Performance Gains in Conjugate Gradient Computation with Linearly Connected GPU Multiprocessors

Performance Gains in Conjugate Gradient Computation with Linearly Connected GPU Multiprocessors

Stephen J. Tarsa
Harvard University

Tsung-Han Lin
Harvard University

H.T. Kung
Harvard University

Abstract

Conjugate gradient is an important iterative method used for solving least squares problems. It is compute-bound and generally involves only simple matrix computations. One would expect that we could fully parallelize such computation on the GPU architecture with multiple Stream Multiprocessors (SMs), each consisting of many SIMD processing units. While implementing a conjugate gradient method for compressive sensing signal reconstruction, we have noticed that large speed-up due to parallel processing is actually infeasible due to the high I/O cost between SMs and GPU global memory. We have found that if SMs were linearly connected, we could gain a 15x speedup by loop unrolling. We conclude that adding these relatively inexpensive neighbor connections for SMs can significantly enhance the applicability of GPUs to a large class of similar matrix computations.

1 Introduction

The recent ubiquity of cheap, powerful GPUs, featuring hundreds of processing units (PUs) at a cost of less than \$1 per PU, has made them a primary building block of modern high-performance computing. Built to accelerate real-time graphics rendering, GPUs are comprised of multiple independent Stream Processors (SMs), each consisting of many SIMD PUs. In general, they are amenable to embarassingly data parallel tasks that require high computational throughput. Active research in recent years has focused on adapting algorithms in various application domains to the specifics of this architecture, and general-purpose GPU computing has found success in diverse fields including finance and biology [1].

However, in the process of adapting a class of signal processing algorithms for the GPU, we notice that even in a simple case of computation on linear systems with small input data sets, popular iterative algorithms like conjugate gradient methods cannot be efficiently implemented across multiple SMs. Fundamentally, this is due to synchronization and I/O costs that occur whenever such algorithms must aggregate operands computed by different SMs, e.g. to calculate vector norms, or sort component magnitudes. We show that synchronization on the current GPU architecture, either using the host CPU, or with a barrier implemented in GPU global memory, causes running time to be dominated by the high latency penalty between SMs and global memory. In this paper, we argue that one dimensional (1D) interconnections between neighboring SMs would substantially increase parallel speedup gains on future generations of GPUs by reducing synchronization and operand aggregation time.

We focus on optimizing the conjugate gradient computation to demonstrate these limitations, and showcase the possible speedups from such suggested interconnections. Conjugate gradient is well-known and widely-applied for solving least squares problems and finding solutions to sparse linear systems. It is of interest to us due to its central role in decoding compressively sensed signals [2]. It is compute-bound, requires mostly simple matrix operations, and for many applications of interest in compressive sensing, operates on a small amount of input data that can be stored entirely in GPU global memory. Intuitively, this is an ideal case for GPU acceleration.

Our paper proceeds as follows: we briefly review the latest nVidia Fermi GPU architecture and CUDA, its computing platform and programming model. We present conjugate gradient, its applications, and our strategy for acceleration using the GPU. We show how

This material is in part based on research sponsored by Air Force Research Laboratory under agreement number FA8750-10-2-0180. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

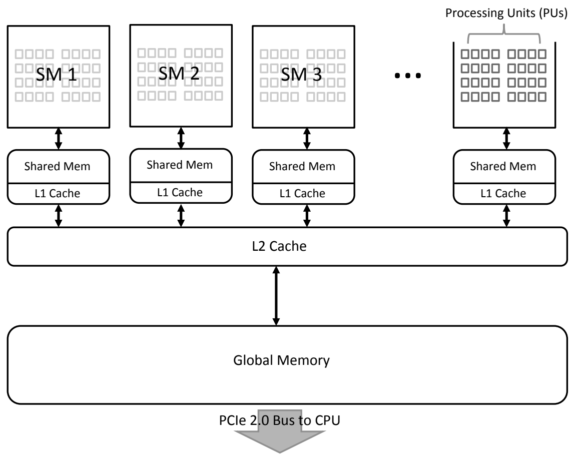

Figure 1: A simplified depiction of the nVidia Fermi architecture. SMs have limited on-chip memory shared among their PUs. Global memory is visible to all SMs, but while its aggregate bandwidth is high (152.0 GB/sec on the 570 GTX), round trip access latency from PUs is also high, and can vary between 400-600 clock cycles [3].

