Design Space Exploration for GPU-Based Architecture

Zhenyu Ye
z.ye@student.tue.nl

August 27, 2009

Abstract

Recent advances in Graphics Processing Units (GPUs) provide opportunities to exploit GPUs for non-graphics applications. Scientific computation is inherently parallel, which is a good candidate to utilize the computing power of GPUs. This report investigates QR factorization, which is an important building block of scientific computation. We analyze different mapping mtheods of QR factorization on GPUs. The bottlenecks of them are identified using an analytical model. The limitations of GPUs and its implications on selection of algorithm are discussed. To explore the design space of parallel processing architecture, an equal-area model is proposed. Given an application and an amount of chip area, the equal-area model can provide quantitative information on performance and bottleneck for making design decisions.

Contents

1 Introduction **3**

2 GPU Architecture **4**
2.1 Hardware Architecture 4
2.2 Programming Model 8

3 Analytical Model of GPU **8**
3.1 Kernel Launch Overhead 10
3.2 Pipeline Warp Parallelism 13

4 QR Factorization **13**
4.1 Householder QR Factorization 14
4.2 Givens QR Factorization 16
4.3 Performance Analysis of QR Factorization 18
4.3.1 QR Factorizations on Distributed Memory Systems 19
4.3.2 QR Factorizations on Shared Memory Systems 21
4.4 Experimental Results and Analysis 24

5 Architecture Exploration **25**
5.1 An Elementary Scratchpad Memory Model 26
5.2 An Equal-Area Model 28
5.3 Analysis 30

6 Related Work **32**

7 Conclusion **34**

8 Future Work **35**

9 Acknowledgments **35**

References **35**

Appendices **39**

A Area estimation with CACTI 3.2 **39**

B Examples of QR Factorization **40**
B.1 Householder QR Factorization 40
B.2 Givens QR Factorization 40

1 Introduction

Graphics Processing Units (GPUs) were invented as a special-purposed hardware to accelerate the rendering of 3D graphics. In order to satisfy the demand of complicated shading algorithms, shaders in GPU evolved from dedicated hardware into programmable processors. Modern GPU architectures, such as NVIDIA GeForce[24], ATI Radeon[2], and Intel Larrabee[38], provide massive number of programmable processors for both graphics and non-graphics applications.

Linear algebra algorithms are building blocks of scientific computations. Linear algebra algorithms are inherently parallel, which provides great potential to benefit from parallel processing systems. Early attempts to parallel linear algebra algorithms mainly focus on distributed memory systems, such as clusters of workstations. Emerging shared memory systems, such as multicore architecture, introduced new challenges to algorithm designers. QR factorization is an important linear algebra algorithm for scientific computation. Two methods are widely used to perform QR factorization, one is Householder QR factorization, the other is Givens QR factorization. While Givens QR outperforms Householder QR on distributed memory systems, such as clusters of workstations, Householder QR has significant advantages over Givens QR on shared memory systems, such as multicore CPUs and GPUs. This report investigates the reason of this difference.

The NVIDIA GeForce GPU is an example of an interleaved multithreading architecture, which utilizes massive number of threads to hide memory latency and pipeline latency. It utilizes several levels of memory hierarchy to capture data locality of the applications. The design trade-offs for such architecture is not straightforward. This report proposes an equal-area model to estimate the performance of an application on GeForce-like architecture. It provides information for high-level architecture design decisions.

The contributions of this report are:

1. Extend existing analytical model of GPUs with kernel launch overhead and pipeline warp parallelism.

2. Compare different methods of mapping QR factorization on distributed memory systems and shared memory systems. Use an analytical model to identify the performance bottleneck of QR factorization on GPUs. To the best of our knowledge, this is the first work to analyze the trade-offs for different QR algorithms on different architectures.

3. Propose an equal-area model to perform architecture exploration for parallel processing systems. To the best of our knowledge, this is the first attempt to model the performance of GeForce-like architecture with area constraints.

This report is arranged as follows. In section 2, architecture and programming model of GPUs are introduced. In section 3, extensions to existing analytical model are introduced. In section 4, mapping methods of Givens QR algorithms on GPUs are introduced. Performance of Householder and Givens QR algorithms on distributed memory systems and shared memory systems are analyzed. The bottlenecks of Givens QR algorithm on GPUs are identified by an analytical model. Its implications on algorithm selection for GPU architecture are discussed. In section 6, we compare our work with previous work.

Graphics Pipeline	Larrabee	GeForce 8800
Vertex Shader	CPU	PE
Geometry Shader	CPU	PE
Rasterization	CPU	Raster
Pixel Shader	CPU	PE
Attribute Interpolation	CPU	SFU
Texture Filtering	Tex Filter Logic	Tex Unit
ROP	CPU	ROP Unit
Blending	CPU	ROP Unit

Table 1: Comparison of different approaches implementing graphics pipeline

2 GPU Architecture

Modern Graphics Processing Unit (GPU) contains massive number of programmable processors and a number of dedicated hardware for graphics application. NVIDIA GeForce 8800 GTX is an example of this architecture, as shown in Figure 1. GPUs were designed to render 3D graphics, which is implemented in several pipeline stages, as shown in Figure 2. There are different approaches to map graphics pipelines onto GPUs, depending on the capabilities of the GPU and the trade-offs between flexibility and efficiency. Mapping of graphics pipelines on NVIDIA GeForece and Intel Larrabee are compared in Table 1. Programmable processors on modern GPUs are capable of running non-graphics applications. On the other hand, dedicated hardware on GPUs are difficult to be utilized in non-graphics applications. Therefore, we omit dedicated hardware and focus on the programmable processors in the following discussion. The following discussion is divided into two parts. First, the hardware architecture of GPUs is introduced. Then, the programming model of GPUs is introduced.

2.1 Hardware Architecture

Figure 1 shows the block diagram of NVIDIA GeForce 8800 GTX. It contains eight Texture Processing Clusters (TPCs), each consists of two Stream Multiprocessors (SMs) and one Texture Unit shared by two SMs. Each SM contains 8 Processing Elements (PEs). To fully utilize the massive number of PEs, multithreading is applied. Threads are scheduled to eliminate both horizontal waste, due to control divergence of the SIMD SM, and vertical waste, due to global memory latency, as shown in Figure 3. NVIDIA called the microarchitecture of SM "Single Instruction Multiple Thread (SIMT)". Table 2 shows a classification of David Patterson and John Hennessy[30], where SIMT is classified as dynamic Data Level Parallelism (DLP). In this classification, Interleaved Multithreading SIMD architectures[27][37] are classified as static DLP, because data is processed statically on the horizontal direction and is processed in lock step at runtime. On the other hand, SIMT architecture creates horizontal threads either statically[24] or dynamically[10] to process horizontal data, maintains contexts of each horizontal threads, replaces horizontal lock step with barrier synchronization for both horizontal and vertical threads.

NVIDIA did not reveal detailed implementation of its SIMT architecture. According to descriptions of SMs in GeForce 8800 GTX[24], this paper proposes an implementation of an SIMT architecture, as shown in Figure 4. At runtime, an SM groups 32 threads into one warp, either statically[24] or dynamically[10], which is scheduled to execute on eight PEs.

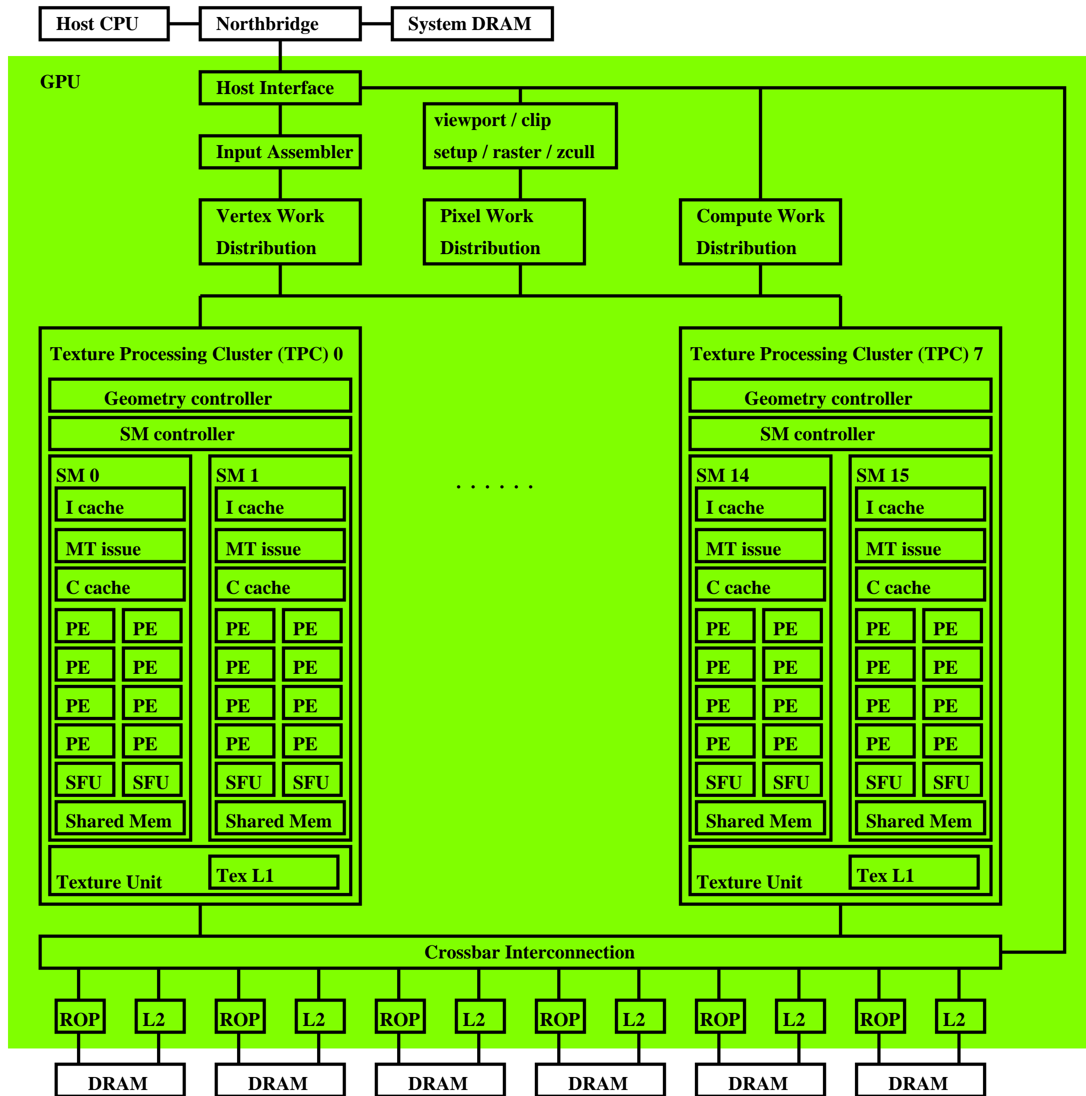


Figure 1: Block Diagram of NVIDIA GeForce 8800 GTX. SM: Stream Multiprocessor. MT issue: Multithread Issue Unit. SFU: Special Function Unit.

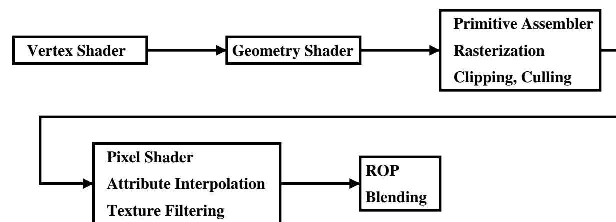


Figure 2: A Typical Graphics pipeline

	Static	Dynamic
ILP	VLIW	Superscalar
DLP	SIMD	SIMT

Table 2: Classification of SIMT by David Patterson and John Hennessy

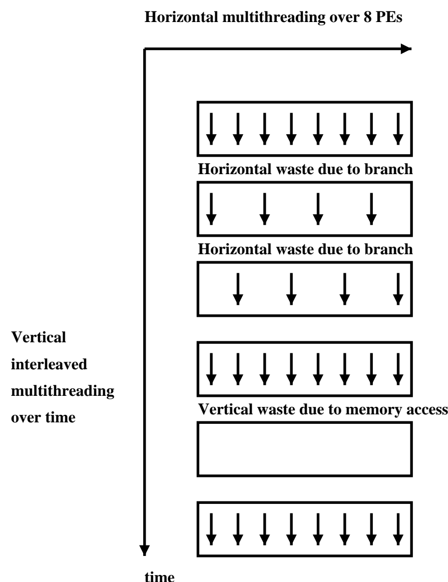


Figure 3: Multithreading on both horizontal direction and vertical direction.

The MT Issue Unit broadcasts the same instruction to all PEs in a similar way as SIMD. Threads in the same warp begin at the same instruction of the program, but are later free to branch to different locations, which is managed by branch stacks. When threads within the same warp diverge at a branch, both paths of the branch are executed, masking out threads of the other path.

