implications and the overall performance gain from using 3D stacked memory.

6.2.1 Serial issue of commands

This refers to the state, when only 1 user port (pre-determined as per program) requests access to the HMC at any point of time. Thus software thread contention overheads do not exist.

(a) READ requests only

The graph in figure 6.3 shows that as the batch size increases, the peak port throughput (Giga Operations per Second i.e. GOPS) and peak port bandwidth increases. This is because the software overheads on the host, for preparing and enqueuing a request flit to be issued to the

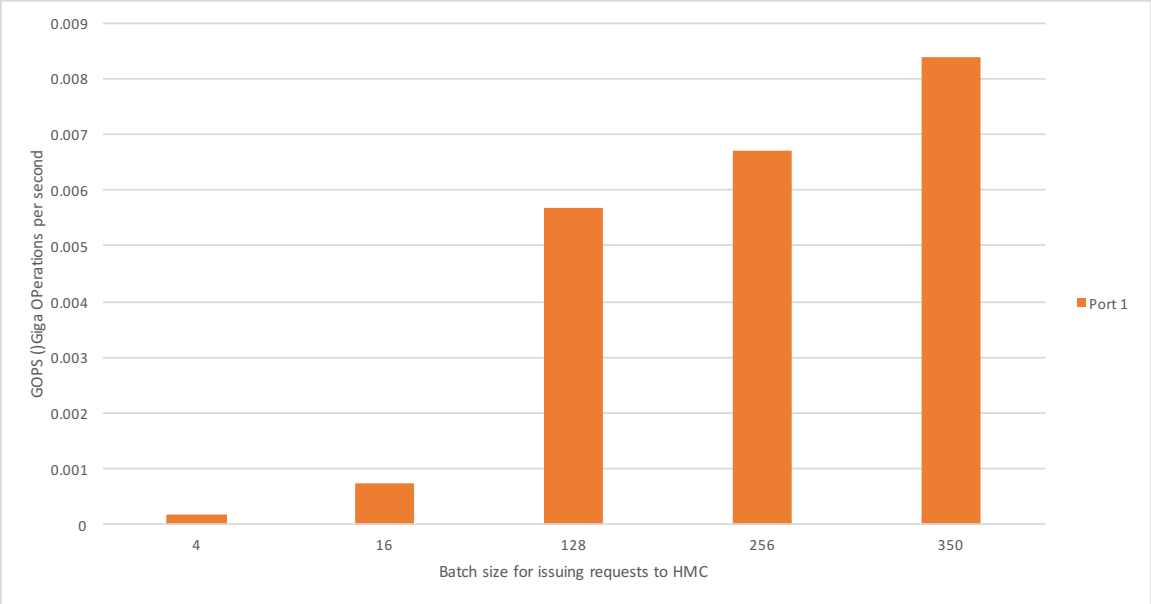
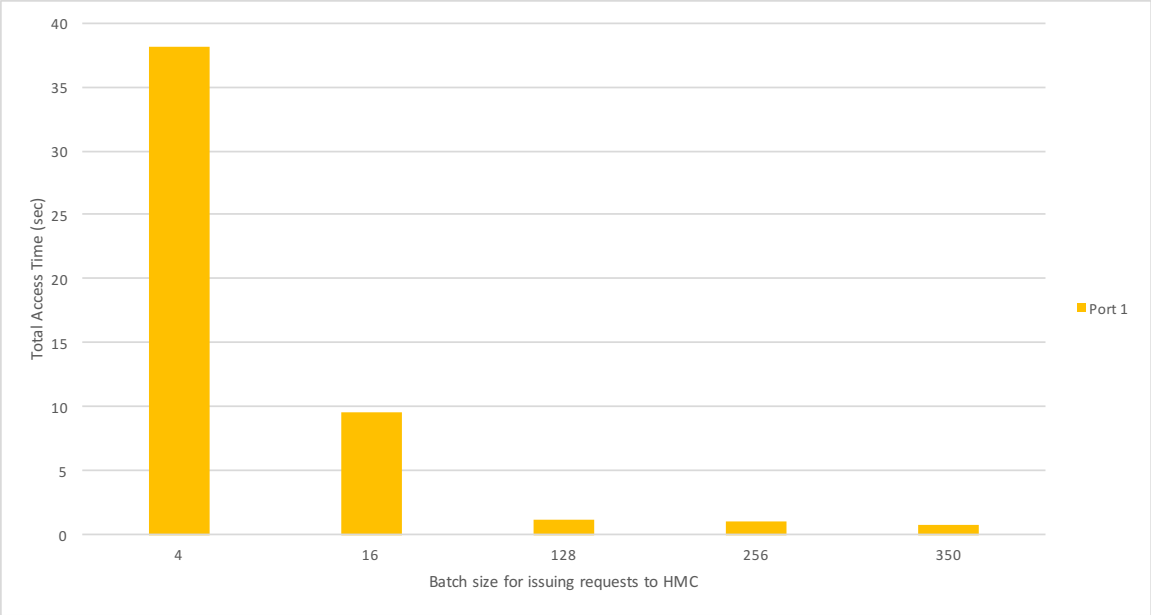


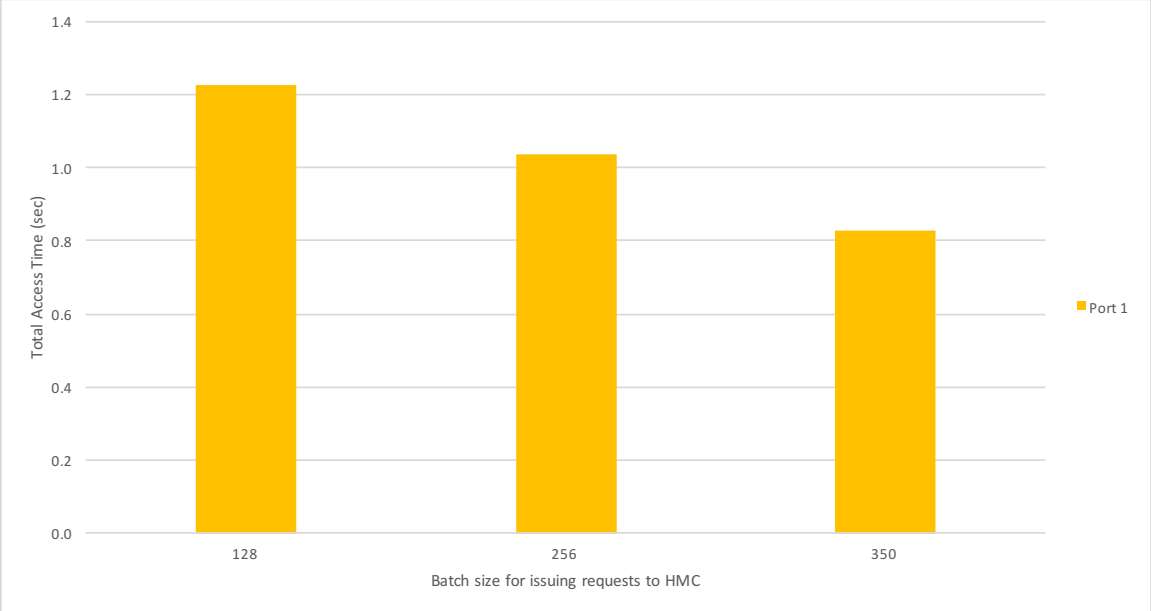
Figure 6.3: Per port GOPS as a function of Batch Size (READ-only)

FPGA, are minimized. Since now these overheads are paid upfront for a batch of command requests. Thus precision for software latency measurement is improved, and precise results are obtained.

In figure 6.4, the cumulative time for servicing the memory trace requests on the HMC decreases as we increase the batch size. This observation also stems from the fact that software overheads in issuing commands to the HMC become smaller as batch size increases. Thus we get much closer to the port's ideal peak read bandwidth. Hence, this reduction in execution time can be linked directly to higher GOPS for larger batch sizes.