the lack of support for direct inter-SM communication causes costs associated with synchronization and operand aggregation to dominate any multi-SM adaptation of conjugate gradient. We then demonstrate gains on potential future GPUs with 1D SM interconnections. Finally, we survey the current state of compilers for parallel architectures, arguing that standard optimization techniques such as loop unrolling are readily amenable to GPUs with such interconnected SMs.

2 Background

2.1 GPU Hardware Overview

Using the nVidia GTX 570 as an example, we first briefly describe the Fermi GPU architecture, sketched in Figure 1. The GTX 570 is comprised of 15 independent Stream Multiprocessors (SMs), each an SIMD multiprocessor with 32 PUs operating at 1.4 GHz. The primary on-chip memory for an SM is a 64 KB shared memory pool, visible to its PUs, and organized into 32 banks so PUs may access memory in parallel.

All SMs are connected to a 1.2 GB global memory store. Aggregate bandwidth between the SMs and global memory is 152 GB/s, and the round trip access latency varies between 400-600 clock cycles from PUs [3]. The GPU also features texture and constant memory units with lower latency to SMs. However, since their contents are managed by the host program on the CPU, we eschew their use to minimize CPU intervention during program execution, which is extremely costly due to high latency.

2.2 CUDA Overview

The above system is exposed to application programmers through nVidia's CUDA language and compiler [3]. Data parallel functions are coded in CUDA's C-like syntax, and are termed *kernels*. Upon launch, the GPU replicates a kernel's instructions into multiple data independent *blocks*, and assigns them to SMs. At each SIMD SM, kernel instructions execute simultaneously on the PUs, forming parallel execution *threads*. SMs are generally overscheduled with multiple batches of threads, called *warps*; this allows I/O latencies to be hidden by swapping out warps whose threads are waiting on operands.

In CUDA, execution synchronization is supported only at the lowest level, for threads. When branching and I/O serialization cause threads to fall out of step, parallelism is reduced, so CUDA's `__syncthreads()` primitive restores proper execution order. In order for the GPU to schedule and load balance kernel execution over SMs, blocks are assumed independent, and so no synchronization is supported among SMs. However, the host-CPU may launch multiple kernels at once to pipeline launch overhead, and a *stream* identifier is provided to enforce kernel execution order; kernels launched into the same stream will execute in serial, while those in different streams may execute asynchronously.

Throughout, global memory is the only persistent resource that is visible to all threads and all blocks. A program on the host CPU is responsible for managing data transfer to and from global memory, as well as kernel launches.

2.3 Conjugate Gradient Algorithm and Applications

The conjugate gradient (CG) computation on normal equations, our target for acceleration, is an efficient and numerically stable algorithm that can be used to solve least squares problems [4]. Its steps are shown in Algorithm 1. CG forms its least-squares estimation by iteratively refining search direction vectors and updating a residual vector. The most compute intensive steps are matrix-vector multiplications (steps 3 and 7). The remaining steps are a series of simple vector additions and scalar multiplications. However, the scalar multipliers α and β are computed from vector norms of the results of steps 3 and 7. We will see that, if we accelerate the matrix vector multiplications by parallelizing across SMs, aggregating their results to calculate norms will be costly.