Each SM in the GeForce 8800 GTX can maintain contexts of 24 warps. It contains 8192 registers dynamically allocated by threads at runtime. It contains 16 KBytes shared memory partitioned into 16 banks, which are accessed by PEs via a crossbar. Operands of PEs are collected from shared memory and registers. Operands go through several pipeline stages in the operand collector before arriving at PEs. PEs in the GeForce 8800 GTX run at the core clock rate, which is twice the clock rate of other components in SM. One warp is executed as two halfwarps. One halfwarp of 16 threads is executed by 8 PEs in two core clock cycles. At the first core clock cycle, PEs read the operands of the first 8 threads from operand collector and perform operation. At the second core clock cycle, PEs read operands for the next 8 threads from the operand collector and duplicate the same operation in the previous cycle. Results are written back to shared memory or registers every two core clock cycles.

The SM Issue Unit consists of one instruction fetch and decode unit, one instruction window for each warp, one scoreboard for each warp, and one scoreboard processing unit. The scoreboard processing unit prioritizes and selects warps to be fetched from instruction cache and warps to be issued for execution every four core clock cycles. The scheduling unit may use simple scheduling algorithms such as round robin or use sophisticated algorithms such as "post-dominator" described by Wilson Fung[10]. The GeForce 8800 uses a narrow instruction window for each warp, which limits performance if there are not enough warps available to hide the pipeline latency. The GeForce 8800 requires at least 6 warps to hide the pipeline latency of 24 core clock cycles. By increasing the size of the instruction window and scoreboard RAM, the SIMT architecture can utilize potential instruction level parallelism and issue new instructions before the writeback of previous instructions of the same warp.

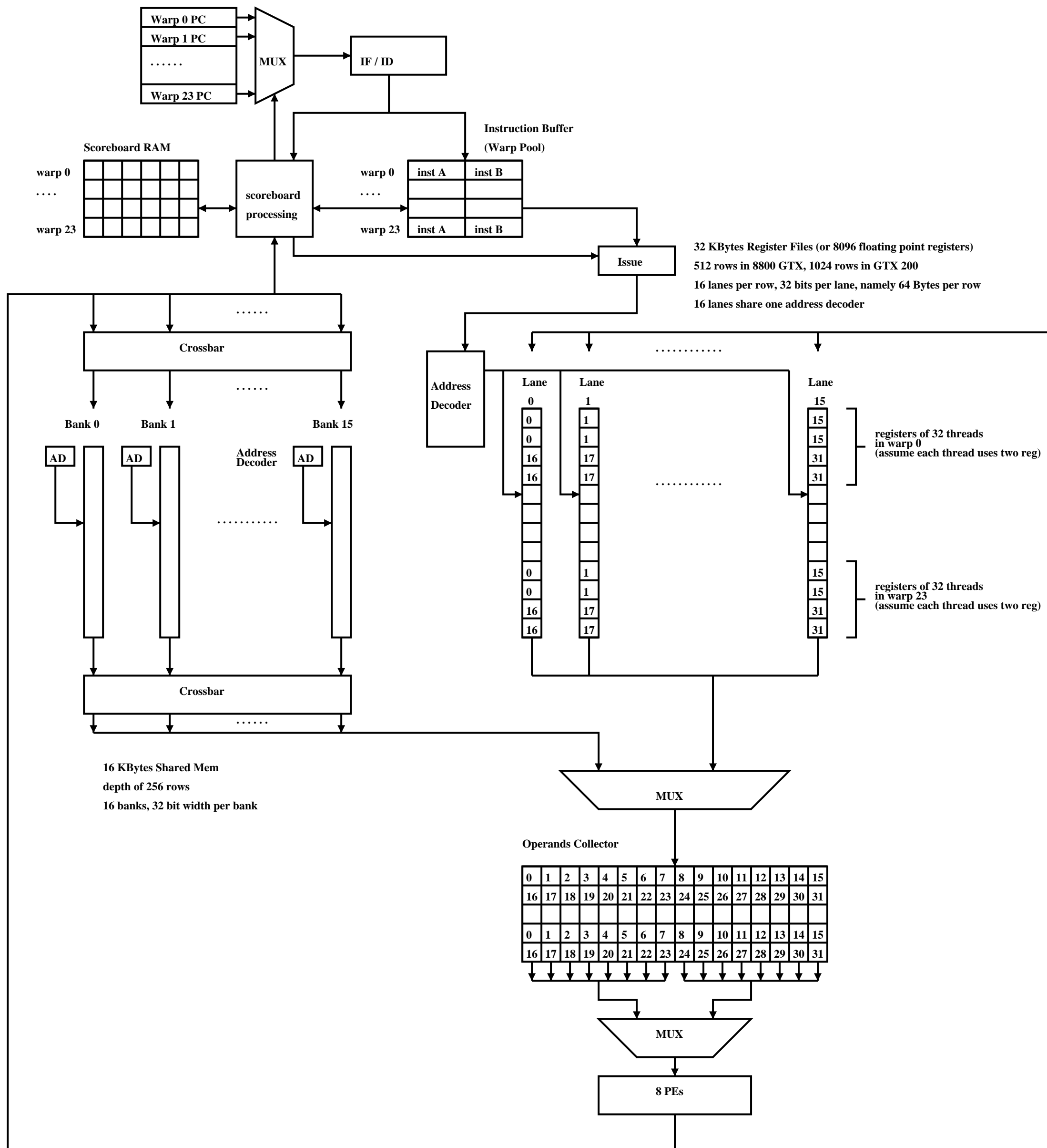


Figure 4: Proposed implementation of the Stream Multiprocessor

However, the trade-offs between performance and area are beyond the scope of this paper.

2.2 Programming Model

Compute Unified Device Architecture (CUDA) is a language provided by NVIDIA to program its GPUs. CUDA extends the C language with directives for an SIMT architecture. CUDA forces programmers to express data level parallelism in terms of thread level parallelism. Unlike traditional SIMD programs, which implicitly specify data dependency via program order and lock-step execution, CUDA programs explicitly specify thread dependency via barrier synchronization. A CUDA program consists of host code and device code. The device code, also called kernels, is offloaded to GPU by the host code. CUDA provides two levels of thread hierarchy, as shown in Figure 5. At the first level, multiple threads are grouped into a thread block, within which threads can synchronize via barriers specified by programmer. At the second level, multiple thread blocks are further grouped into a grid, within which thread blocks are independent to each other. Threads within the same thread block are always assigned to the same SM. Different thread blocks can be assigned either to the same SM or to different SMs. Thread blocks can be executed in arbitrary order, which is scheduled at run time.

MIMD threads are dynamically grouped into warps and scheduled to run on SIMD hardware at runtime. The warp concept, as an implementation method, is transparent to CUDA programmers. One thread block in a CUDA program is implemented as one Cooperative Thread Array (CTA) in hardware. Thread blocks and the CTA have one to one correspondence, and are used interchangeably. Figure 6 shows an example of executing one CUDA kernel a GPU[1].

3 Analytical Model of GPU

Due to the complicated run time dynamism and multiple levels of memory hierarchy, analyzing the bottlenecks of applications mapped to GPU is not straightforward. Roofline model[?] provides methods to analyze the bottlenecks of applications mapped to multicore architectures. It uses computation intensity of the application, memory bandwidth and peak performance of the architecture, as model parameters. Given an application and an architecture, Roofline model tells whether the application is computation bounded or bandwidth bounded. While Roofline model is powerful as a high-level analysis method, it does not model the overlap behavior of computation threads and memory access threads in GPU architectures.

Sunpyo Hong and Hyesoon Kim[15] proposed an analytical model for GPUs. They introduced concepts of Memory Warp Parallelism(MWP) and Computation Warp Parallelism(CWP). MWP measures the number of warps that can simultaneously access global memory. CWP measures the number of warps that can overlap with memory access. The bottlenecks of applications are classified into three cases. Methods to estimate the performance are different in each case.

Case 1: Not enough warps running. MWP and CWP are limited by *active_warps_per_SM*.

Case 2: Memory bound. CWP is larger than MWP.

[1] This example assumes a GPU of 5 SMs and each SM can schedule 24 warps. In 8800 GTX, which has 16 SMs, a kernel of 15 CTAs will be assigned to 15 different SMs, leaving one SM idle.

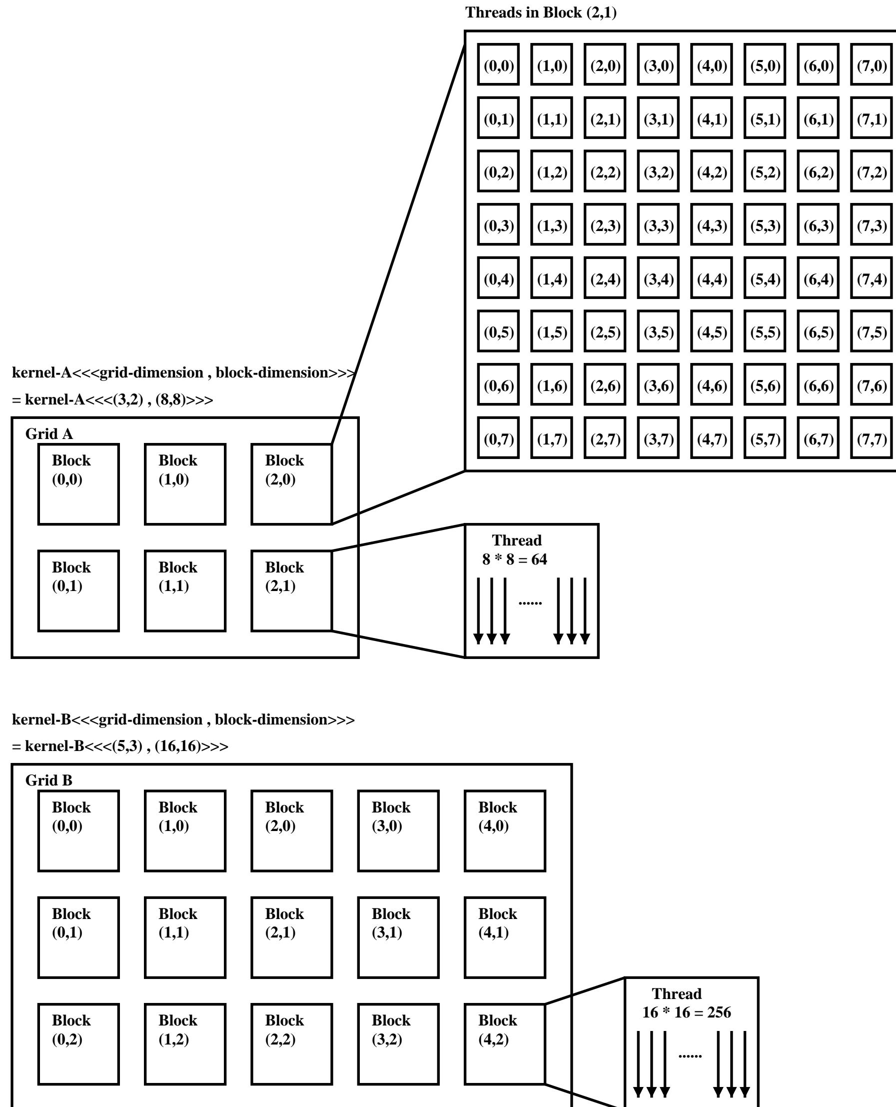


Figure 5: Thread hierarchy in CUDA programming model. Kernel-A has 6 thread blocks. Each thread block consists of 64 threads. Kernel-B has 15 thread blocks. Each thread block consists of 256 threads.

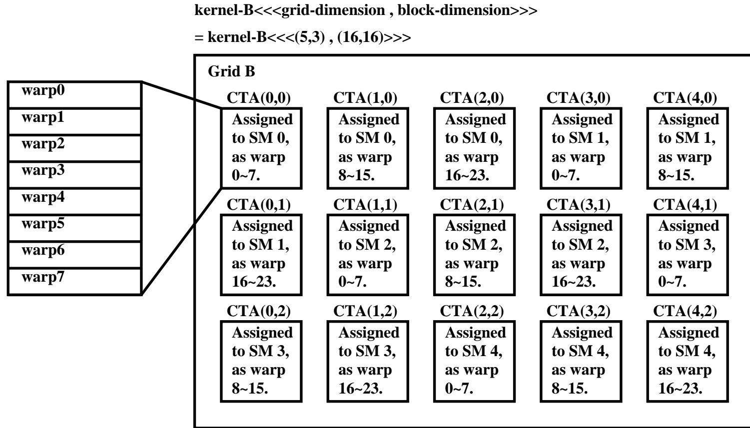


Figure 6: Thread hierarchy in hardware. Example of processing a kernel of 15 CTAs on GPU. Each CTA consists of 256 threads. Eight warps are formed at run time to processes one CTA. At most three CTAs can be scheduled to one SM.

Parameter	Definition	Obtained
Overhead_per_kernel	kernel launch overhead	computed
Args	number of arguments passed from CPU to GPU	Programmer specifies
Overhead_per_block	increment of *Overhead_per_kernel* by increment of one block	Machine Conf.
Overhead_per_arg	increment of *Overhead_per_kernel* by increment of one argument	Machine Conf.
Overhead_base	*Overhead_per_kernel* for kernel with one block and zero argument	Machine Conf.

Table 3: Model Parameters for Kernel Launch Overhead

Case 3: Computation bound. MWP is larger than CWP.

It provides more insights into the bottlenecks of applications on GPU architectures than the Roofline model. Although it provides accurate performance estimations for most applications, experiments show that it may not be sufficient for some cases. This paper extends it with kernel launch overhead and Pipeline Warp Parallelism(PWP).