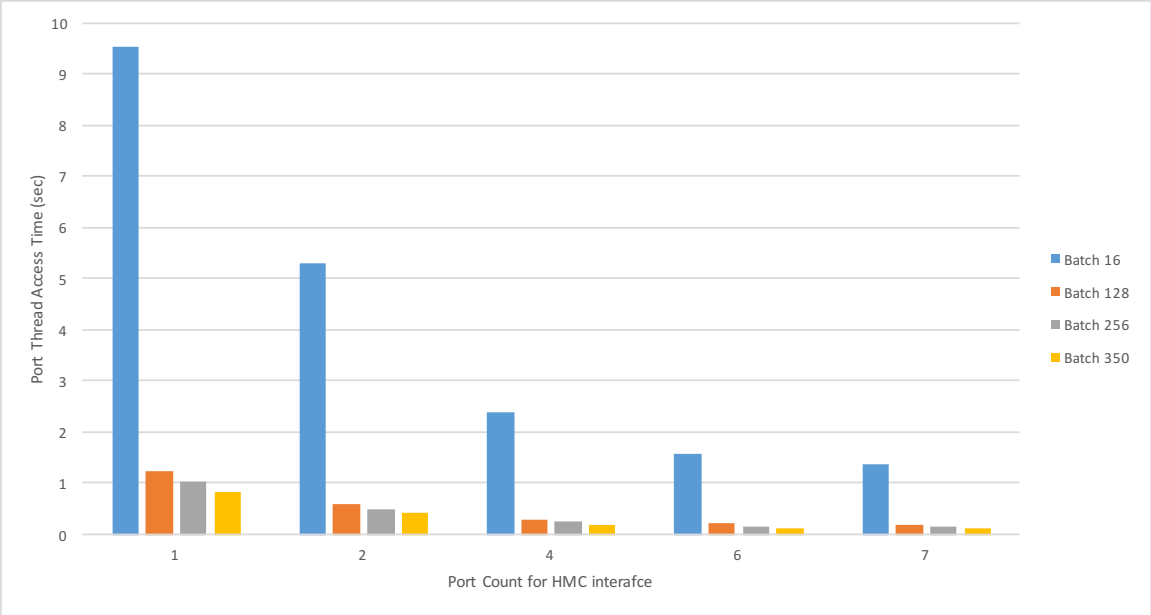


(a) Trace Emulation time vs. Batch Size (READ-only)

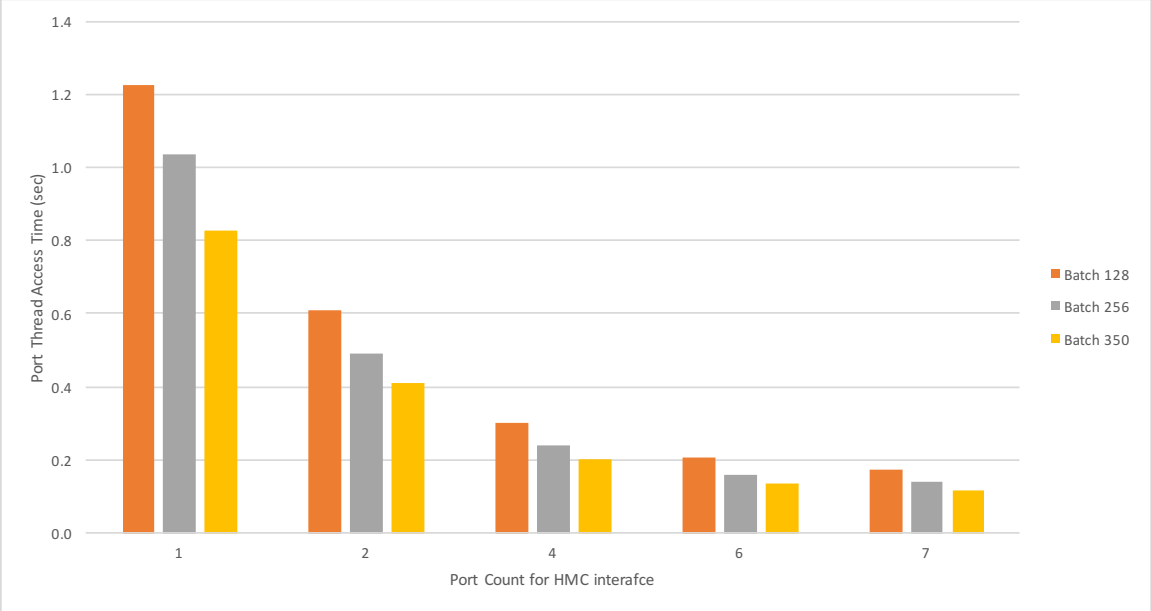


(b) A closer view of result plots for higher batch values

Figure 6.4: Cumulative time to emulate memory trace on the HMC as a function of Batch Size (READ-only)



(a) Predicted cumulative time on HMC vs. number of ports (READ-only)



(b) A closer view of result plots for higher batch values

Figure 6.5: Predicted cumulative time to service memory requests on the HMC as a function of number of ports (READ-only)

We extend the serial issue implementation across multiple ports. This is done in order to make prediction of runtime if the memory requests actually originated from compute cores mapped to the lower most logic layer of the HMC. This layer or the connected FPGA would be the likely location of compute cores for scatter/gather/DGEMM core primitives.

Such a hardware model can be approximated using our framework. We do this by assuming an intelligent data mapping engine [9] allocates blocks in the 3D-stack according to the workload distribution of primitives across the logic layer. Thus each compute core would make memory requests for only the locations housed in the same vault. Thus generated requests by each port are isolated to the local vault controller. This behavior may be replicated by distributing the memory requests from the trace file across multiple ports, but allowing only one port's thread at a time to issue requests to the HMC. Thus each thread simulates a behavior where it has complete control of the HMC link. When one port completes its request allocation, it transfers the control in round-robin fashion to the next port thread. This gives us the figure 6.5 showing prediction for ideal multi-port performance for parallel issue, if the requests were generated from the logic layer of the 3D-stacked HMC. One key point from this figure is that batch size must be high enough to overcome the latency of DMA-style operations since low batch values like 16 provide very poor overall performance.

(b) WRITE requests only

We perform separate experimental analysis for write requests, because of their "posted" nature. Since these requests do not wait for a response to acknowledge their completion, the performance metrics for write requests are expected to be much faster than those for read requests. The only throttling parameter for them is the internal queue size of the HMC vault controllers, which can accommodate only limited number of pending requests at once.

For measuring the latency of servicing posted writes, we employ a different technique. Since the posted writes do not return a response, we sandwich a group $N$of write requests between 2 READ requests, to be issued as one batch. We then use the vendor framework hardware register, `SUM_RD_LATENCY` that counts the cumulative number of cycles taken to service all read requests issued since initialization. The first request of the issued batch, being a read request, triggers the cycle count probe in the FPGA. Since all the requests from

the same port stream buffer are serviced in order by the HMC controller, the response to the last request in the batch marks an end to the probe. The last response to the batch, being a read response, copies the elapsed cycle count into the hardware probe register. We then measure the elapsed cycles count (with a small error margin) taken to issue N write requests, and then we calculate the write request throughput. We keep the value of N large in order to minimize any error due to read request overheads.