Conjugate gradient is a widely-applied optimization algorithm in many communities, including scientific computing, signal and image processing, and machine learning. It is of particular interest to us due to its role in reconstructing compressively sensed signals [2]. In

Algorithm 1 Conjugate Gradient on Normal Equations

```
Input: vector $y$, matrix $A$, initial approximation vector $x_0$
Output: vector $x_i$
 1: $r_0 = y - Ax_0$;  $s_0 = p_0 = A^T r_0$;  $\gamma_0 = \|s_0\|_2^2$
 2: while $\gamma_i > tol$ do
 3:     $q_i = Ap_i$
 4:     $\alpha = \gamma_i / \|q_i\|_2^2$
 5:     $x_{i+1} = x_i + \alpha p_i$
 6:     $r_{i+1} = r_i - \alpha q_i$
 7:     $s_{i+1} = A^T r_{i+1}$
 8:     $\gamma_{i+1} = \|s_{i+1}\|_2^2$
 9:     $\beta = \gamma_{i+1} / \gamma_i$
10:     $p_{i+1} = s_{i+1} + \beta p_i$
11: end while
```

compressive sensing applications, such as those in medical imaging, security, spectrum sensing, and wireless sensor networks, faster signal reconstruction translates directly into application-level gains [5][6][7]. For example, in CS-based spectrum sensing, faster decoding leads directly to the ability to scan a wider frequency band within a given delay budget [8]. GPUs are a potentially cost-effective, portable accelerator that would make an immediate impact for these applications.

2.4 Dense Linear Algebra on GPUs

Currently, dense linear algebraic computation is supported on the GPU primarily by CUBLAS, a package maintained by nVidia to incorporate improvements from the research community. It includes routines for matrix and vector operations that can be decomposed into independent blocks for batched execution over multiple SMs, but it relies on CPU coordination when results must be aggregated [9]. The implementation of CG provided with CUBLAS uses these routines to implement Algorithm 1, and stages the iteration loop on the CPU [10].

CUBLAS's routines are intended as generic building blocks for linear algebra-heavy algorithms, at the expense of sub-optimal runtime. For example, since CPU-to-GPU communication can exhibit up to 11 μs latency, coordinating on the host machine provides programming flexibility, but introduces large delays. As recently acknowledged by other authors, achieving best performance on the GPU in general requires minimizing CPU intervention [11].

3 Accelerating CG with Multiple SMs

As discussed in Section 2.3, solving a least squares problem using CG requires many iterations of the loop in Algorithm 1, whose most compute-intensive steps (3 and 7) are matrix-vector multiplications. Fortunately, these are parallelizable, and our strategy for acceleration on the GPU is to decompose A and A^T row-wise, and group the resulting vector multiplications into different blocks. The results must then be aggregated to compute scalar multipliers in steps 4 and 8, as depicted in Figure 2.

In order to show how implementing CG on the GPU leads to a significant I/O bottleneck, we first model the idealized I/O cost of Algorithm 1. We examine the total number of operands that must be loaded from global memory into SM shared memory for access by PUs. This ignores the upload cost from the CPU, which is fixed and independent of GPU architecture. We also assume that the cost of fetching operands into PUs from SM shared memory is negligible. The I/O cost is then:

$$T_{I/O} = [(2MN + N) + \lambda(2N + 2M)]/g \qquad (1)$$

Since A and A^T are reused in successive iterations, we load A, A^T, and x_0 once. This accounts for $2MN + N$ total operands when A is $M \times N$. Then, in each iteration, we store computed vectors s and p to global memory and subsequently refresh them. Over λ iterations, this accounts for $\lambda(2N + 2M)$ operands. In Equation 1, g is the pipeline rate between global and shared memories.

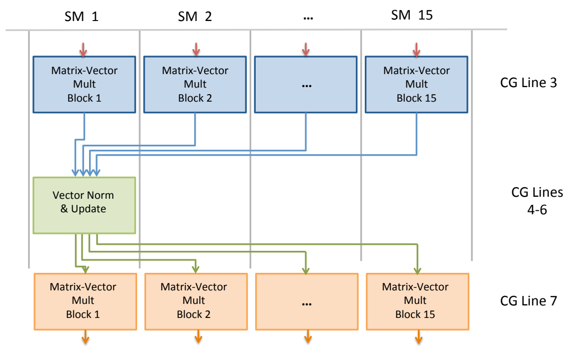
SM 1
SM 2
...
SM 15
Matrix-Vector Mult Block 1
Matrix-Vector Mult Block 2
...
Matrix-Vector Mult Block 15
CG Line 3
Vector Norm & Update
CG Lines 4-6
Matrix-Vector Mult Block 1
Matrix-Vector Mult Block 2
...
Matrix-Vector Mult Block 15
CG Line 7

Figure 2: A flow diagram showing the data dependency of steps in CG. Matrix-vector multiplications are parallelizable, but vector norms require aggregating results, leading to a need for barrier synchronization and operand exchange.