3.1 Kernel Launch Overhead

Kernel launch overhead for kernels with different number of thread blocks(50-1000), blocks with different number of threads(32, 64, 128), and threads with different number of arguments(0-10) is shown in Figure 7. The detail for small number of blocks(2-50) are shown in Figure 8. We use additional model parameters for kernel launch overhead, as shown in Table 3.

Based on benchmark results, we have the following observations.

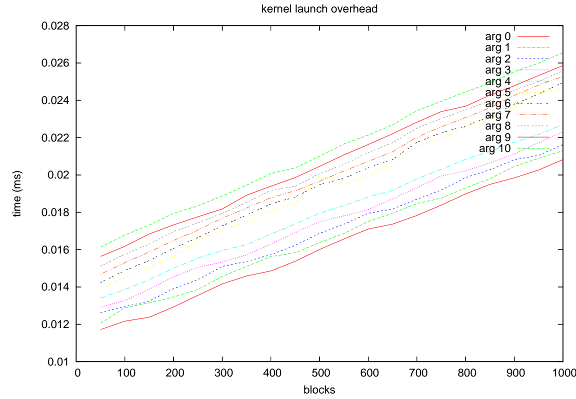
1. *Overhead_per_kernel* has approximately linear relation with *Blocks*. *Args* does has little effect on *Overhead_per_block*.

2. *Args_per_thread* has approximately linear relation with *Overhead_per_kernel*. *Blocks* has little effect on *Overhead_per_arg*.

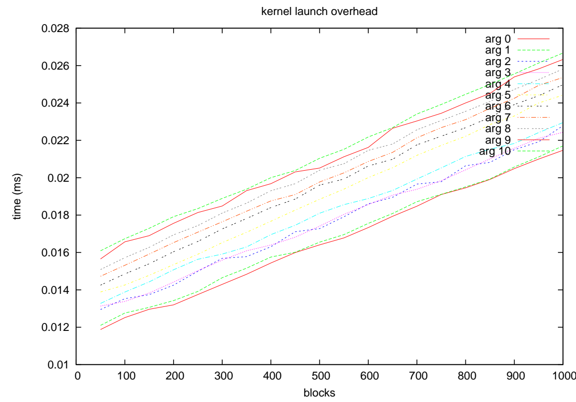
3. *Threads_per_block* has little effect on *Overhead_per_kernel*.

Equation 1 computes *Overhead_per_kernel*.

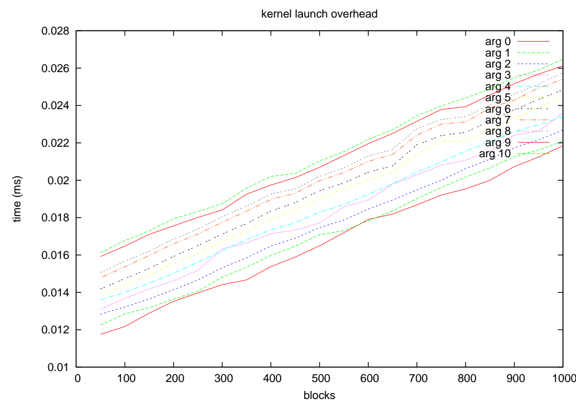
$$Overhead_per_kernel = Overhead_per_block \times Blocks + Overhead_per_arg \times Args + Overhead_base \quad (1)$$



(a) Each block consists of 32 threads

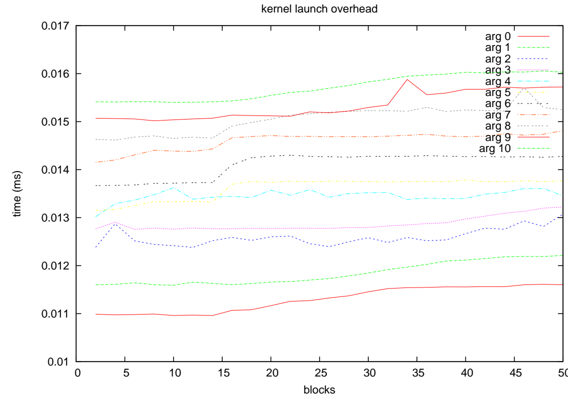


(b) Each block consists of 64 threads

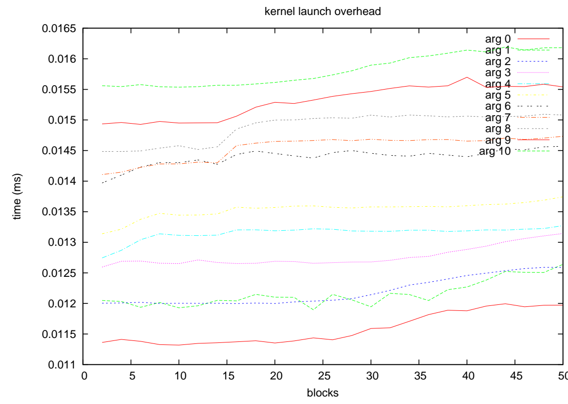


(c) Each block consists of 128 threads

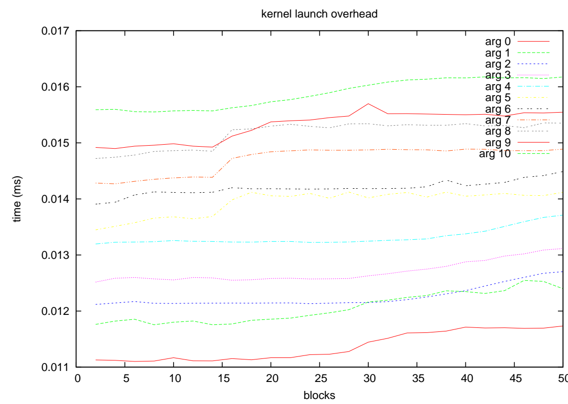
Figure 7: Kernel launch overhead for kernels with different number of thread blocks(50-1000), blocks with different number of threads(32, 64, 128), and threads with different number of arguments(0-10).



(a) Each block consists of 32 threads



(b) Each block consists of 64 threads



(c) Each block consists of 128 threads

Figure 8: Kernel launch overhead for small number of blocks(2-50)

Parameter	Value
Overhead_Base	0.011 ms (17600 cycles)
Overhead_per_arg	0.0005 ms (800 cycles)
Overhead_per_block	0.00001 ms (16 cycles)

Table 4: Model Parameters measured in GeForce 8800 GT (core freq 1.6 GHz)

For GeForce 8800 GT, the model parameters are shown in Table 4.

3.2 Pipeline Warp Parallelism

This report introduces the concept of Pipeline Warp Parallelism (PWP), which represents the maximum number of warps per SM that can be in the execution pipeline of SM. In Sunpyo Hong and Hyesoon Kim's analytical model[15], *Comp_cycles* is used to represent the computation cycles of each warp, as shown in Equation 2.

$$\begin{aligned} Comp_cycles &= Issue_cycles \times \\ &\quad (Comp_insts + Mem_insts) \end{aligned} \tag{2}$$

This parameter *Issue_cycles* comes from machine configuration. The parameters *Comp_insts* and *Mem_insts* come from source code analysis. In GeForce 8800 architecture, pipeline latency of a compute instruction is at least 24 cycles. The next instruction can not be issued until the previous instruction of the same warp writes back the result. With an *Issue_cycles* of 4, *Active_warps_per_SM* should be at least 6 to hide the pipeline latency. We extend the parameter *Comp_cycles* to model the situation where pipeline latency can not be hidden. PWP and *Comp_cycles* are computed by Equations 4 and 5. For applications with fine-grained synchronizations, such as Givens QR factorization, the optimum number of active warps per SM may be less than six, in which case the effect of PWP can be observed, as shown in Figure 9.

$$PWP_full = \frac{Pipeline_latency}{Issue_cycles} \tag{3}$$

$$PWP = MIN\,(PWP_full, Active_warps_per_SM) \tag{4}$$

$$Comp_cycles = Issue_cycles \times \left(\frac{PWP_full}{PWP} \times Comp_insts + Mem_insts \right) \tag{5}$$

4 QR Factorization

Scientific computing is one of the most computation demanding applications. Due to the inherent parallelism of scientific computing, it benefits from parallel processing architectures. QR factorization, as an important building block of scientific computing algorithms, has been mapped onto different parallel processing architectures. Mapping QR factorization on parallel architectures is not trivial. The trade-offs between different QR factorization algorithms on

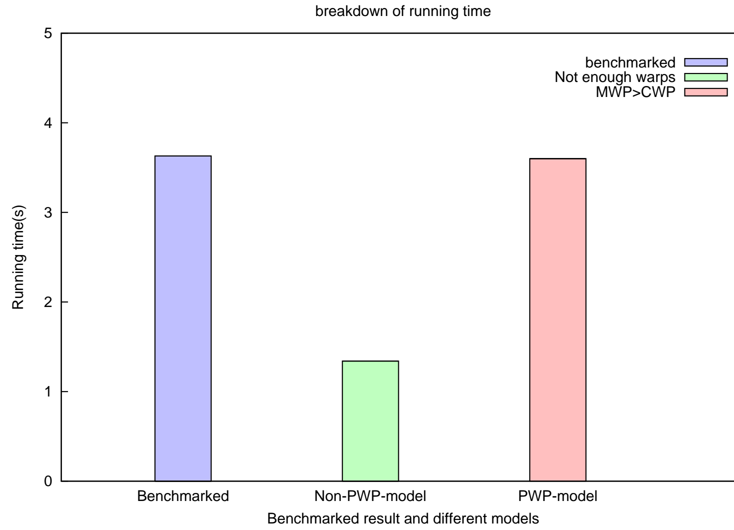


Figure 9: Givens QR factorization with $Active_warps_per_SM$ equal to 2.

different architectures will be discussed in this chapter. We first provide a short introduction to QR factorization algorithms.

This report considers QR factorization for real number. QR factorization for complex number are performed in similar manner. For a real matrix A of m rows and n columns, the QR factorization of A satisfies,

$$A = QR \tag{6}$$

where R is an upper triangular matrix and Q is an orthogonal matrix, namely,

$$Q^T Q = I \tag{7}$$

There are two algorithms to perform QR factorization. One is Householder QR factorization. The other is Givens QR factorization. They will be discussed in the following sections.

4.1 Householder QR Factorization

The Householder QR algorithm annihilates A in a column-by-column manner, as shown in Equation 8.

$$\begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{pmatrix} \tag{8}$$

To annihilate column $A(i:m,i)$, the algorithm generates a Householder reflector v_i, such that,

$$\left(I - 2\frac{v_i v_i^T}{\|v_i\|^2} \right) A(i:m,i) = \alpha e_1 = \alpha(1,\ 0,\ ...,\ 0)^T \tag{9}$$

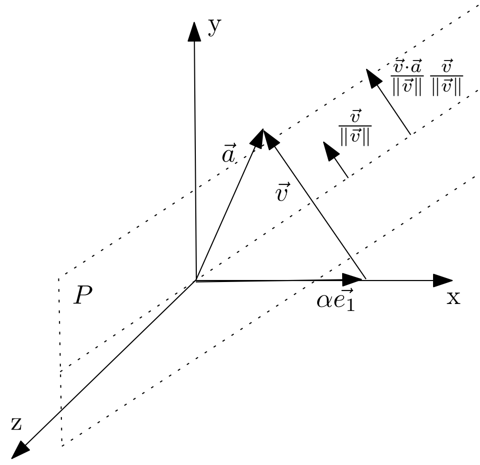


Figure 10: Annihilating vector $\vec{a}$ to $\alpha\vec{e_1}$

α has its sign equal to $A(i,i)$ and its value equal to the norm of vector $A(i:m,i)$, as shown in Equation 10.

$$\alpha = sign(A(i,i)) \cdot \|A(i:m,i)\| \tag{10}$$

The Householder reflector v_i can be obtained by Equation 11.

$$v_i = A(i:m,i) - \alpha e_1 = (\ A(i,i) - \alpha,\ A(i+1,i),\ A(i+2,i),\ ...,\ A(m,i)\ )^T \tag{11}$$

It can be verified that v_i obtained from Equation 11 is correct, as shown in Equation 12.

$$\left(I - 2\frac{v_i v_i^T}{\|v_i\|^2}\right) A(i:m,i) = A(i:m,i) - 2\frac{v_i v_i^T}{\|v_i\|^2} A(i:m,i) = A(i:m,i) - v_i = \alpha e_1 \tag{12}$$

Figure 10 shows an example of a three dimensional vector $\vec{a}$ reflected to $\alpha\vec{e_1}$ over plane P. The unit normal vector of P is $\frac{\vec{v}}{\|\vec{v}\|}$. Projection of $\vec{a}$ on $\frac{\vec{v}}{\|\vec{v}\|}$ has the value $\frac{\vec{v}\cdot\vec{a}}{\|\vec{v}\|}$, which is equal to $\frac{\|\vec{v}\|}{2}$. Therefore, $\vec{v} = 2\frac{\vec{v}\cdot\vec{a}}{\|\vec{v}\|}\frac{\vec{v}}{\|\vec{v}\|}$.