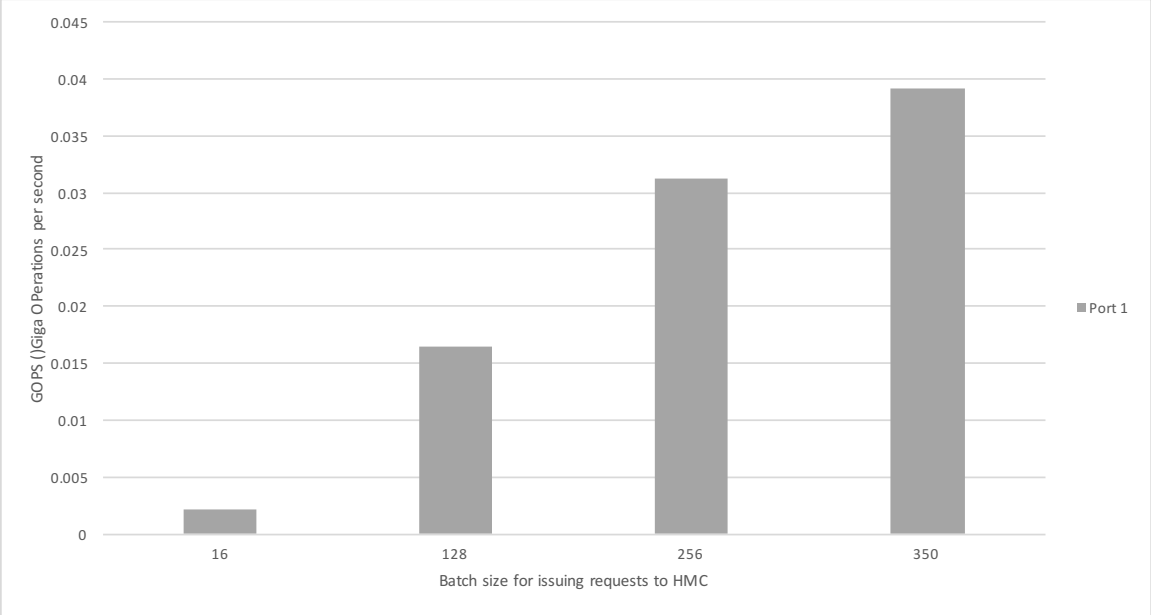
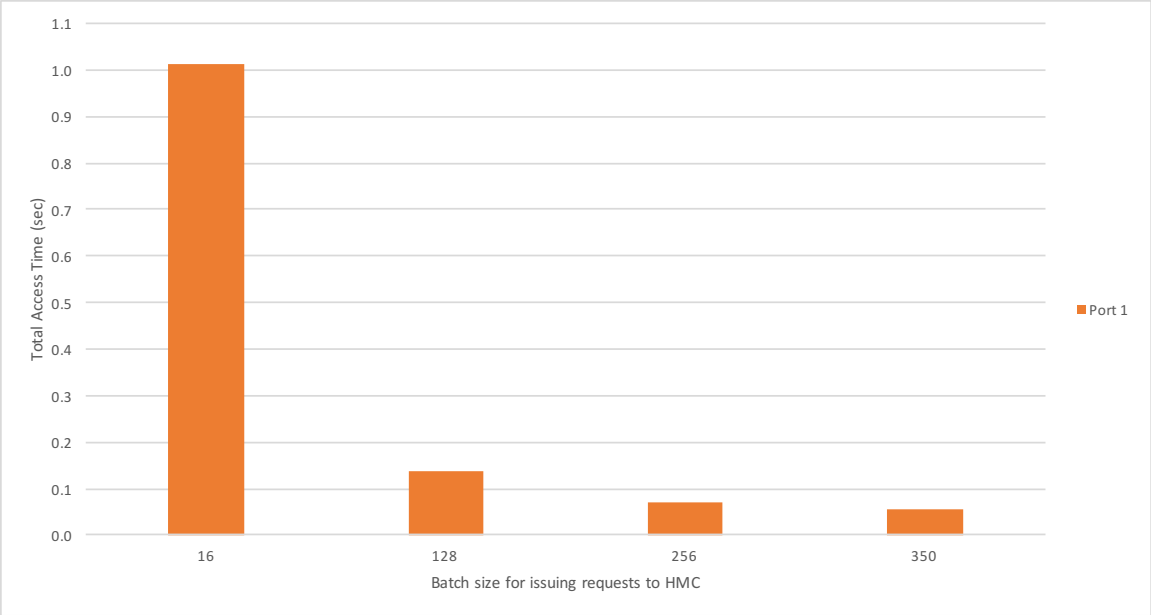


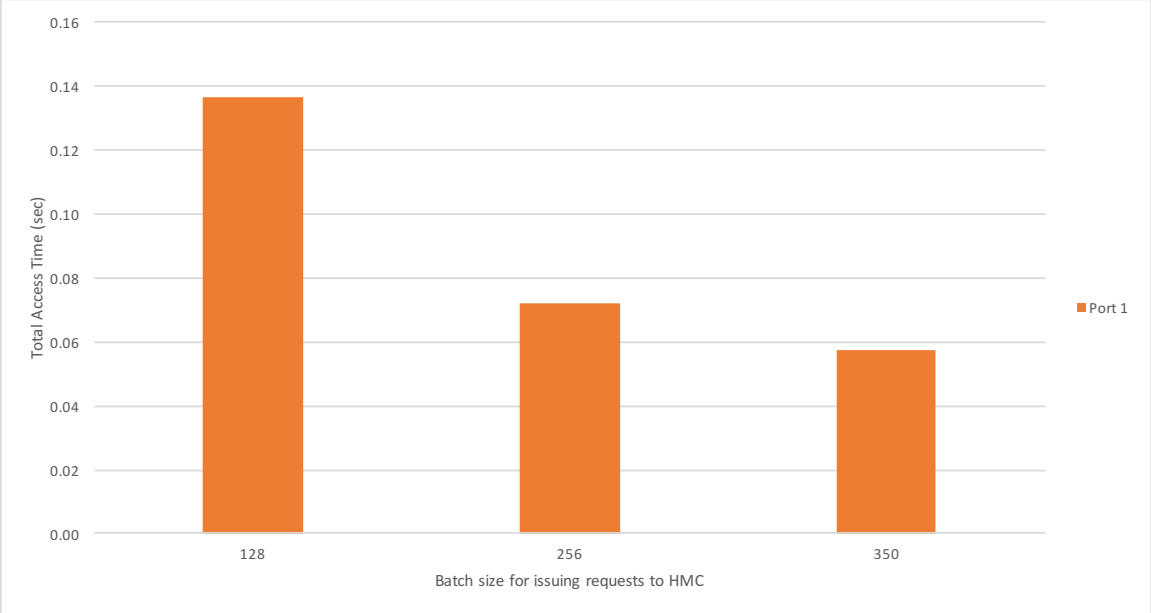
Figure 6.6: Per port GOPS as a function of Batch Size (WRITE-only)

Figure 6.6 shows a trend of rising aggregate link throughput (GOPS) values for higher batch sizes. Note also that due to the non-posted nature of the operations, the max GOPS for the WRITE-only case is almost 0.040 while the READ-only case maxes out at approximately 0.008, a factor of 5 difference.

Plots in figure 6.7 show the expected behavior of falling aggregate runtime for trace on the HMC memory system, with an increase in the batch issue size. This stems from the fact that higher batch size aids in higher aggregate link throughput (GOPS) for write requests and runtime is improved by almost 2x when using a larger batch size (350 vs. 128).

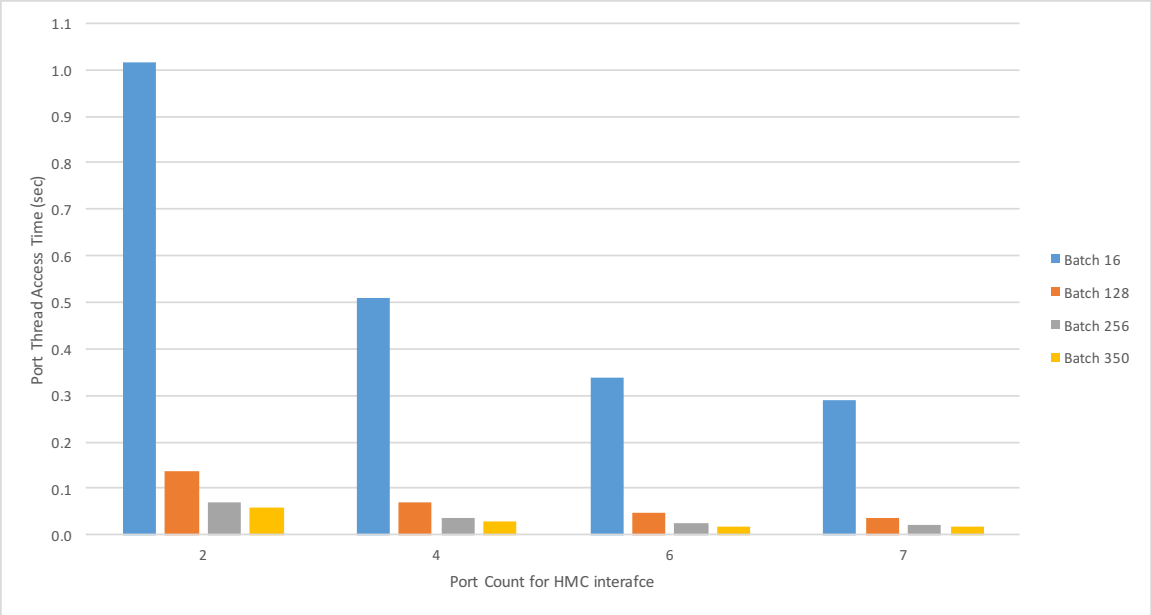


(a) Cumulative time vs. Batch Size (WRITE-only)