3.1 CPU Staging

Since CUDA does not support inter-SM synchronization, iterative algorithms must be implemented by launching subsequent iterations as ordered kernels in a common stream, the strategy taken by CUBLAS. This means that the contents of caches and shared memory are flushed and reloaded for every kernel, and reusable operands are not persistent across iterations. For CG, we incorporate the cost of reloading A and A^T into Equation 1, and the penalty becomes apparent:

$$T'_{I/O} = [(2MN + N) + \lambda(2MN + 2N + 2M)]/g \qquad (2)$$

The I/O cost given by Equation 2 is 18x worse when $M = 128$, $N = 256$, and $\lambda = 20$. Emprically, $\lambda = 20$ iterations are often required for CG convergence in this setting, though at $\lambda = 10$, the I/O cost is still 9x higher than the ideal case.

3.2 Global Memory Barrier

For reusable operands to be persistent, we must unroll CG into a single kernel and implement a barrier for synchronization and operand exchange within the lifetime of a block. We first do this using global memory, the only resource for communication among SMs without CPU involvement in the current architecture. Using a simple kernel that evalulates $A^{\lambda}u$ over λ iterations, we then study kernel performance. Structurally, this kernel is similar to CG in that elements of A may be reused, and that the most recently updated u must be aggregated at every iteration. For the results presented, A is a 384×384 element matrix of single-precision floats, the largest that can be accomodated by the current shared memory size.

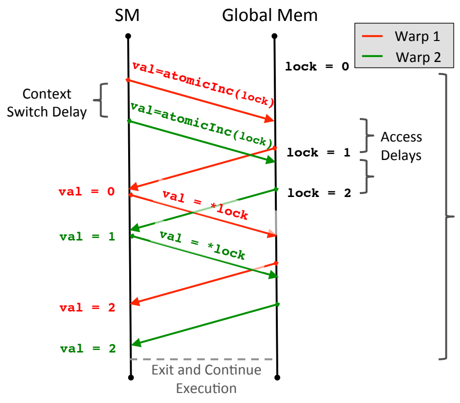

Figure 3: The operation of our software barrier at a single SM with two warps. The lock incurs two sources of delays: context switching delay, and atomic access delay. Note that CUDA's `atomicInc()` function returns the lock's value prior to its value being changed.

Our barrier works as follows: after guaranteeing the visibility of interim results to other SMs, we enter the barrier and use CUDA's `atomicInc()` function to atomically increment a counter in global memory, busy waiting while checking the counter value. Currently, CUDA supports a `__threadfence()` primitive that guarantees the visibility of memory updates independently by warp, therefore each warp must check in at the barrier. This is important because as the number of warps is tuned to optimize I/O throughput, we will show that relying on global memory for synchronization will result in an exorbitant performance penalty for large numbers of warps. Figure 3 illustrates barrier operation at an SM scheduled with two warps.

If we examine the operand I/O of our test kernel, we see that it is sharply reduced when compared to a corresponding CPU-staged implementation, as was our goal. For both methods, A is loaded in 5888 cycles on average, close to the theoretical lower bound. The vector u is loaded in an average of 688 cycles, which is dominated by the 600 cycle latency penalty to global memory. The total observed operand I/O is reduced from 131,520 to 19,648 cycles using the global memory barrier. These results are summarized in Figure 6. Extrapolating to CG, which requires two matrix loads, and two vector loads per iteration, we project a 7x reduction in operand I/O time, a reduction that can be further improved by reducing the latency of operand exchanges.