Householder matrix is defined in Equation 13

$$H_i = I - 2\frac{v_i v_i^T}{\|v_i\|^2} \tag{13}$$

If we define $\tau_i = \frac{2}{\|v_i\|^2}$, then the Householder matrix can be written as Equation 14.

$$H_i = I - \tau_i v_i v_i^T \tag{14}$$

One Householder matrix is generated and applied to matrix A on each iteration. After n iterations, matrix A becomes upper triangle, as shown in Equation 15.

$$H_n H_{n-1} ... H_2 H_1 A = R \tag{15}$$

The orthogonal matrix Q can be obtained from Equation 16

$$Q = H_1^{-1} H_2^{-1} ... H_{n-1}^{-1} H_n^{-1} = H_1 H_2 ... H_{n-1} H_n \tag{16}$$

Algorithm 1 is a simple implementation of the Householder QR algorithm. It use two standard routines, SLARFP[2] and SLARF[3], of LAPACK to produce and apply the Householder reflector.

Algorithm 1: HOUSEHOLDER-QR(A)

input : Matrix A with m rows and n columns
output: Matrix R
1 **for** $i = 1$ to n **do**
2   Call routine $SLARFP$ to produce Householder reflector v_i and τ_i for column A(i:m,i).
3   Call routine $SLARF$ to apply the Householder matrix H_i to the trailing matrix.
4 **end**
5 **return** R

Instead of updating one column in each iteration, the compact WY representation[36], also called YT representation, of the Householder QR algorithm updates a panel of multiple columns in each iteration, as shown in Algorithm 2. The Householder matrix H is formed by Y and T matrix, as shown in Equation 17. For panel size of r columns, Y is an $m \times r$ matrix generated by r Householder reflector v, and T is an $r \times r$ matrix generated by standard lapack routine $SLARFT$[4]. Details of the YT representation is available at [36].

$$H = I + YTY^T \tag{17}$$

Algorithm 2: HOUSEHOLDER-QR-PANEL(A)

input : Matrix A with m rows and n columns
output: Matrix R
1 **for** $k = 1$ to (n/r) **do**
2   $s \leftarrow (k-1) \cdot r + 1$
3   **for** $i = s$ to $(s + r - 1)$ **do**
4   Call $SLARFP$ to produce Householder reflector v_i and τ_i for column A(i:m,i).
5   Call $SLARF$ to apply Householder matrix H_i to trailing matrix A(i:m,i:s+r-1).
6   **end**
7   Call $SLARFT$ to generate the T matrix.
8   Apply Householder matrix H to trailing matrix A(s:m,s+r:n).
9 **end**
10 **return** R

4.2 Givens QR Factorization

The Givens QR factorization annihilates matrix A in an element-by-element manner, as shown in Equation 18.

[2]Available at `http://www.netlib.org/lapack/explore-html/slarfp.f.html`
[3]Available at `http://www.netlib.org/lapack/explore-html/slarf.f.html`
[4]Available at `http://www.netlib.org/lapack/explore-html/slarft.f.html`

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ 0 & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} \rightarrow$$
$$\begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{pmatrix} \tag{18}$$

To annihilate an element A(j,k), another non-zero element in the same column k, say A(i,k), needs to be used to generate a rotation angle θ such that,

$$\begin{pmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} A(i,k) \\ A(j,k) \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix} \tag{19}$$

Equation 19 performs a clockwise rotation of vector $(A(i,k), A(j,k))^T$ by angle θ, as shown in Figure 11. In each iteration of the Givens QR algorithm, a Givens matrix $G(i,j,\theta)$ is generated by two elements $A(i,k)$ and $A(j,k)$. In the same iteration, $G(i,j,\theta)$ is applied to two rows $A(i,:)$ and $A(j,:)$, as shown in Equation 20. Algorithm 3 is a simple implementation of the Givens QR algorithm. It uses two standard routines, $SROTG$[5] and $SROT$[6], of LAPACK to generate and apply the Givens matrix.

$$\begin{aligned} G(i,j,\theta)A &= \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & cos\theta & \cdots & sin\theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -sin\theta & \cdots & cos\theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} * & \cdots & * & * & * & \cdots & * \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & A(i,k) & * & \cdots & * \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & A(j,k) & * & \cdots & * \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ * & \cdots & * & * & * & \cdots & * \end{pmatrix} \\ &= \begin{pmatrix} * & \cdots & * & * & * & \cdots & * \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & r & *' & \cdots & *' \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & *' & \cdots & *' \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ * & \cdots & * & * & * & \cdots & * \end{pmatrix} \end{aligned} \tag{20}$$

[5] Available at `http://www.netlib.org/lapack/explore-html/srotg.f.html`
[6] Available at `http://www.netlib.org/lapack/explore-html/srot.f.html`

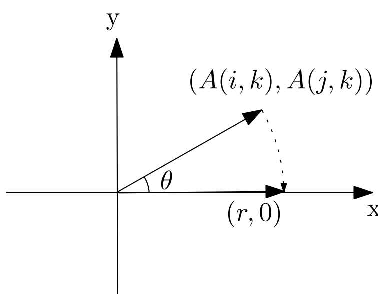


Figure 11: Annihilating element $A(j,k)$ by element $A(i,k)$

Algorithm 3: GIVENS-QR(A)

---

input : Matrix A with m rows and n columns
output: Matrix R
1 **for** $k = 1$ to n **do**
2   **for** $j = m$ to $n+1$ **do**
3     $i \leftarrow j - 1$
4   Call routine *SROTG* to generate $G(i,j,\theta)$ from elements $A(i,k)$ and $A(j,k)$.
5   Call routine *SROT* to apply $G(i,j,\theta)$ to rows $A(i, k+1:n)$ and $A(j, k+1:n)$.
6   **end**
7 **end**
8 **return** R

4.3 Performance Analysis of QR Factorization

Both Householder QR and Givens QR have been mapped and optimized on distributed memory systems decades ago. On clusters of workstations, which is a typical distributed memory system, Givens QR may outperform Householder QR due to lower communication complexity[7].

Both Householder QR and Givens QR have been mapped and optimized on multicore shared memory systems, including GPUs. The Householder QR algorithm has been partially[43] and completely[19] mapped on GPUs. The Givens QR has also been mapped on GPUs[26][18]. The Householder QR outperforms Givens QR by two orders of magnitude on GPUs.

It is observed that the Givens QR is more suitable on distributed memory systems, while the Householder QR is more suitable on shared memory systems. To the best of my knowledge, no previous work has quantitatively explained the reason of the above observation. This section analyzes and explains the performance of two different algorithms on two different architectures.

The analysis assumes the amount of computation is the same for different mapping methods of the same algorithm. The amount of communication is modeled by two parameters, λ, the time needed to transfer a word, and β, the start-up time of one communication. Table 5 summarize the notations used in the analysis. For the convenience of comparison, only the highest order term is kept during the calculation.

Parameters	Description
m	Number of rows of matrix A
n	Number of Columns of matrix A
p	Number of Processors
b	Block-based algorithms use block size of $b \times b$. Panel-based updates use panel of b columns.
λ	Time needed to transfer a word
β	Start-up time of one communication
HDM	Householder QR on Distributed Memory Systems
GDM	Givens QR on Distributed Memory Systems
HSM	Householder QR on Shared Memory Systems
GSM	Givens QR on Shared Memory Systems
HDMC	Column-based *HDM*
GDMR	Row-based *GDM*
HSMB	Use Algorithm 2 to generate panel, then perform block-based matrix multiplication[43].
GSMR	Row-based *GSM* when scratchpad memory can store two rows.
GSMRNS	Row-based *GSM* when scratchpad memory can not store two rows.
GSMC	Column-based *GSM*
GSMP	Panel-based *GSM* when scratchpad memory can store one column.
GSMPNS	Panel-based *GSM* when scratchpad memory can not store one column.
GSMB	Block-based *GSM*
**_comm*	Communication time
**_comp*	Computation time

Table 5: Summary of notations

4.3.1 QR Factorizations on Distributed Memory Systems

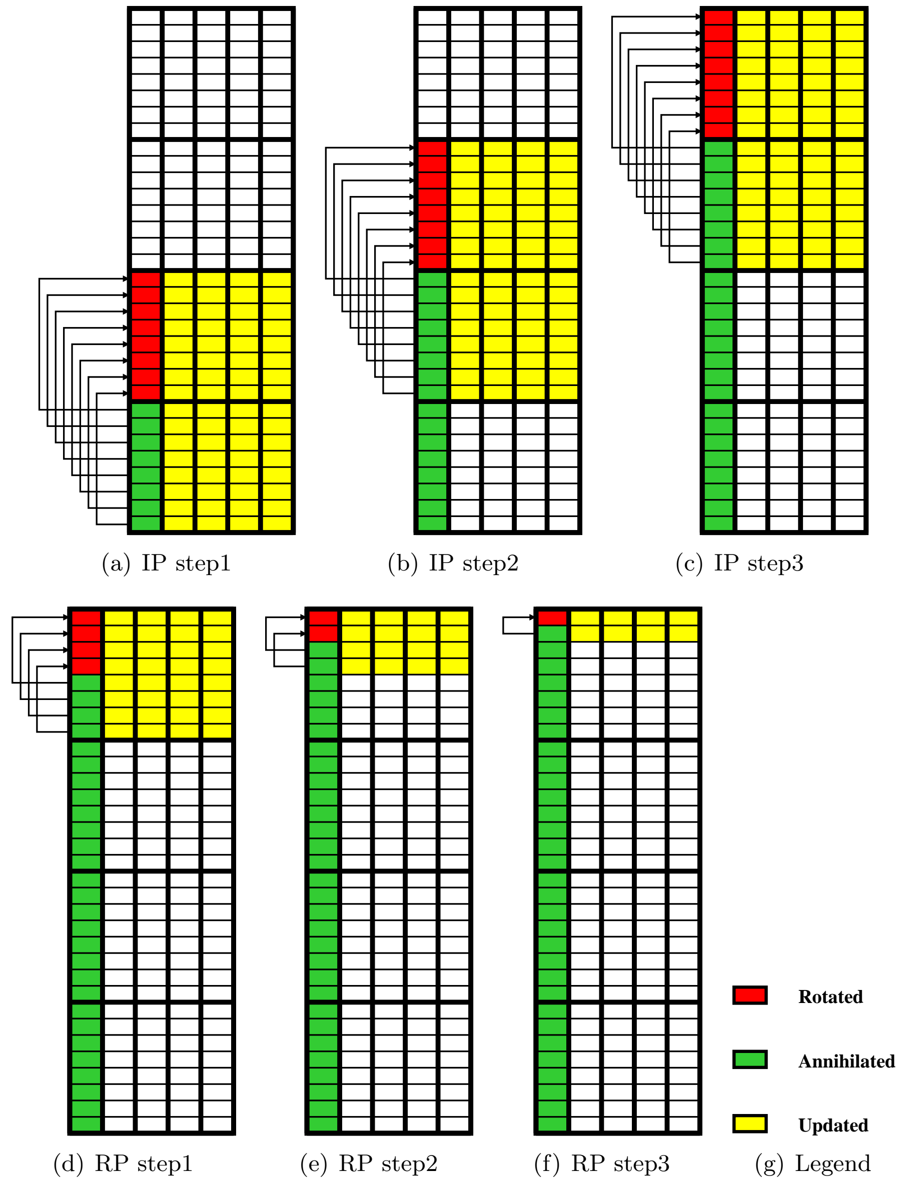
The communication time of column-based HDM(HDMC) is shown in Equation 21. Each processor stores several columns of matrix A in its local memory. In each iteration, one processor generates a Householder reflector, and then broadcasts it to other processors to update the trailing matrix.

$$\begin{aligned} HDMC_comm &\approx \sum_{i=0}^{n-1}(m-i)\cdot\lambda + n\cdot\beta \\ &\approx \frac{(2m-n)n}{2}\cdot\lambda + n\cdot\beta \end{aligned} \tag{21}$$

The communication time of row-based Givens QR algorithm on Distributed Memory System(GDMR) is shown in Equation 22. Row i of matrix A is stored in the local memory of processor (i *modulo* p). In each iteration, one column is eliminated and the trailing matrix is updated. Each iteration has two phases, an internal rotation phase(IP) and a recursive elimination phase(RP)[31]. The IP phase does not incur communication, while the RP phase requires $\log p$ transactions. During each transaction, one row is transferred between two processors. Figure 12 shows an example of GDMR.

$$\begin{aligned} GDMR_comm &\approx \sum_{i=0}^{n-1}(\log p)(n-i)\cdot\lambda + (n\log p)\cdot\beta \\ &\approx (\frac{n^2}{2}\log p)\cdot\lambda + (n\log p)\cdot\beta \end{aligned} \tag{22}$$

In distributed memory systems, β is relatively small compared to λ. Therefore, only the time for data transfers is considered in the comparison. For matrices with $m > n$ and



(a) IP step1 (b) IP step2 (c) IP step3

(d) RP step1 (e) RP step2 (f) RP step3 (g) Legend

Figure 12: GDMR of a 32×5 matrix mapped to 8 processors. Row i is stored in the local memory of processor (i *modulo* 8).

a small number of processors, the communication time of GDMR is smaller than HDMC. This explains the reason Givens QR algorithm may outperform Householder QR on clusters of workstations, as shown by [7]. However, for matrices with $m \approx n$ or large number of processors, the HDMC may outperform GDMR. The trade-offs between HDMC and GDMR over m, n and p is illustrated in Figure 13.