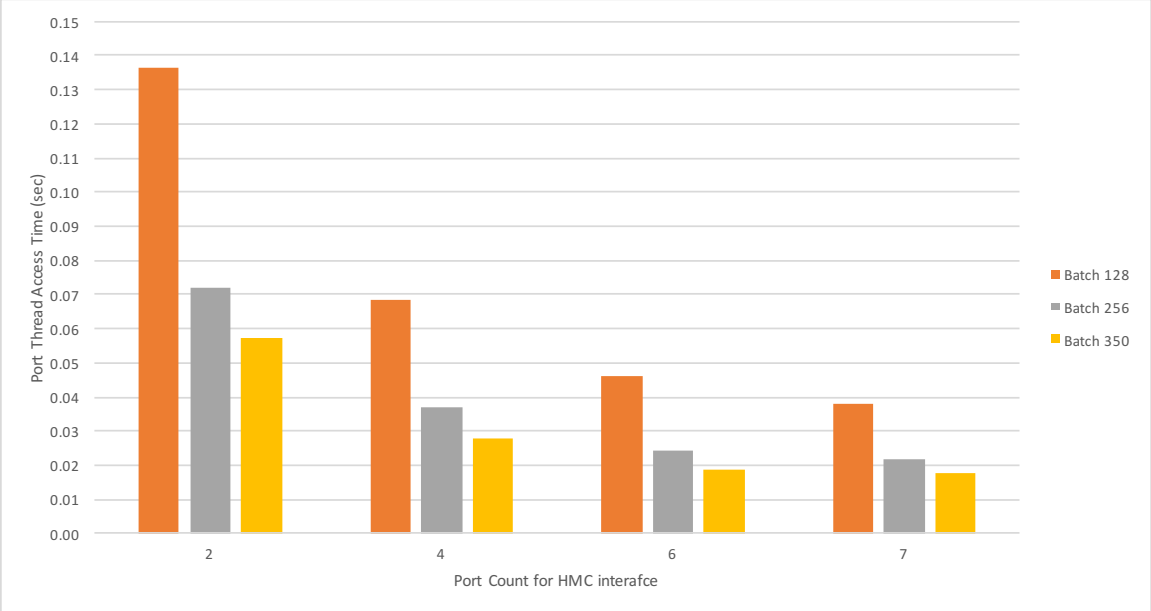


(b) A closer view of result plots for higher batch values

Figure 6.7: Cumulative time to service memory requests on the HMC as a function of Batch Size (WRITE-only)



(a) Predicted cumulative time vs. number of ports (WRITE-only)



(b) A closer view of result plots for higher batch values

Figure 6.8: Predicted cumulative time to service memory requests on the HMC as a function of number of ports (WRITE-only)

Figure 6.8 shows that due to higher aggregate link throughput, the active runtime for each individual port falls as the active port count increases. This decrease in runtime is close to 2x when increasing from 2 ports to 4 (0.58 s to 0.28 s) and slightly lower when going from 4 to 6 ports (0.28 s to 0.19 s)

(c) READ-WRITE requests

When issuing both read and write requests over the same port, the overall port and link bandwidth gets split across the read and write requests. The figure 6.9 shows the distribution of the link bandwidth for such a scenario. With rising batch sizes, the aggregate port throughput increases. Write throughput is lower than the read throughput. This is strictly because of the ratio of read requests to the write requests. We measure the time to service a batch of mixed requests and use this to calculate the read and write throughput ($\frac{NumRequestsInBatch}{TimeToServiceBatch}$). Hence the ratio of the read throughput to the write throughput is $\frac{AvgREADRequestsInBatch}{AvgWRITERequestsInBatch}$. Although the ratio of read command count to write command count varies from batch to batch, an overall ratio of read request count to write request count helps us interpret this better. For our memory trace, the $\frac{READcount}{WRITEcount}$ is 1.5477.

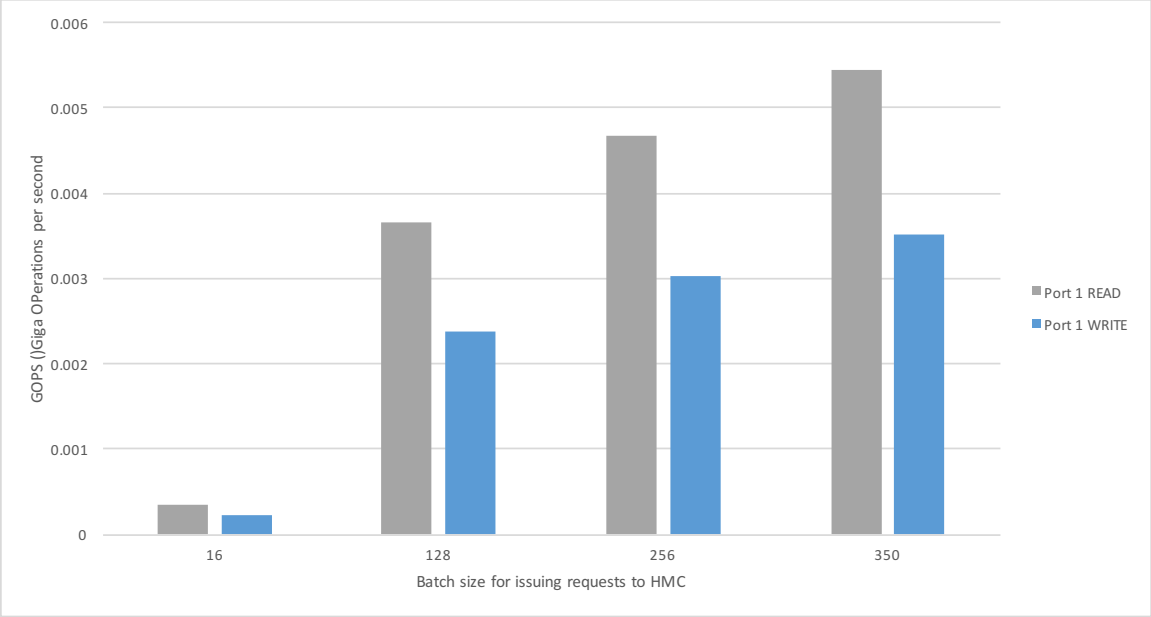
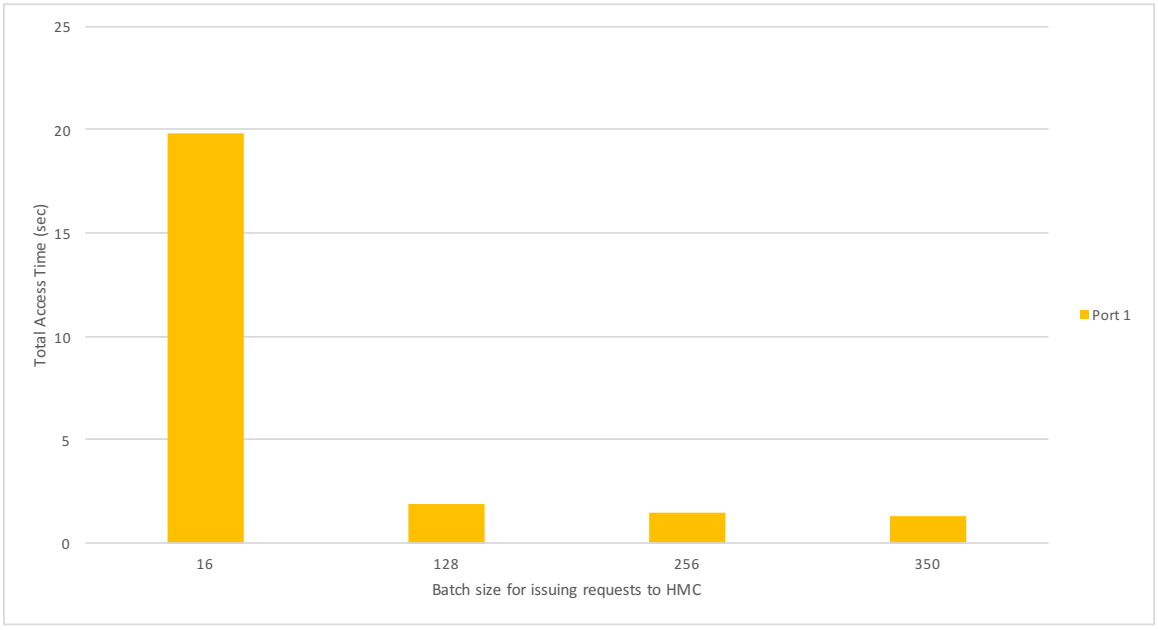