However, the overall runtime using the global memory barrier is worse, underscoring the poor performance of synchronizing via global memory. As illustrated in Figure 3, there are two sources of delay in the barrier that affect runtime: context switching time between warps, and access delay to the lock. These delays compound as a function of the number of warps, as shown in Figure 4. For example, with a single warp at each SM, the barrier delay averages 1250 cycles, or only two round-trip times to global memory, as expected. However, with eight warps, the barrier delay is already 10,000 cycles. Since our test kernel requires 32 warps to achieve maximum I/O throughput, the resulting runtime using the synchronization barrier is an order of magnitude worse: 1.4×10^6 cycles *vs.* 2.8×10^5 cycles for the CPU-staged kernel, due to high barrier overhead.

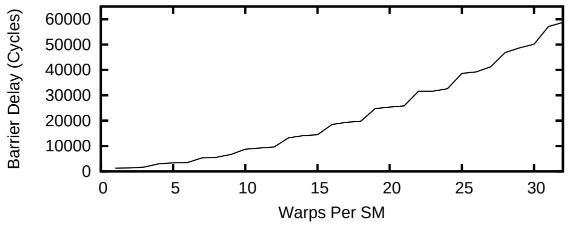

Figure 4: The number of cycles spent in the lock at each SM as a function of the number of warps. As warps are added to pipeline I/O, the software barrier becomes more costly, overwhelming I/O cost at only eight warps.

3.3 Hardware-Supported Neighbor Synchronization

Low latency synchronization and operand exchange can be achieved with relatively simple neighbor connections between SMs, allowing us to realize runtime speedups by reducing operand I/O. While fast all-to-all SM communication is expensive and does not scale well with the number of SMs, connections between neighbor SMs are

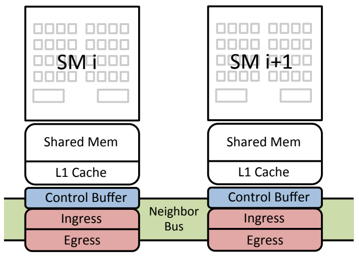

Figure 5: Our proposed 1D SM-interconnection architecture adds only inexpensive connections between adjacent SMs and the appropriate buffers to support operand exchange.

relatively cheap and scale easily. Furthermore, any all-to-all communication using a centralized memory unit must ultimately service one request at a time, causing access delay to vary signficantly, especially under high load. This effect is apparent in our global memory barrier. However, with simple neighbor communication, local connections serve only two SMs, and many connections can operate in parallel.

Our proposed architecture is shown in Figure 5. In it, we make the following assumptions: each pair of adjacent SMs is connected by a full duplex local bus with $152 GB/s$ bandwidth, equal to that of the global memory bus; ingress and egress buffers are added for data exchange with dedicated slots to store 32 single precision floating point operands from each SM; an additional buffer is added to store control flags; and latency between SMs is on the order of 10 cycles, which is half of the observed latency between PUs and the current L2 cache. In this design, only operand buffers need to be scaled when more SMs are added, since we assume they are large enough to hold operands from all SMs, in order to propagate operand updates with a single transfer.

To emulate the synchronization and exchange process when only neighbor communication is possible, we similarly implement this barrier using buffers in global memory, and augment the busy-wait loop to propagate control flags and data to neighbors. We then plug in the previous assumptions to estimate the expected performance. Traversing the linear topology end-to-end requires 14 transfers, and transferring the contents of the data and control buffers takes 17 cycles, leading to a total synchronization and operand exchange time of only 240 cycles per iteration. As shown in Figure 6, the resulting implementation of our test kernel would have a 15x reduction in total I/O over the CPU-staged method. When we extrapolate to CG, we achieve a similar reduction in I/O of 15x. At the dimensions used in our test kernel, CG's lower bound for computation is roughly 26,000 cycles meaning that the implementation is now compute bound and could be sped up with additional computational resources.