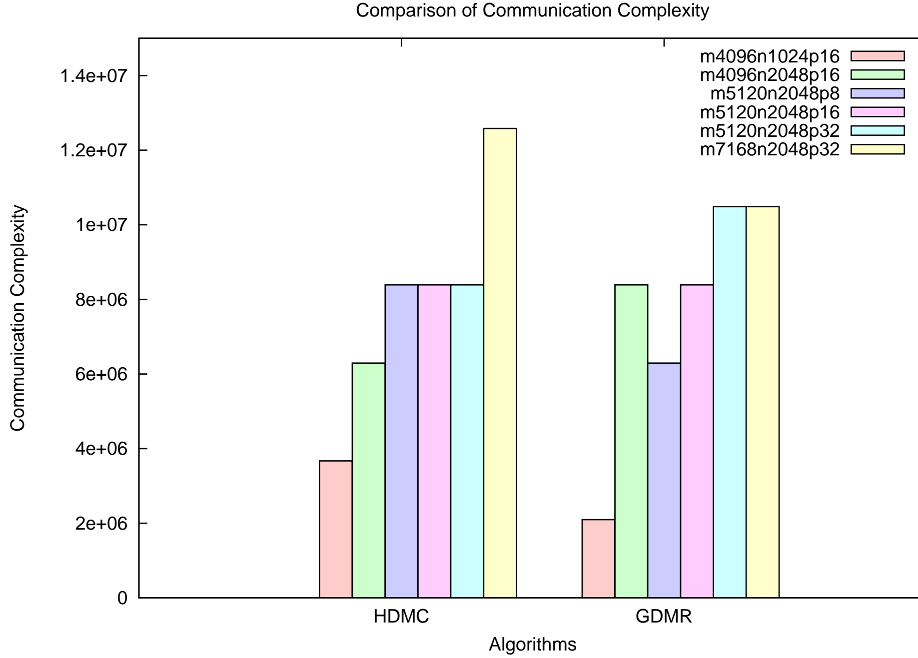


Figure 13: Communication complexity of two QR algorithms on distributed memory systems.

4.3.2 QR Factorizations on Shared Memory Systems

The communication time of block-based HSM is shown in Equation 23. Table 6 summarizes the notations used in the analysis.

Parameters	Description	Mem access in i iteration
TYA_ld	Load panel TY and matrix A to perform multiplication.	$2(m - i \cdot b)(n - i \cdot b)$
TYA_st	Store panel TYA resulting from multiplicaiton of TY and A.	$b(m - i * b)$
YTYA_ld	Load panel Y and TYA to perform multiplication.	$(m - i * b)(n - i \cdot b)$
YTYA_st	Store panel YTYA resulting from multiplicaiton of Y and TYA.	$(m - i * b)(n - i \cdot b)$

Table 6: Summary of notations

$$
\begin{aligned}
& HSMB_comm \\
\approx\; & \sum_{i=0}^{n/b} [(TYA_ld + TYA_st + YTYA_ld + YTYA_st) \cdot \lambda + \beta] \\
\approx\; & \sum_{i=0}^{n/b} [4(m - i \cdot b)(n - i \cdot b) + b \cdot (m - i \cdot b)] \cdot \lambda + \frac{n}{b} \cdot \beta \\
\approx\; & \left(\sum_{i=0}^{n/b} 4(m - i \cdot b)(n - i \cdot b) + \sum_{i=0}^{n/b} b \cdot (m - i \cdot b) \right) \cdot \lambda + \frac{n}{b} \cdot \beta \\
\approx\; & \left(\frac{4(b+n)(b \cdot n + 3m \cdot n - n^2)}{6b} + \frac{(b+n)(2m-n)}{2} \right) \cdot \lambda + \frac{n}{b} \cdot \beta \\
\approx\; & \frac{4(3m-n)n^2}{6b} \cdot \lambda + \frac{n}{b} \cdot \beta \qquad (23)
\end{aligned}
$$

If the scratchpad memory of the cluster can store one row of the matrix, the communication time of a row based GSM algorithm is equal to:

$$
\begin{aligned}
GSMR_comm \;&\approx\; \sum_{i=0}^{n-1} [(2(m-i) \cdot (n-i) + 2p \cdot (n-i)) \cdot \lambda + (\log b) \cdot \beta] \\
&\approx\; \left(\frac{3(3m-n)n^2}{6} + p \cdot n(n+1) \right) \cdot \lambda + (n \log p) \cdot \beta \\
&\approx\; \frac{3(3m-n)n^2}{6} \cdot \lambda + (n \log p) \cdot \beta \qquad (24)
\end{aligned}
$$

If the scratchpad memory of the cluster can not store one row of the matrix, the communication time of a row based GSM algorithm is equal to:

$$
\begin{aligned}
GSMRNS_comm \;&\approx\; \sum_{i=0}^{n-1} [(4(m-i) \cdot (n-i) + 2p \cdot (n-i)) \cdot \lambda + (\log b) \cdot \beta] \\
&\approx\; \left(\frac{4(3m-n)n^2}{6} + p \cdot n(n+1) \right) \cdot \lambda + (n \log p) \cdot \beta \\
&\approx\; \frac{4(3m-n)n^2}{6} \cdot \lambda + (n \log p) \cdot \beta \qquad (25)
\end{aligned}
$$

The communication time of a column based GSM algorithm is equal to:

$$
\begin{aligned}
GSMC_comm \;&\approx\; \sum_{i=0}^{n-1} [(2(m-i)(n-i) + (m-i)(n-i)) \cdot \lambda + \beta] \\
&\approx\; \frac{3(3m-n)n^2}{6} \cdot \lambda + n \cdot \beta \qquad (26)
\end{aligned}
$$

If the local memory of the cluster can store one column of matrix A, then we can update a panel of multiple columns in each iteration. The communication time of GSMP is equal to:

$$\begin{aligned} GSMP_comm &\approx \sum_{i=0}^{n/b} [(2(m-i\cdot b)\cdot(n-i\cdot b)+b\cdot(m-i\cdot b)(n-i\cdot b))\cdot\lambda+\beta] \\ &\approx \frac{(2+b)(b+n)(b\cdot n+3m\cdot n-n^2)}{6b}\cdot\lambda+\frac{n}{b}\cdot\beta \\ &\approx \frac{(\frac{2}{b}+1)(3m-n)n^2}{6}\cdot\lambda+\frac{n}{b}\cdot\beta \end{aligned} \tag{27}$$

If the local memory of the cluster can not store one column of the matrix, then the communication time of GSMP is equal to:

$$\begin{aligned} GSMPNS_comm &\approx \sum_{i=0}^{n/b} [(2b(m-i\cdot b)\cdot(n-i\cdot b)+b\cdot(m-i\cdot b)(n-i\cdot b))\cdot\lambda+\beta] \\ &\approx \frac{(2b+b)(b+n)(b\cdot n+3m\cdot n-n^2)}{6b}\cdot\lambda+\frac{n}{b}\cdot\beta \\ &\approx \frac{3(3m-n)n^2}{6}\cdot\lambda+\frac{n}{b}\cdot\beta \end{aligned} \tag{28}$$

A block based GSM algorithm loads and stores a $b\times b$ block of matrix A into its local memory. Each iteration has two phases, an internal rotation phase(IP) and a recursive elimination phase(RP). The communication time is equal to:

$$\begin{aligned} GSMB_comm &\approx \sum_{i=0}^{n/b} \left[\left((2+\frac{1}{2})(m-i\cdot b)(n-i\cdot b)+(1+\frac{1}{2})(m-i\cdot b)(n-i\cdot b)\right)\cdot\lambda+2\cdot\beta\right] \\ &\approx \frac{4(b+n)(b\cdot n+3m\cdot n-n^2)}{6b}\cdot\lambda+\frac{2n}{b}\cdot\beta \\ &\approx \frac{4(3m-n)n^2}{6b}\cdot\lambda+\frac{2n}{b}\cdot\beta \end{aligned} \tag{29}$$

Figure 14 compares the communication complexity of different algorithms in the above analysis.

According to the analysis above, HSMB and GSMB have the least memory access time. However, GSMB requires fine-grained synchronization to triangulate blocks, which cause threads within a warp to diverge at branch, while HSMB only performs matrix multiplication within blocks, which does not cause divergence at branch. Therefore, GSMB is expected to be slower than HSMB. To verify this conjecture, prototype CUDA programs of GSMB and GSMP are constructed. The prototype programs are optimistic on performance because they always perform coalesced global memory access and ignore the irregular working set size, which adds overheads in a correct program. Experiments show the performance overhead of a correct program over a prototype program is less than 10%. Therefore, the prototype program is good enough for a rough performance estimation. The prototype program shows

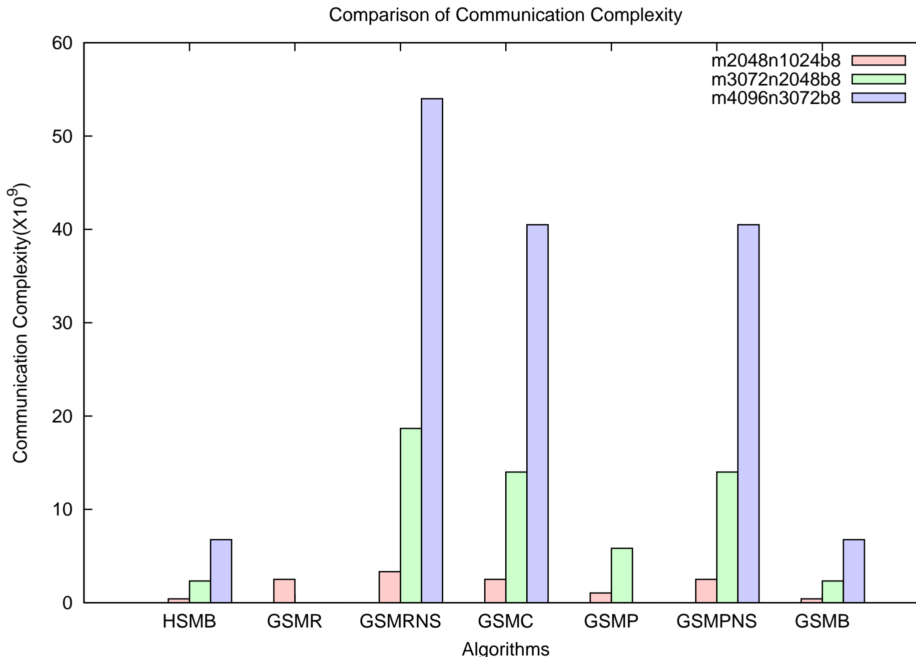


Figure 14: Communication complexity of different QR algorithms on shared memory systems.

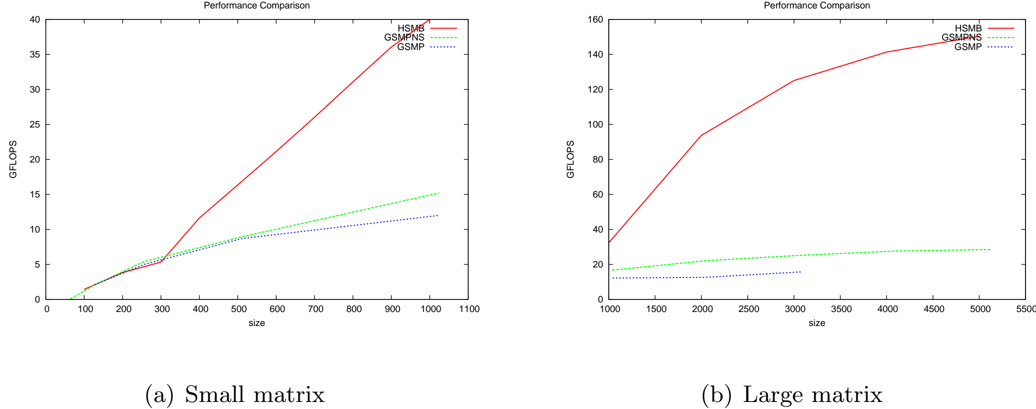
GSMB is slower than both HSMB and GSMP, due to the reason discussed above. Therefore, GSMB is not considered for further optimization and verification.

According to the analysis above, when the scratchpad memory can store one column or two rows of the matrix A, GSMP has less communication time than GSMR. When the scratchpad memory can not store one column or two rows of the matrix A, GSMPNS has less communication time than GSMRNS. Therefore, GSMP and GSMPNS are selected for further optimization and verification.

4.4 Experimental Results and Analysis

Several optimizations are performed on GSMP and GSMPNS, including global memory coalescing, shared memory coalescing, caching, loop unrolling, panel size tuning, block size tuning. The global memory coalescing provides an improvement with a factor of 10 in performance, while the other optimization techniques provide minor improvements. Loop unrolling even causes 10% performance degradation, because additional register usage limits the number of concurrent threads on each SM. Results of Givens QR algorithms for large and small matrices are shown in Figure 15.

The bottlenecks of GSMP and GSMPNS are identified using an analytical model. The bottleneck of GSMPNS is $(CWP > MWP)$, which means the performance is bounded by memory accesses. The analytical model shows GSMP is computation bounded. However, this does not mean GSMP can fully utilize the GPU. For large matrices of 3072×2048, GSMP



(a) Small matrix (b) Large matrix

Figure 15: Comparing HSMB with GSMP and GSMPNS

can run at most one thread block per SM. If *threads_per_block* is small (< 128), PWP causes *comp_cycles* to increase significantly. If *threads_per_block* is large (> 256), *comp_cycles* will increase because more data is processed in the RP stage where more branches and synchronization instructions are executed. Figure 16 shows the results of benchmark and analytical model. Benchmarks and the analytical model both show that best performance is obtained when *threads_per_block* is between 128 and 256. This is the result of trade-offs between PWP and RP overhead. On the other hand, HSMB performs matrix multiplications on GPUs, which is neither limited by PWP nor control overhead. This explains the reason Givens QR is not able to outperform Householder QR on GPUs.