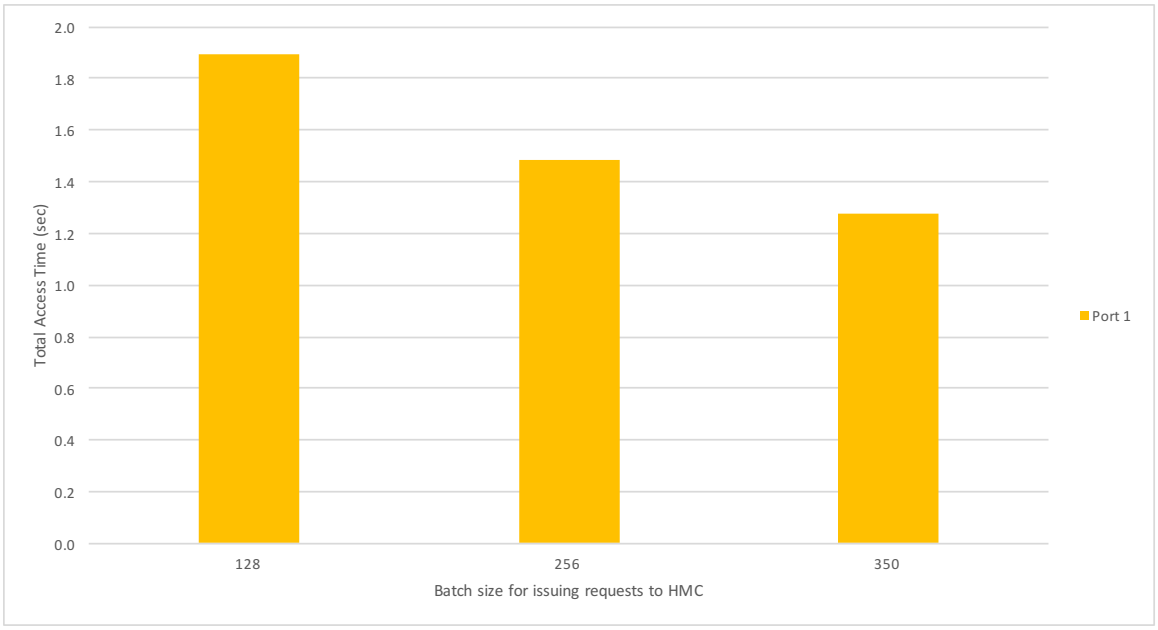
Figure 6.9: Per port GOPS as a function of Batch Size

Figure 6.10 shows the trend for trace run-time on the HMC as the batch size is varied, when requests are issued in a mixed stream on the active port. From it, we infer that cumulative runtime for the memory trace on the HMC decreases with an increase in the batch size. Comparing the absolute run time with those for write request streams, we infer

that latency of responses for read requests govern the achievable throughput in a mixed stream. This is as expected, since posted write requests do not receive a response.

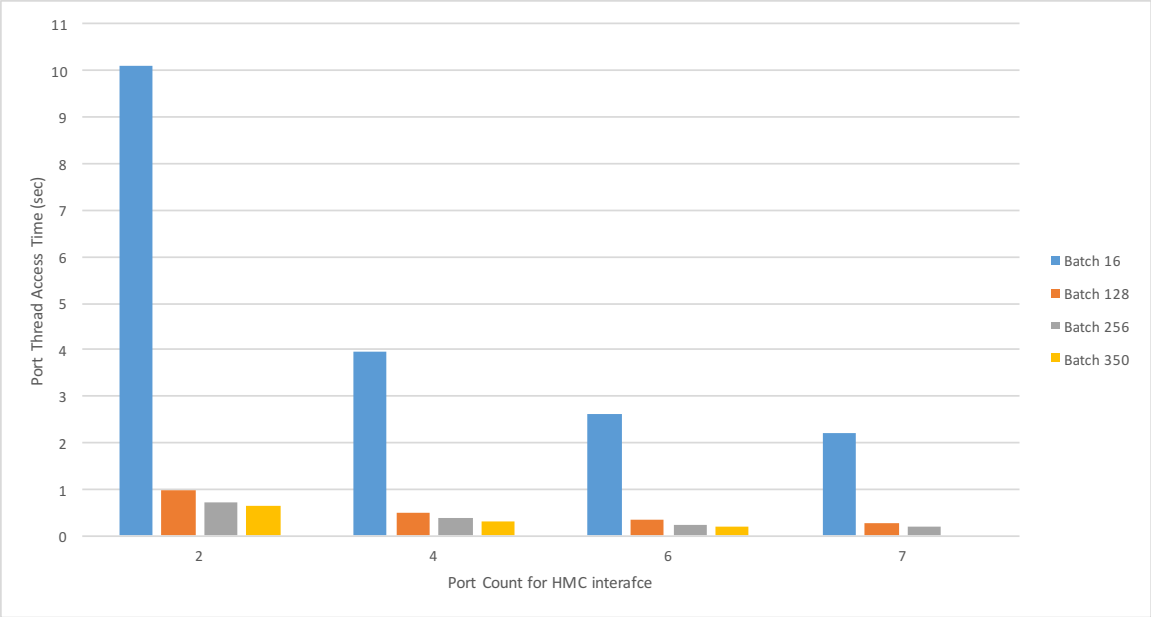


(a) Cumulative time vs. Batch Size

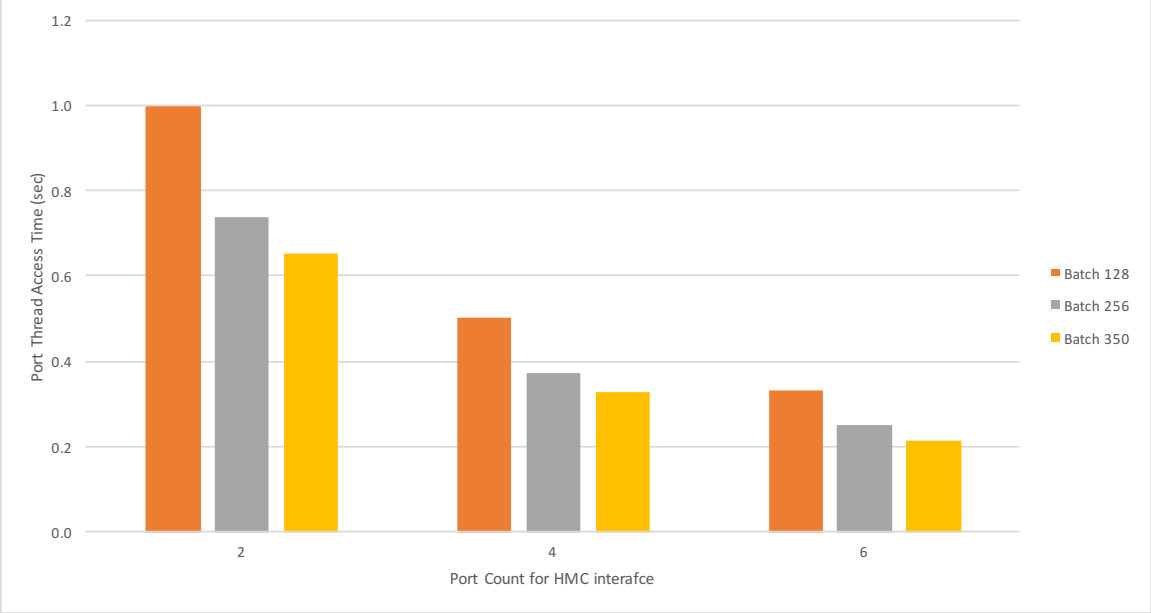


(b) A closer view of result plots for higher batch values

Figure 6.10: Cumulative time to service all memory requests on the HMC as a function of Batch Size