5 Architecture Exploration

Graphics Processing Units(GPUs) have evolved from dedicated hardware to programmable processors. NVIDIA GeForce has massive number of Processing Elements (PEs) that can reach peak performance of 500 GFLOPS per chip, which is an order of magnitude higher than CPUs. GeForce uses interleaved multithreading, also called barrel processing, to hide off-chip memory latency. The performance of such architecture may be limited by available threads, register pressure, scratchpad memory size, bandwidth. Given a budget of chip area, it is difficult to make decisions on optimum PEs number, registers size, scratchpad size, bandwidth, that lead to highest performance. This report proposed a model to answer these questions.

Zvika Guz, et al.[12] published a model to analyze the trade-offs between many-core approach and many-thread approach. Their model shows, given suitable applications and architectural parameters, the many-thread approach can reach higher performance than the few-thread approach. However, their model has limitations. There are two important questions which cannot be answered by their model:

1. Computer architects are usually given a budget of chip area, which has great impacts on performance. Thus this question need to answered: "Given an application and a specific chip area, what is the optimum number of cores and threads?"

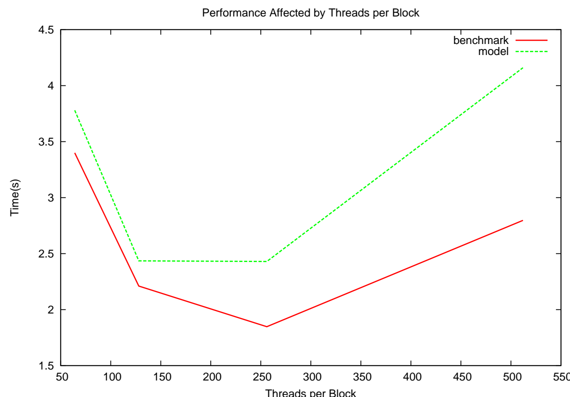


Figure 16: Performance of GSMP by benchmark and analytical model ($m = 3072$, $n = 2048$, $b = 4$). Running time is affected by threads per block. The analytical model overestimates the RP overhead, because it is pessimistic on branch[15].

2. Scratchpad memory is superior to cache in terms of performance, area, and power consumption[4]. Scratchpad memory is widely used in embedded systems, such as Cell and GeForce. The model of Guz uses cache as local memory, which is the main cause of the performance valley. Replacing cache with scratchpad memory is not straightforward. An elementary approach would flatten the valley, as shown in Section 5.1, which provides little information on design trade-offs. A more advanced model, which considers both area and characteristics of applications, is necessary to predict the performance.

This report introduces an equal-area model. It has the following contributions.

1. We proposed an area model for GPU-like architecture. The area of register files and scratchpad memory will constraint the number of concurrent threads.

2. The proposed model utilizes memory access patterns of the application, mainly on data locality and working set size. This report uses matrix multiplication as example, but the model can be applied to applications with other memory access patterns.

3. The proposed model is able to answer this question: "Given an application and specific chip area, what is the optimum number of cores and threads?"

5.1 An Elementary Scratchpad Memory Model

To the best of my knowledge, the model proposed by Zvika Guz, et al.[12] is the first and only one that tries to analyze the trade-offs between Larrabee-like large-memory few-thread architecture and GeForce-like small-memory many-thread architecture. The model of Guz uses cache as local memory. In GeForce architecture, the local memory is scratchpad. Although GeForce also provides caches for SMs, they are read-only and do not maintain consistency with global memory. The constant cache has limited bandwidth, while the texture cache has large latency[35]. Therefore, GPU kernels store working sets of threads on the scratchpad memory.

Table 7: Parameters of Elementary Scratchpad Model

Parameter	Description	Value
N_{PE}	Number of PEs	1024
f	Frequency of PEs	1 (GHz)
CPI_{exe}	Cycle per instruction	1
t_{avg}	Average global memory latency	600 cycles
r_m	Global mem access to computation ratio	0.05 to 0.2
$smem$	Scratchpad memory size	128 (KB)
BW_{max}	Memory bandwidth	50 to 100 (GB/s)
b_{reg}	Load/Store size per thread per access	4 (Bytes)
$workingset_{min}$	Minimum working set per thread	12 (Bytes)

Zvika Guz's model uses Equation 30 and 31 to measure the performance and utilization. The model uses the parameter η to represent the utilization of the PEs.

$$Performance = N_{PE} \cdot \frac{f}{CPI_{exe}} \cdot \eta \tag{30}$$

$$\eta = min\left(1, \frac{n_threads}{N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)}\right) \tag{31}$$

To model the scratchpad memory, the formula of t_{avg} and r_m need to be changed accordingly. In GeForce, the access latency to scratchpad memory is comparable to register latency, which can be hidden by a small number of concurrent threads. Thus we only consider the latency of accessing global memory. We assume t_{avg} is 600 cycles. Typical value of r_m for applications mapped to GPUs is between 0.05 to 0.2[35]. To model the effects of BW limitation, Equation 30 is rewritten into 32. To model the limitation of scratchpad memory size, $smem$, we assume on average each thread has a working set size $workingset_{min}$. If the number of concurrent threads is larger than $smem/workingset_{min}$, we assume only a number of threads with working sets fitting within the scratchpad memory can proceed. To model the limitation of scratchpad memory size, Equation 31 is changed to Equation 33. A summary of the model parameters is listed in Table 7. Here assumes each thread has a working set size of three words, two as input data and one as output data.

$$Performance = min\left(N_{PE} \cdot \frac{f}{CPI_{exe}} \cdot \eta,\ \frac{BW_{max}}{r_m \cdot b_{reg}}\right) \tag{32}$$

$$\eta = min\left(1, \frac{min\left(n_threads, \frac{smem}{workingset_{min}}\right)}{N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)}\right) \tag{33}$$

The result of this elementary model is shown in Figure 17. The bandwidth and scratchpad memory size introduce cut-off effects on the performance. Due to the absence of area information and simple assumptions on both application and architecture, this elementary model provides little information for design trade-offs. Thus we move on to a more advanced model.

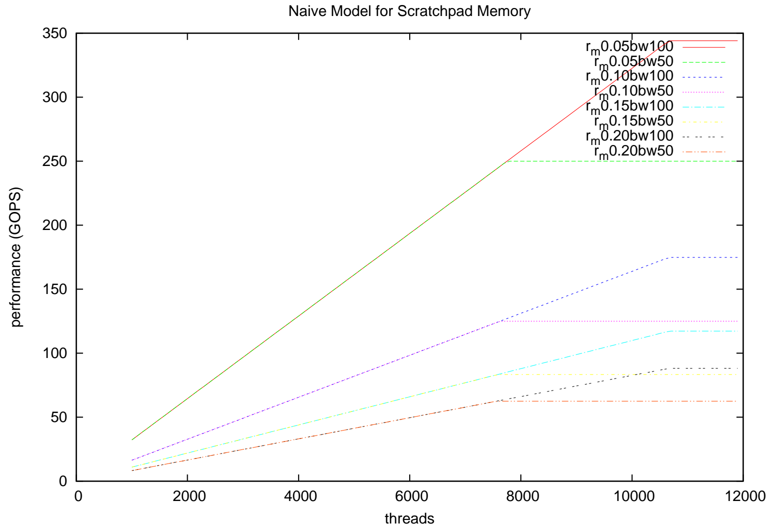


Figure 17: An elementary model for scratchpad memory

5.2 An Equal-Area Model

Based on the observation of the elementary model, the equal-area model makes the following improvements.

1. It models the area of register files and scratchpad memory required by concurrent threads.
2. It models the register usage and working set size of applications.
3. It models the memory bandwidth using Memory Warp Parallelism[15], which introduces cut-off effect on *effective_threads*, as shown in Equation 38. It provides better performance estimation than Guz's model, in which bandwidth introduces cut-off effects only on final performance.

The equal-area model reuses the performance equation of Guz's model, as shown in Equation 34. We change the utilization factor η into Equation 35.

$$Performance = N_{PE} \cdot \frac{f}{CPI_{exe}} \cdot \eta \tag{34}$$

$$\eta = min\left(1, \frac{effective_threads}{N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)}\right) \tag{35}$$

We define *effective_threads* as the number of concurrent threads that can be used to hide global memory latency, as shown by Equation 36. *effective_threads* are limited by the availability of scratchpad memory for its working set and the bandwidth for concurrent memory transfer. Define *threads_full_smem* as the maximum number of concurrent threads with working

Table 8: Area of different components

Components	Description	Obtained	Area (mm^2)
Total Area of 16 SM	Without Tex Cache	Die Photo	118.88
Area of One SM	Without Tex Cache	Computed	$\frac{118.88}{16} = 7.43$
16KB Shared Scratchpad Memory	16 banks 1RW	CACTI	0.59
8096 32-bit Register File	16 lanes 2R1W	CACTI	2.85
8 PEs, 2 SFUs, Constant Cache, I-Cache, MT Issue Unit, etc.		Computed	$7.43 - 0.59 - 2.85 = 3.99$

Table 9: Values of Area Parameters

Parameters	Obtained	Value (mm^2)
smem_area_per_word	Measured scratchpad area divided by size.	$\frac{0.59}{4\times1024} = 0.144 \times 10^{-3}$
reg_area_per_word	Measured register area divided by size.	$\frac{2.85}{8096} = 0.352 \times 10^{-3}$
Total_area	Die Photo	118.88
fixed_area_per_SM	Computed in Table 8	3.99

set data on shared memory. Assume on average each thread requires at least $workingset_{min}$ bytes of shared memory, which is application specific. The value of *threads_full_smem* is computed by Equation 37.

$$effective_threads = min\,(n_threads,\ threads_full_smem,\ threads_full_BW) \tag{36}$$

$$threads_full_smem = N_{SM} \times \frac{smem_{SM}}{workingset_{min}} \tag{37}$$

$$threads_full_BW = \frac{bandwidth \times t_{avg}}{f \times bytes_per_thread} \tag{38}$$

The die area of GeForce 8800 GTX is 470 mm^2 in 90 nm technology[24]. According to the measurement of die photo, approximately 119 mm^2 is consumed by 16 Stream Multiprocessors. Our model sets 118.88 mm^2 as the area limit that can be used for SMs. The area of each SM is 7.43 mm^2, which includes 1.03 mm^2 for scratchpad memory and 2.63 mm^2. The remaining area of 3.77 mm^2 consists of 8 PEs, 2 SFUs, Constant cache, Instruction Cache, MT Issue Unit, etc., which is consider to be fixed in the model. The area of each component is listed in Table 8.

Based on the result of previous experiments[39], this model assumes the area of register file and scratchpad memory scales linearly with the storage capacity. The scaling parameters are provided in Table 9.

The size of scratchpad memory per SM, *smem*, is calculated by Equation 39.

$$smem = \frac{total_area - n_threads \times reg_per_thread \times reg_area_per_word}{N_{SM} \times smem_area_per_word} \tag{39}$$

$$N_{SM} = \frac{N_{PE}}{PE_per_SM} \tag{40}$$

In systems with scratchpad memory, r_m is highly application dependent. Thus, this report uses matrix multiplication as an example. For matrix multiplication, the minimum working

Table 10: Values of Parameters

Components	Description	Obtained	Value
Total_area	Area for all SMs	fixed	118.88 mm^2
Fixed_area_per_sm	fixed area per SM	fixed	3.99 mm^2
Smem_area_per_word	smem area per word	fixed	0.144×10^{-3} mm^2
Reg_area_per_word	register area per word	fixed	0.352×10^{-3} mm^2
PEs_per_SM	number of PEs per SM	fixed	8
N_{PE}	Number of PE	Tuning parameter	128 to 216
n_threads	Number of concurrent threads	Tuning parameter	1000 to 10000
N_{SM}	number of SM	Computed	Equation 40
Bandwidth	bandwidth to global memory	Tuning parameter	32 to 104(GB/s)
Regs_per_thread	register used per thread	Tuning parameter	10 to 30
CPI_{exe}	Cycle per instruction of PE	Tuning parameters	1
f	frequency of PE	Tuning parameter	1.5(GHz)
t_{avg}	latency to global memory	Tuning parameter	400 cycles
r_m	memory to computation ratio	Computed	Equation 41
Smem	size of smem in words	Computed	Equation 39
Effective_threads	n_threads limited by smem and BW	Computed	Equation 36
Threads_full_smem	n_threads limited by smem size	Computed	Equation 37
Threads_full_BW	n_threads limited by bandwidth	Computed	Equation 38
Bytes_per_thread	ld/st bytes per thread	Tuning parameter	4 bytes
$workingset_{min}$	Minimum workingset per thread	Tuning parameter	3 words

set for a thread is two input elements and one output elements. Thus $workingset_{min}$ is equal to 3 words. For matrix multiplication between matrices of size $m \times k$ and $k \times n$, the amount of MAC computation is $k \times m \times n$. For blocked matrix multiplication of block size $bm \times bk$ and $bk \times bn$, the amount of memory access is $k \times m \times n \times (\frac{1}{bm} + \frac{1}{bn})$[43]. To generalize the analysis, we assume bm is equal to bn. Thus r_m is equal to $\frac{2}{bn}$. Increasing block size bn will reduce r_m, which will increase performance, as shown by Equation 35. On the other hand, bn is limited by the size of scratchpad memory, $smem$. From the above discussion, the minimum value of r_m for matrix multiplication is shown in Equation 41.

$$r_m = \frac{2}{\sqrt{smem}} \tag{41}$$

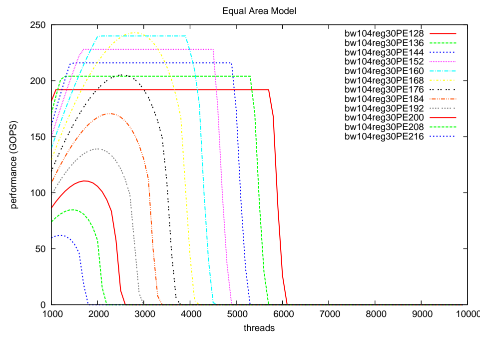
Table 10 summarizes parameters and their values specified in the following discussion. The result of tuning is shown in Figure 18.