(a) Predicted cumulative time vs. number of ports



(b) A closer view of result plots for higher batch values

Figure 6.11: Predicted cumulative time to service all memory requests on the HMC as a function of number of ports

Plots in figure 6.11 show the predicted ideal performance of the memory system when read, and write requests are issued in a mixed stream, to multiple GUPS ports in parallel. In the ideal scenario, each port would be statically bound to a link to the HMC, and would be free of contention delays. As the port count increases, the burden on each individual port

reduces and thus individual port active run times reduce.

6.2.2 Parallel issue of commands

To implement parallel issue of commands, multiple host threads issue batches of commands from the OpenCL trace to their respective port queues on the FPGA. This includes reading the respective circular buffers on the host memory, and issuing `WriteStream()` calls to respective user port streams. These hardware Pico streams in the FPGA then contend for access to the HMC link. For these experiments, we vary the HW port count from 1 to 7. Tests with 8 and 9 ports crashed, possibly due to an issue with the provided PICO framework.

(a) READ requests only

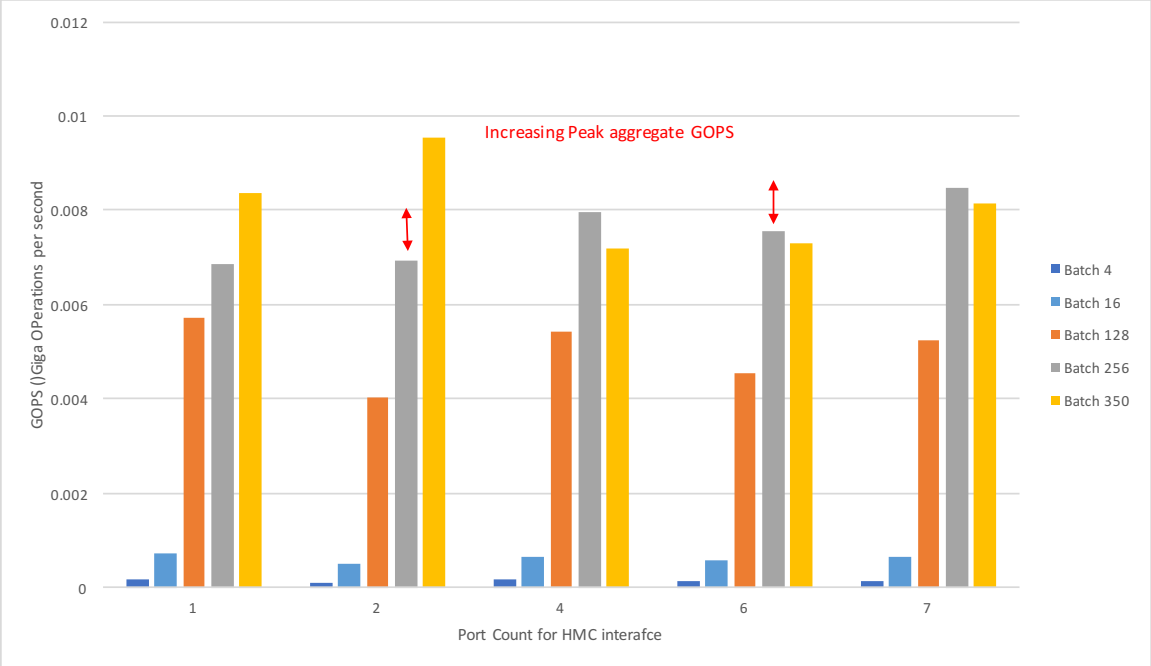


Figure 6.12: Aggregate GOPS as a function of port count and batch size (READ-only)

Figure 6.12 shows measured performance (GOPS) as a function of port count and batch size for multiple threads issuing memory requests to respective user ports in parallel. The

expected behavior of increasing aggregate GOPS is observed only as we increase the port count from one to two. Beyond this, the aggregate READ GOPS saturates. This behavior is because the parallel user ports are being emulated by parallel competing threads running on a CPU host. These threads contend for access to the global Pico board object in order to issue requests to their respective streams and poll for responses. For port counts higher than two, these software thread contention overheads exceed the granularity of hardware latency for servicing issued memory requests. Thus we do not observe increasing performance we might expect from running solely on the FPGA with more ports. This is verified by reading port specific control registers in the FPGA. These registers which store cumulative cycle count spent in order to service global command count, suggest that as port count increases, aggregate read performance (in the hardware) grows as well.

We believe we can observe the true boost in measured GOPS if the memory request generator is implemented solely on the FPGA. One local independent request generation unit for each user port would help measure true read performance metrics, and this implementation would be free of software-related contention overheads.

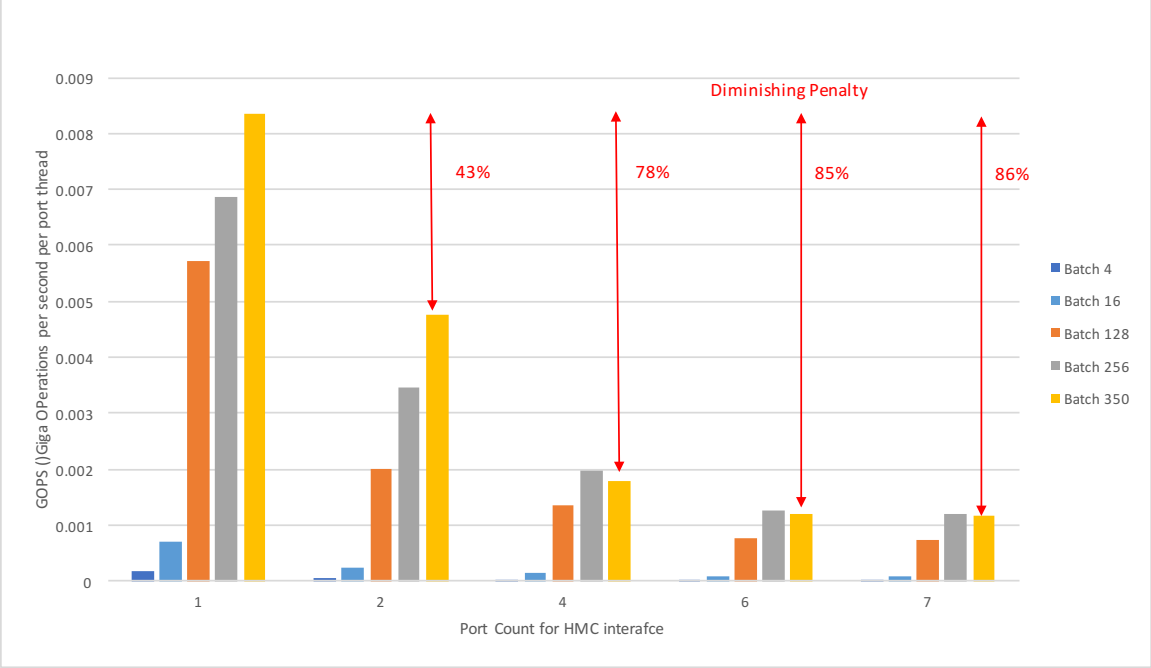


Figure 6.13: Per port GOPS as a function of batch size and port count (READ-only)

Figure 6.13 shows the GOPS throughput for one port when N ports are actively issuing requests to the HMC in parallel. This measurement from the software stack again suffers from inaccuracies due to thread contention overheads. Although the port throughput is expected to decrease as compared to that of single port implementation, the penalty is expected to be somewhat marginal. But since our measurements are done in the software stack by means of timers, the contention overheads play a bigger role. Nevertheless, we are able to see a diminishing penalty effect of overhead in the dropping port GOPS values which leads to increasing aggregate GOPS values.

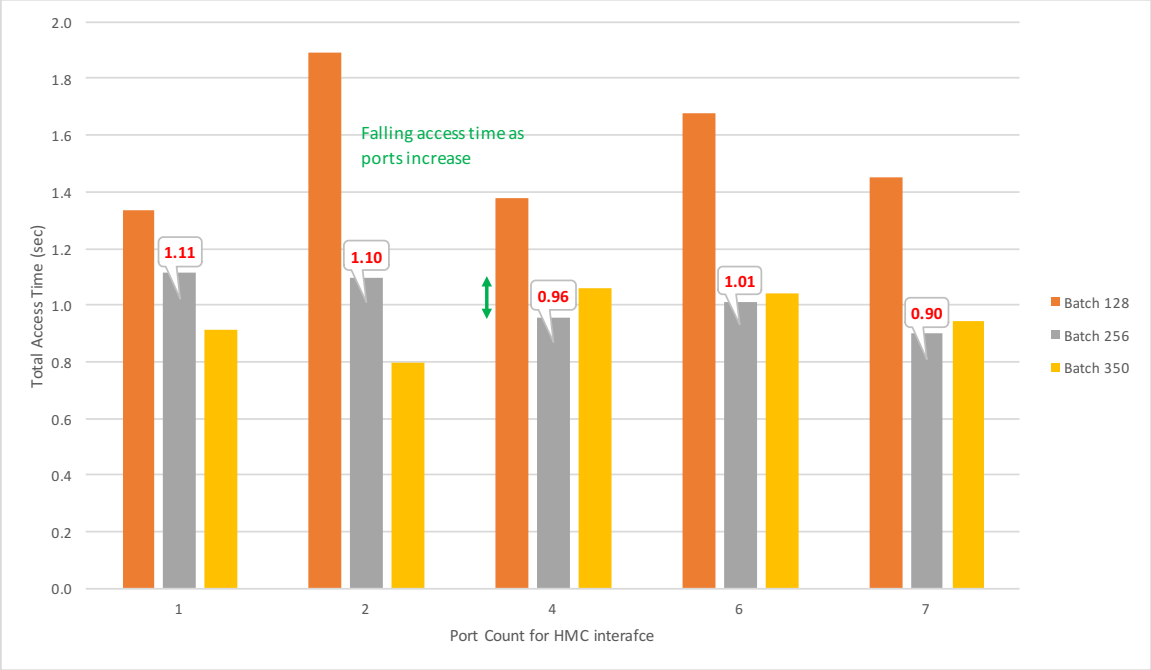


Figure 6.14: Cumulative time to service requests from the trace file on the HMC as a function of port count and batch size (READ-only)

In figure 6.14, we report that the observed performance is optimal for a batch size of 256. This is because for this value, the software issue batch size matches the command queue FIFO depth in hardware. This leads to minimal software overheads to poll for responses of commands still waiting in the queue FIFO on the FPGA. Thus we observe a trend of decreasing overall cumulative time to execute the requests from the memory trace on the HMC, as the port count increases.

(b) WRITE requests only

As explained previously in section 6.2.1, the measurement technique for posted writes uses batches of writes sandwiched by two read requests, which can be measured using existing vendor-provided counters.

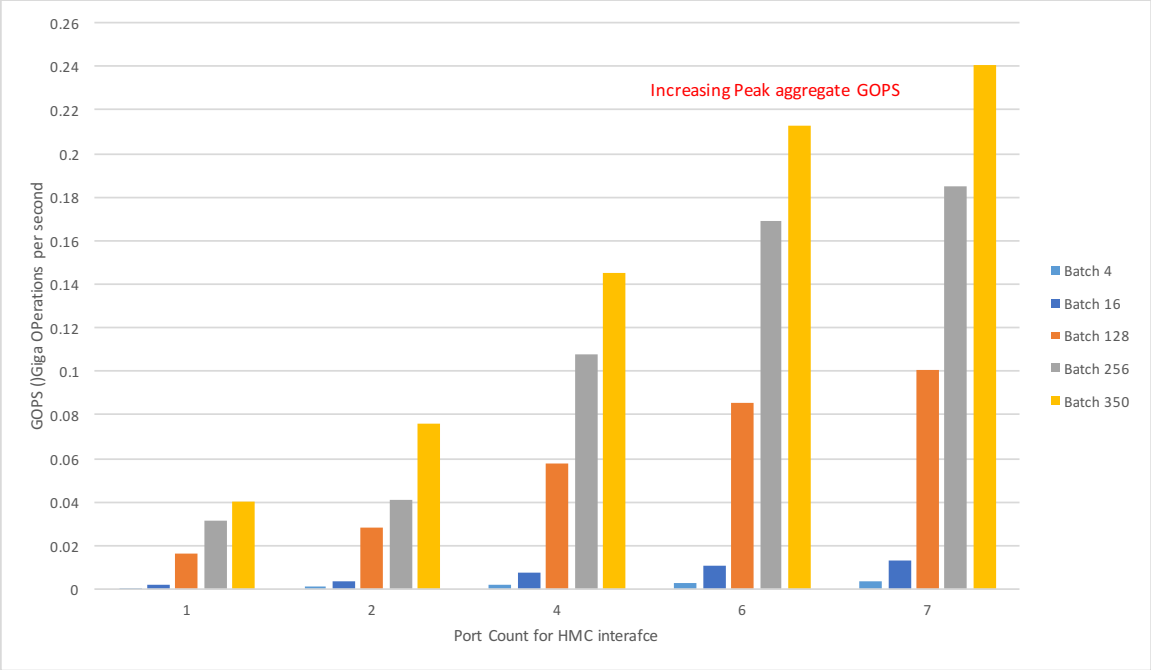


Figure 6.15: Aggregate GOPS as a function of port count and batch size (WRITE-only)

In figure 6.15, we do observe the expected boost in GOPS as the port count increases. This is because of the nature of posted write requests. Since they do not require a response packet, software thread contention overheads for response polling are drastically reduced, and performance can increase to a maximum of 0.24 GOPS with batch size 350.

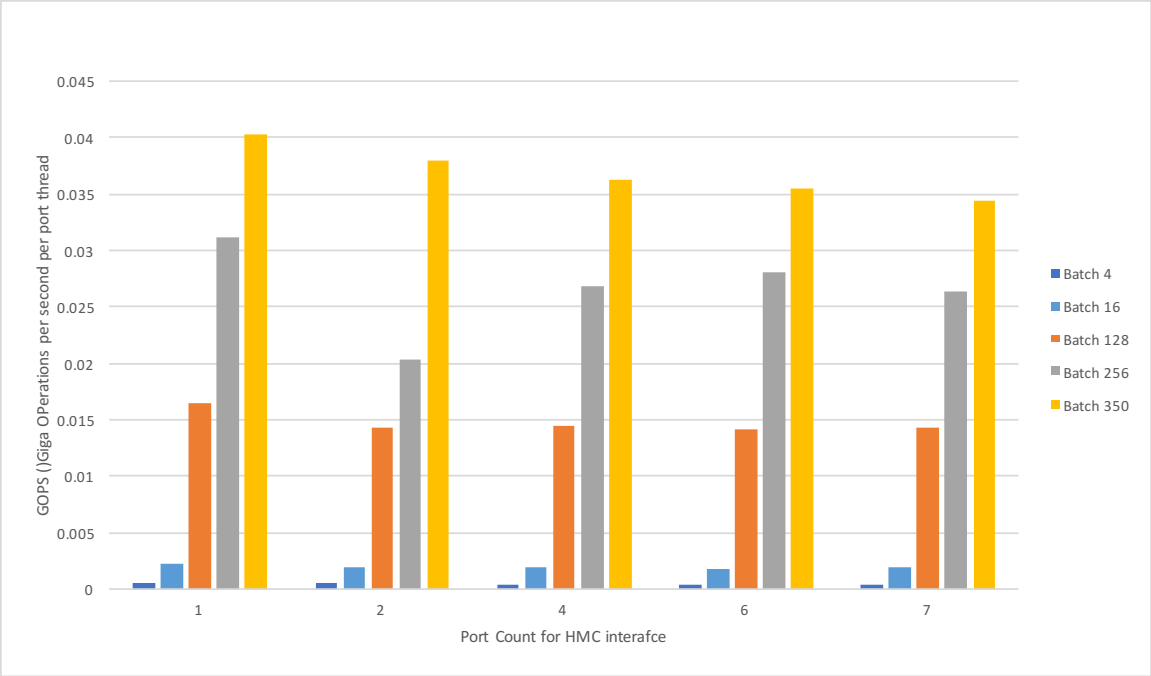


Figure 6.16: Per port GOPS as a function of batch size and port count (WRITE-only)

Figure 6.16 reports per-port GOPS, and like the aggregate test case, the measurements are more precise for write requests due to reduced response polling overhead. Thus we see that the true penalty in port performance is marginal with respect to the observed increase in aggregate GOPS.