5.3 Analysis

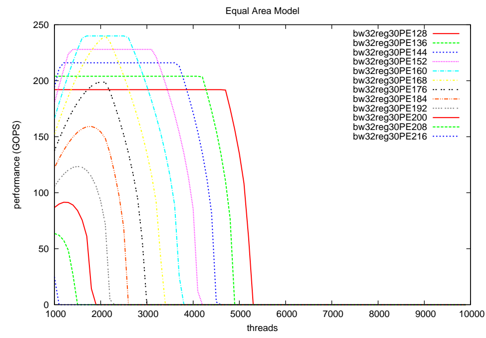
Based on results shown in Figure 18, we have the following observations and explanations.

1. Given specific bandwidth, *regs_per_threads* and N_{PE}, we observe that the performance and the number of threads have the following relations. While *n_threads* are small, increasing *n_threads* will increase the performance, because additional threads can hide memory latency. While the number of threads is enough to hide memory latency, the performance may enter a flatten performance region where PEs are fully utilized, namely, η is equal to 1. However, if the number of PEs is large, the flatten performance region may not be seen, because there may not be enough area left for registers and scratchpad, which means *Effective_threads* are limited by either *Threads_full_BW* or *Threads_full_smem* before η reaches 1.

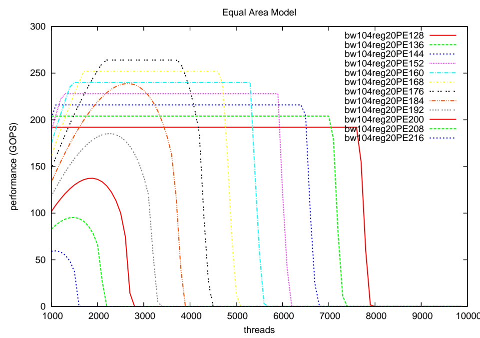
   (a) *Effective_threads* limited by *Threads_full_BW*. For systems with high bandwidth and large *regs_per_threads*, such as Figure 18(a), *effective_threads* is always limited



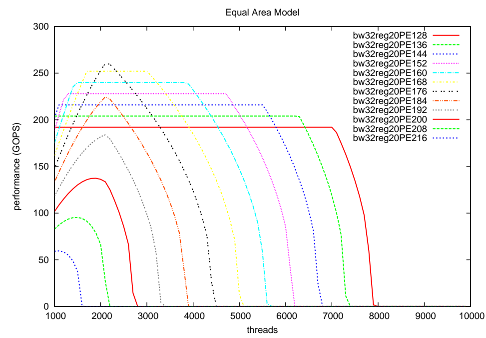
(a) bw104reg30, $OPT_{N_{PE}} = 160$



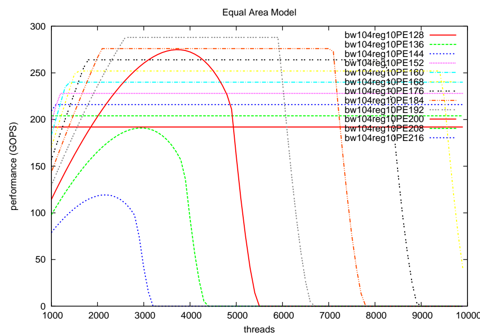
(b) bw32reg30, $OPT_{N_{PE}} = 160$



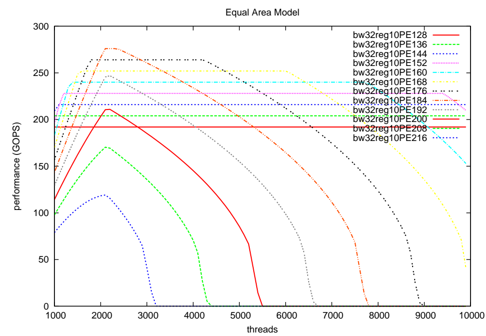
(c) bw104reg20, $OPT_{N_{PE}} = 176$



(d) bw32reg20, $OPT_{N_{PE}} = 176$



(e) bw104reg10, $OPT_{N_{PE}} = 192$



(f) bw32reg10, $OPT_{N_{PE}} = 184$

Figure 18: Tuning *bandwidth*(bw), *regs_per_thread*(reg) and N_{PE}(PE) to find the optimum number of PEs ($OPT_{N_{PE}}$).

by *threads_full_smem*, because limited scratchpad size stops *effective_threads* from growing to saturating the bandwidth.

(b) *Effective_threads* limited by *Threads_full_smem*. For systems with small bandwidth and small *regs_per_thread*, such as Figure 18(f), *effective_threads* will be limited first by *threads_full_BW*, because more threads can have $workingset_{min}$ in scratchpad memory, which will saturate the relatively small bandwidth.

2. Given specific bandwidth and *regs_per_threads*, there exists an optimum number of PEs for highest performance. While N_{PE} is small, they can be fully utilized with relatively less threads, because more area can be used as scratchpad, which reduces r_m. With small N_{PE}, peak performance can be reached with fewer threads, but it comes at a price of low peak performance. Increasing the number of PE can increase the peak performance, but the peak performance may not be reach if N_{PE} is too large, because *effective_threads* are limited by either *threads_full_smem* or *threads_full_BW*, as described above.

3. Given specific bandwidth, the optimum number of PE ($OPT_{N_{PE}}$) increases as *regs_per_thread* decreases. For the same number of *n_threads* and N_{PE}, smaller *regs_per_thread* allows more area for scratchpad memory, which reduces r_m and increases utilization.

4. Given specific *regs_per_thread*, reducing bandwidth may reduce $OPT_{N_{PE}}$, as shown in Figure 18(e) and 18(f), but it also may not reduce $OPT_{N_{PE}}$, as shown in Figure 18(c) and 18(d). The bandwidth has an cut-off effect on *effective_threads*, as shown in Figure 19. Whether the reduction of bandwidth changes optimum number of PE depends on whether the N_{PE} of highest performance on the cut has changed while moving the cut from right to left.

6 Related Work

Sunpyo Hong and Hyesoon Kim have proposed the first analytical model for GPUs[15]. Vasily Volkov and James Demmel have benchmarked dense linear algebra on GPUs, and mapped parts of the Householder QR algorithm on GPUs[43]. Andrew Kerr, Dan Campbell and Mark Richards have mapped the complete Householder QR algorithm on GPUs[19]. Andrew Kerr, Gregory Diamos, et al. have analyzed the characteristics of CUDA kernels and working on ocelet tool chain[6][20]. Wilson Fung, et al. have proposed dynamic warp formation for GPU architecture[10]. Ali Bakhoda, et al. have released GPGPU-sim tool chain that simulates ptx code[3].

Interleaved multithreading, also called vertical threading or barrel processing, has been adopted for decades[42]. The Multithreaded Vector Architecture[8] proposed by Roger Espasa is an example of interleaved multithreading applied to vector architecture. The Simultaneous Multithreaded(SMT) Vector Architecture[8] proposed by Roger Espasa, et al. has extended multithreading to horizontal direction. In Espasa's SMT vector architecture, the register file for different threads are not decoupled, instead, they are renamed into the same register pool, in a similar way as SMT superscalar architectures[40][14]. The complexity of register file limits its scalability for larger number of threads.

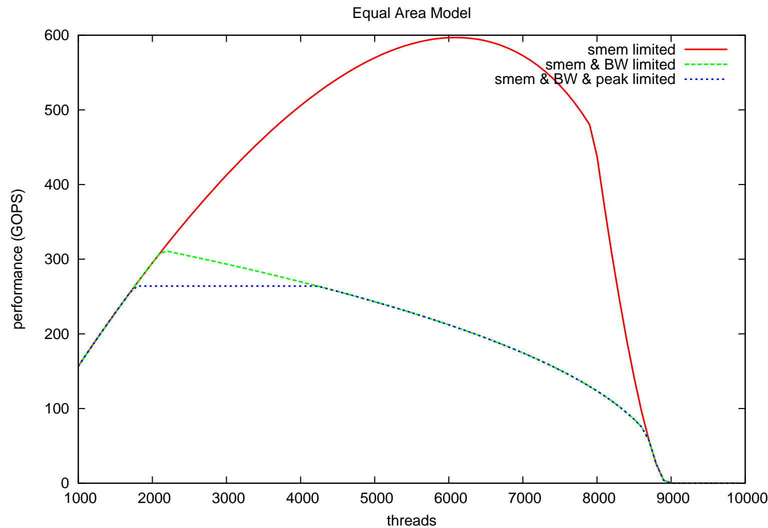


Figure 19: Limitations of performance

Stephen W. Keckler and William J. Dally have proposed the concept of processor coupling[16], which is implemented in the M-Machine[9]. M-Machine uses both vertical and horizontal multithreading, described as V-Thread and H-Thread, to fill the VLIW PEs. M-Machine replaces the lock-step execution of [16] with barrier synchronization, which is similar to CUDA programming model. M-Machine maps MIMD threads onto MIMD hardware. On the other hand, the SIMT architecture maps MIMD threads onto SIMD hardware. M-Machine uses VLIW PEs to utilize ILP within each thread, which is not present in current SIMT architecture. There are other attempts to apply horizontal multithreading to VLIW processor at operation level[29] and cluster level[11].

Tirath Ramdas, Gregory K. Egan, et al. have proposed methods to map MIMD threads onto SIMD hardware, which converts Thread Level Parallelism(TLP) into Data Level Parallelism(DLP) at run time[32]. Their architecture does not decouple threads on horizontal and vertical directions, which increases the complexity of thread selection logic. They introduced the concept of thread window[33] as a complexity effective way to implement thread selection. This architecture was designed for computational chemistry applications. However, some of the concepts, such as thread window, may be applied to SIMT architecture.

Vector Lane Threading(VLT), proposed by Suzanne Rivoire, et al. uses horizontal multithreading on vector lanes[34]. Each vector thread has independent instruction stream fetched by control processors. To provide sufficient instruction fetch bandwidth, the control processors are duplicated(CMP), or multiplexed(SMT), or both(CMT). Although VLT introduces little overhead on the vector lanes, it requires complex control processor.

The Vector-Thread(VT) architecture[23], implemented in the SCALE processor[22], was proposed by Ronny Krashinsky, et al. It utilizes both vertical threading and horizontal threading over its Virtual Processors(VPs). The programming model hides the vector width of the hardware by mapping VPs to hardware lanes at run time. Horizontal threads can

execute different instruction streams. To provide enough instruction bandwidth, instruction locality are exploited by both compiler and hardware. The VT compiler[13] converts Loop-Level Parallelism(LLP) into TLP at compile time. The VT compiler also exploits instruction locality by forming Atomic Instruction Block(AIB). Complexity-effectiveness and performance is an important trade-off in the design of SCALE processor[21], which leads to an area efficient implementation.

The Multithreaded Lockstep Execution Processor(MELP) proposed by Jaegeun Oh, et al.[28] is another example of converting TLP into DLP by mapping MIMD threads onto SIMD hardware. Threads in MELP are synchronized at barrier, which is similar to SIMT architecture. MELP allows threads to diverge at the branch, executing both branches in a similar way as SIMT architecture. However, MELP only uses horizontal threading, due to its simple memory hierarchy and short pipeline.

A source-to-source SIMT code generation method was proposed by Cornwall, et al.[5]. It is based on algorithmic skeleton and focuses on computer vision application. CUDA-lite[41], a source-to-source SIMT code generation method, was proposed to facilitate memory optimization.

There are recent attempts to explore the design space of accelerator architecture. Hung Ho Ahn, Mattan Erez, and William J. Dally have performed design space exploration on ILP, DLP, and TLP in stream processors[1]. The TLP investigated in their design space is horizontal threading on cluster-level. Aqeel Mahesri, et al. have explored the design space of accelerator architecture for visual computing[25]. They came to the conclusion that, with the same area, MIMD core outperforms SIMD core for a broad range of visual computing applications. In their proposed architecture, two vertical threads are enough to hide memory latency. Partially based on these results, John Kelm, et al. proposed the Rigel[17] architecture and programming model. Rigel achieves high computation density and energy efficiency, but requires applications to have high computation to memory ratio. The design space of architecture is greatly affected by the characteristics of the applications. Andrew Kerr, et al.[20] proposed a set of metrics, including control flow, data flow, SIMD and MIMD parallelism, etc. for the analysis of GPGPU applications. These metrics of application can be used as the guidance for architectures exploration.