CHAPTER 7
CONCLUSIONS

Our analysis of the HPC community's LU decomposition application, focuses on exploiting benefits of 3D-stacked memory architectures. Our analysis was focused on memory accesses that simulate the scatter/gather kernel primitive for the Schur Complement update stage of the application. During our experiments, we observe run time performance benefits and take note of the design decisions and software knobs that can be used to acheive peak performance numbers.

The work contributions of this thesis include a detailed testing of the SUPERLU application suite, including different platforms for different situations that were previously evaluated due to limited support for non-CUDA GPU accelerators. From those analysis, we draw conclusions about critical algorithm and hardware co-design decisions.

The following points must be given high priority while designing an accelerator on the 3D-stacked memory architecture model, particularly HMC:

1. A higher value of batch issue size on the driver software side does not always guarantee peak performance. This goes against common intuition of issuing requests in pending state on the host driver, to be able to hide the memory latency of HMC servicing already issued requests. This may often lead to a user port contending for access to a HMC link to poll for partial set of responses, in hope to issue more requests to keep the hardware request FIFO always filled. This steals useful link access time from ports which could issue a larger batch of requests or read back a larger group of responses. Thus ***the ideal scenario is when the batch size in driver algorithm for issue and polling responses is made equal to the hardware command FIFO depth***. This ensures a higher link utilization by making every transaction between the host and HMC deliver maximum payload.

2. ***The issue batch size is also a function of the number of ports requesting access to the HMC link.*** For lower port counts, a larger batch size may be beneficial, since contention is low. But for higher port counts, the link contention on the host driver to poll for responses can kill the peak achievable throughput. Thus for higher port counts, the ideal batch size lies close to the command FIFO depth of the hardware's vault controllers.

3. ***A deeper exploration of the hardware algorithm co-design to identify the optimal command FIFO depth is needed.*** Our ideal NDP test platform, the AC-510, currently requires further work and vendor support to run a multi-threaded version of SuperLU. This would also be beneficial to achieve a hardware platform wherein the requesting ports would be free of software contention, and precise measurements of cycle counts spent to service different requests could be made.

4. ***The sweet spot for compute sharing as proposed by HALO algorithm is also platform dependent.*** The division of labor between the host and the accelerator for the DGEMM phase prior to the Schur's Complement stage is highly dependent on the hardware platform chosen. Our analysis shows that a block size of 150 on the CPU outperform the performance of all other block size values for the same platform across different matrix sizes. Whereas the ideal block size for a GPU platform was 200, the largest value of the parameter tested in our experiments. While a higher block size may be favorable for a GPU based platform, the deciding factor must be a hybrid of compute capability and data transfer overheads. Thus for a FPGA based accelerator, even lower values of block sizes may deliver a higher performance metric for the application if host and accelerator compute and data movement are properly matched.

CHAPTER 8
FUTURE WORK

The analysis and results described before motivate us for a deeper design space exploration of the 3D-stacked memory architecture. The following steps seem logical extension opportunities for our work.

1. It would be useful to implement the proxy application on the FPGA platform. The Pico Computing framework for translating OpenCL kernels to Verilog hardware implementations would be useful to obtain real prototype performance metrics for the HMC memory system. A programmable memory request generation engine in the hardware, issuing requests of programmed batch size would help get rid of software contention overheads. This would help us better understand the performance implications of multi-port scheduling policies. Currently we are testing a beta driver that uses 1 port, which as our results showed has relatively poor performance.

2. We would like to extend our implementation to support a wider range of commands. This would help us better utilize the HMC specification 1.1 protocol much more efficiently. In particular, atomic commands such as ”BIT WRITE” would be useful to perform Read-Modify-Write operations of granularity lower than the standard 16B. Since majority of double precision operations make 8B memory request transactions, such atomics would be a great addition to the benchmark firmware.

3. Additional timing and dependency information if augmented to the memory trace, would help make better accurate performance measurements. A dynamic binary instrumentation tool for GPU platforms would aid greatly to this effort.

4. A memory trace derived from a GPU simulator or a binary instrumentation tool like GT-Pin would help generate a more accurate memory trace. Such a trace should ideally

be augmented with associated timing information, and batch/warp id information. This would help emulate a parallel issue scenario much closer to the GPU hardware pipeline.

5. Future hardware that uses a shared 3D-stacked memory to service both the host (CPU) and the accelerator (GPU) would greatly reduce data transfer overheads. This has the potential for gaining benefits of data remapping by distributing DGEMM operations over dense data blocks in local vaults.

REFERENCES

[1] P. Sao, X. Liu, R. Vuduc, and X. Li, "A sparse direct solver for distributed memory xeon phi-accelerated systems," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, IEEE, 2015, pp. 71–81.

[2] *SuperLU project website*, `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/`.

[3] D. Abramson, C. Enticott, and I. Altinas, "Nimrod/k: Towards massively parallel dynamic grid workflows," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08, Austin, Texas, 2008, 24:1–24:11, ISBN: 978-1-4244-2835-9.

[4] P. Sao, R. W. Vuduc, and X. S. Li, "A distributed cpu-gpu sparse direct solver.," in *Euro-Par*, 2014, pp. 487–498.

[5] H. Kim, H. Kim, S. Yalamanchili, and A. F. Rodrigues, "Understanding energy aspects of processing-near-memory for hpc workloads," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15, Washington DC, DC, USA, 2015, pp. 276–282, ISBN: 978-1-4503-3604-8.

[6] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 302–325, Sep. 2005.

[7] *Amd's 3d-stacked high bandwidth memory (hbm) rolls out*, `http://fudzilla.com/news/graphics/36995-amd-fiji-hbm-limited-to-4gb-stacked-memory`.

[8] *Hybrid memory cube replacing ddr ram technology*, `https://www.extremetech.com/computing/167368-hybrid-memory-cube-160gbsec-ram-starts-shipping-is-this-the-technology-that-finally-kills-ddr-ram`.

[9] M. Gokhale, S. Lloyd, and C. Hajas, "Near memory data structure rearrangement," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15, Washington DC, DC, USA: ACM, 2015, pp. 283–290, ISBN: 978-1-4503-3604-8.

[10] *Clblas, an implementation of basic linear algebra subprograms, levels 1, 2 and 3 using opencl*, `https://github.com/clMathLibraries/clBLAS`.

[11] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili, "Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, 2012, pp. 58–67.

[12] M. Kambadur, S. Hong, J. Cabral, H. Patil, C. K. Luk, S. Sajid, and M. A. Kim, "Fast computational gpu design with gt-pin," in *2015 IEEE International Symposium on Workload Characterization*, Oct. 2015, pp. 76–86.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: ACM, 2005, pp. 190–200, ISBN: 1-59593-056-6.

[14] *Pin 2.13 user guide and tutorial*, `https://software.intel.com/sites/landingpage/pintool/docs/65163/Pin/html/`.

[15] *Pico-computing software stack and driver framework*, `http://picocomputing.com/products/framework/`.

[16] *Ac-510 hmc accelerator board product brief*, `http://picocomputing.com/ac-510-superprocessor-module/`.