7 Conclusion

This report investigates existing analytical model and extends it with kernel launch overhead and pipeline warp parallelism(PWP). As shown in our experiment, when there are not enough threads to hide pipeline latency, existing model can not provide accurate estimation of running time. Therefore, extending existing model with PWP is necessary. Experiment result shows the extension of PWP solves this problem.

This report analyzes performance of different QR factorization algorithms mapped to distributed memory systems and shared memory systems. The analysis can explain the observations of previous work and our experiment. The bottleneck of QR algorithms is identified by the extended analytical model. The analysis and experiment results lead to the conclusion that Householder QR factorization is more suitable for GPUs than Givens QR factorization.

This report proposes an equal-area model for architecture exploration. The equal-area model is based on GeForce-like many-thread architecture. It considers the data locality of

applications and the area constraints of architecture. In the matrix multiplication example, it estimates the performance, bottleneck, and optimum number of cores and threads for the given chip area. It shows the trade-offs of architecture decisions are complicated, even for a simple application. Therefore, an equal-area model is necessary and important in the design space exploration of many-thread architecture.

8 Future Work

Construct a unified model for architecture exploration. Power consumption need be considered in the future architecture exploration. Evaluate SIMT architecture for other application domains. Improve existing methods of mapping algorithmic skeleton to SIMT architecture.

9 Acknowledgments

Special thanks to Prof. Hyesoon Kim for providing early evaluation of their analytical model. I thank Ali Bakhoda for providing early access of their GPGPU-sim. I thank Wilson Fung for answering my question on dynamic warp formation. I thank Bart Mesman and Henk Corporaal for their guidance and encouragement during the project. I thank Cederic Nugteren, Gert-Jan Braak, and Yifan He for helpful discussions on this project. I thank all the people supporting me, without whom this work would be impossible.

References

[1] J. Ahn, M. Erez, and W. Dally. Tradeoff between data-, instruction-, and thread-level parallelism in stream processors. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 126–137. ACM New York, NY, USA, 2007.

[2] ATI. ATI Stream SDK User Guide (v1.4-beta). 2009.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed GPU simulator. *IEEE ISPASS*, 2009.

[4] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM New York, NY, USA, 2002.

[5] J. Cornwall, L. Howes, P. Kelly, P. Parsonage, and B. Nicoletti. High-performance SIMT code generation in an active visual effects library. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 175–184. ACM New York, NY, USA, 2009.

[6] G. Diamos, A. Kerr, and M. Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technology, January 2009.

[7] O. Egecioglu. Givens and Householder reductions for linear least squares on a cluster of workstations. 1995.

[8] R. Espasa and M. Valero. Multithreaded vector architectures. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 237–248, 1997.

[9] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 146–156. IEEE Computer Society Press Los Alamitos, CA, USA, 1995.

[10] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation: Efficient mimd control flow on simd graphics hardware. *ACM Trans. Archit. Code Optim.*, 2009.

[11] M. Gupta and F. Sanchez. Cluster-level simultaneous multithreading for VLIW processors. pages 121–128, 2007.

[12] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *Computer Architecture Letters*, 8(1):25–28, Jan. 2009.

[13] M. Hampton and K. Asanovic. Compiling for vector-thread architectures. In *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 205–215. ACM New York, NY, USA, 2008.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1(2), 2001.

[15] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, New York, NY, USA, 2009. ACM.

[16] S. Keckler and W. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 202–213, 1992.

[17] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 140–151, New York, NY, USA, 2009. ACM.

[18] A. Kerr, D. Campbell, and M. Richards. GPU Performance Assessment with the HPEC Challenge. In *HPEC Workshop 2008*.

[19] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 71–78. ACM New York, NY, USA, 2009.

[20] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of gpgpu kernels. Technical Report GIT-CERCS-09-06, Georgia Institute of Technology, June 2009.

[21] R. Krashinsky. *Vector-thread architecture and implementation*. PhD thesis, Massachusetts Institute of Technology, 2007.

[22] R. Krashinsky, C. Batten, and K. Asanović. Implementing the scale vector-thread processor. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–24, 2008.

[23] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 52–63, 2004.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[25] A. Mahesri, D. Johnson, N. Crago, and S. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture-Volume 00*, pages 164–175. IEEE Computer Society Washington, DC, USA, 2008.

[26] M. McGraw-Herdeg, D. Enright, and B. Michel. Benchmarking the NVIDIA 8800GTX with the CUDA Development Platform. *HPEC 2007 Proceedings*.

[27] A. Munshi, A. Wong, A. Clinton, S. Braganza, W. Bishop, and M. McCool. A parameterizable SIMD stream processor. pages 806–811, 2005.

[28] J. Oh, S. Hwang, H. Nguyen, A. Kim, S. Kim, C. Kim, and J. Kim. Exploiting Thread-Level Parallelism in Lockstep Execution by Partially Duplicating a Single Pipeline. *ETRI Journal*, 30(4):576–586, 2008.

[29] E. Ozer, T. Conte, A. Ltd, and U. Cambridge. High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1132–1142, 2005.

[30] D. Patterson and J. Hennessy. Computer Organization and Design: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design). 2008.

[31] A. Pothen, S. Jha, and U. Vemulapati. Orthogonal factorization on a distributed memory multiprocessor. In *Hypercube Multiprocessors, 1987: Proceedings of the Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee, September 29-October 1, 1986*, page 587. Society for Industrial & Applied, 1987.

[32] T. Ramdas, G. Egan, D. Abramson, and K. Baldridge. Converting massive TLP to DLP: a special-purpose processor for molecular orbital computations. In *Proceedings of the 4th international conference on Computing frontiers*, pages 267–276. ACM New York, NY, USA, 2007.

[33] T. Ramdas, G. Egan, D. Abramson, and K. Baldridge. Run-time thread sorting to expose data-level parallelism. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 55–60, 2008.

[34] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 55–64, 2006.

[35] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Wen-mei. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM New York, NY, USA, 2008.

[36] R. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. 1987.

[37] M. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis. A low-power multithreaded processor for software defined radio. *The Journal of VLSI Signal Processing*, 43(2):143–159, 2006.

[38] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. pages 1–15, 2008.

[39] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, 2001.

[40] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505, 2005.

[41] S. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-lite: Reducing GPU programming complexity. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2008.

[42] T. Ungerer and B. Robic. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[43] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

Appendices

A Area estimation with CACTI 3.2

The area of shared memory and register files in Table 8 are estimated by CACTI 3.2[7]. The usage of of the tool is described in [39]. To generate the area for scratchpad memory and register file, we remove the tag area from cacti, as shown in Table 11 and 12.

Table 11: Area of Scratchpad Memory

Command arguments	**Description**	**Value**
C	Cache size (Bytes)	1024
B	Block size (Bytes)	4
A	Associativity	1
TECH	Technology (μm)	0.090
RWP	Number of read/write port	1
RP	Number of read port	0
WP	Number of write port	0
NSubbanks	Number of subbanks	1
Program variables	**Description**	**Value**
ADDRESS_BITS	Address bus width in bits	8
BITOUT	Data bus width in bits	32
Results	**Description**	**Value**
Total area(mm^2)	Not including tag components	0.037
Scratchpad memory area(mm^2)	16 banks, each of 1 KB	0.59

Table 12: Area of Register File

Command arguments	**Description**	**Value**
C	Cache size (Bytes)	32768
B	Block size (Bytes)	64
A	Associativity	1
TECH	Technology (μm)	0.090
RWP	Number of read/write port	0
RP	Number of read port	2
WP	Number of write port	1
NSubbanks	Number of subbanks	1
Program variables	**Description**	**Value**
ADDRESS_BITS	Address bus width in bits	9
BITOUT	Data bus width in bits	512
Results	**Description**	**Value**
Total area(mm^2)	Not including tag components	2.85

[7] Available at http://www.hpl.hp.com/research/cacti/. Some program variables in source code need to be modified.

B Examples of QR Factorization

A 4×4 matrix A is used as an example to illustrate the procedures to perform Householder QR factorization and Givens QR factorization.

$$A = \begin{pmatrix} 1 & 3 & 2 & 1 \\ 1 & 1 & 4 & 1 \\ 1 & 3 & 4 & 1 \\ 1 & 1 & 2 & -3 \end{pmatrix}$$

B.1 Householder QR Factorization

Householder QR factorization annihilates A in 3 iterations.

1. Generate v_1 and τ_1 to form H_1.

$$v_1 = (1, 0.33, 0.33, 0.33)^T, \ \tau_1 = 1.5$$

   Apply H_1 to A.

$$H_1 A = \begin{pmatrix} -2 & -4 & -6 & 0 \\ 0 & -1.33 & 1.33 & 0.67 \\ 0 & 0.66 & 1.33 & 0.67 \\ 0 & -1.33 & -0.67 & -3.33 \end{pmatrix}$$

2. Generate v_2 and τ_2 for H_2.

$$v_2 = (0, 1, -0.2, 0.4)^T, \ \tau_2 = 1.67$$

   Apply H_2 to A.

$$H_2 H_1 A = \begin{pmatrix} -2 & -4 & -6 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 1.6 & 0.4 \\ 0 & 0 & -1.2 & -2.8 \end{pmatrix}$$

3. Generate v_3 and τ_3 for H_3.

$$v_3 = (0, 0, 1, -0.33)^T, \ \tau_3 = 1.8$$

   Apply H_3 to A.

$$H_3 H_2 H_1 A = \begin{pmatrix} -2 & -4 & -6 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & 0 & -2 \end{pmatrix}$$

B.2 Givens QR Factorization

Givens QR factorization annihilates A in 6 iterations.

1. Generate $G(3,4,\theta)$ to annihilate $A(4,1)$ with $A(3,1)$.

$$G(3,4,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.71 & 0.71 \\ 0 & 0 & -0.71 & 0.71 \end{pmatrix}$$

Apply $G(3,4,\theta)$ to A.

$$G(3,4,\theta)A = \begin{pmatrix} 1 & 3 & 2 & 1 \\ 1 & 1 & 4 & 1 \\ 1.41 & 2.83 & 4.24 & -1.41 \\ 0 & -1.41 & -1.41 & -2.83 \end{pmatrix}$$

2. Generate $G(2,3,\theta)$ to annihilate $A(3,1)$ with $A(2,1)$.

$$G(2,3,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.58 & 0.82 & 0 \\ 0 & -0.82 & 0.58 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Apply $G(2,3,\theta)$ to A.

$$G(2,3,\theta)G(3,4,\theta)A = \begin{pmatrix} 1 & 3 & 2 & 1 \\ 1.73 & 2.89 & 5.78 & -0.58 \\ 0 & 0.82 & -0.82 & -1.63 \\ 0 & -1.41 & -1.41 & -2.83 \end{pmatrix}$$

3. Generate $G(1,2,\theta)$ to annihilate $A(2,1)$ with $A(1,1)$.

$$G(1,2,\theta) = \begin{pmatrix} 0.5 & 0.86 & 0 & 0 \\ -0.86 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Apply $G(1,2,\theta)$ to A.

$$G(1,2,\theta)G(2,3,\theta)G(3,4,\theta)A = \begin{pmatrix} 2 & 4 & 6 & 0 \\ 0 & -1.15 & 1.15 & -1.15 \\ 0 & 0.82 & -0.82 & -1.63 \\ 0 & -1.41 & -1.41 & -2.83 \end{pmatrix}$$

4. Generate $G(3,4,\theta)$ to annihilate $A(4,2)$ with $A(3,2)$.

$$G(3,4,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.5 & 0.87 \\ 0 & 0 & -0.87 & -0.5 \end{pmatrix}$$

Apply $G(3,4,\theta)$ to A.

$$G(3,4,\theta)G(1,2,\theta)G(2,3,\theta)G(3,4,\theta)A = \begin{pmatrix} 2 & 4 & 6 & 0 \\ 0 & -1.15 & 1.15 & -1.15 \\ 0 & -1.63 & -0.82 & -1.63 \\ 0 & 0 & 1.41 & 2.83 \end{pmatrix}$$

5. Generate $G(2,3,\theta)$ to annihilate $A(3,2)$ with $A(2,2)$.

$$G(2,3,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.58 & 0.82 & 0 \\ 0 & -0.82 & 0.58 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Apply $G(2,3,\theta)$ to A.

$$G(2,3,\theta)G(3,4,\theta)G(1,2,\theta)G(2,3,\theta)G(3,4,\theta)A = \begin{pmatrix} 2 & 4 & 6 & 0 \\ 0 & -2 & 0 & -2 \\ 0 & 0 & -1.41 & 0 \\ 0 & 0 & 1.41 & 2.83 \end{pmatrix}$$

6. Generate $G(3,4,\theta)$ to annihilate $A(4,3)$ with $A(3,3)$.

$$G(3,4,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.71 & 0.71 \\ 0 & 0 & -0.71 & -0.71 \end{pmatrix}$$

Apply $G(3,4,\theta)$ to A.

$$G(3,4,\theta)G(2,3,\theta)G(3,4,\theta)G(1,2,\theta)G(2,3,\theta)G(3,4,\theta)A = \begin{pmatrix} 2 & 4 & 6 & 0 \\ 0 & -2 & 0 & -2 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$