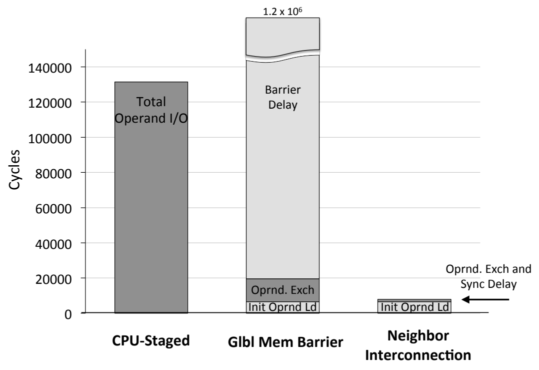

Figure 6: The contribution of synchronization and data exchange costs to total runtime for our test kernel

4 Implications for Future GPUs

We have shown that fast synchronization and operand exchange between SMs are important in order to properly leverage loop unrolling in iterative algorithms. Without such support, current GPU architectures from both nVidia and ATI/AMD induce heavy operand I/O, reducing compute-to-I/O ratio, and therefore available parallel processing opportunities. As future generations of GPUs scale computational resources, inter-SM communication would alleviate these I/O limitations at low hardware cost. For a large class of iterative parallelizable algorithms, these steps are necessary to achieve the benefit of additional computational power.

With neighbor interconnections already a feature of some other parallel processors (e.g. the Cell Broadband Engine [12], and [13]), efficient use of loop unrolling has been a standard compiler optimization in the literature. Discussions of this capability include [14] and [15]. It is therefore reasonable to expect that this optimization can be readily integrated into the CUDA compiler.

5 Conclusion

GPUs are an attractive many-core platform due to their cost, availability, and scalable architecture; however without low-latency hardware-supported intercommunication between SMs, they are ineffective for accelerating a large class of iterative parallelizable algorithms. Given the capabilities of existing compilers and common parallel processors, and the myriad waiting applications such as those relying upon high-performance compressive sensing decoding, the GPU architecture itself is currently the biggest bottleneck to these applications. We have shown that simply adding cheap neighbor interconnections will lead to significant gains and allow general purpose GPU computing to accelerate similar matrix computations.

References

[1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[2] D. Needell and J. Tropp, "CoSaMP: Iterative signal recovery from incomplete and inaccurate samples," *Applied and Computational Harmonic Analysis*, vol. 26, no. 3, pp. 301–321, 2009.

[3] "nVidia CUDA Programming Guide 4.0,," 2011. http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[4] Å. Björck, *Numerical methods for least squares problems*. No. 51, Society for Industrial Mathematics, 1996.

[5] Y. Gwon, H. Kung, and D. Vlah, "DISTROY: detecting integrated circuit Trojans with compressive measurements," in *6th USENIX Workshop on Hot Topics in Security (HotSec)*, 2011.

[6] M. Lustig, D. Donoho, and J. Pauly, "Sparse MRI: The application of compressed sensing for rapid MR imaging," *Magnetic Resonance in Medicine*, vol. 58, no. 6, pp. 1182–1195, 2007.

[7] Z. Tian and G. Giannakis, "Compressed sensing for wideband cognitive radios," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 4, pp. IV–1357, Ieee, 2007.

[8] Y. Gwon and H. T. Kung and D. Vlah, "Compressive sensing with directly recoverable optimal basis and applications in spectrum sensing," in *Harvard University Technical Report TR-08-11*, Harvard Univerisity School of Engineering and Applied Science, 2011.

[9] "nVidia CUBLAS Library User Guide," 2012. http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[10] "nVidia CUDA Libraries SDK Code Samples - Precondition Conjugate Gradient," 2012. http://developer.nvidia.com/cuda-libraries-sdk-code-samples.

[11] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–11, IEEE, 2008.

[12] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementationa performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[13] C. Cascaval, J. Castanos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. Moreira, K. Strauss, and H. Warren Jr, "Evaluation of a multithreaded architecture for cellular computing," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pp. 311–321, IEEE, 2002.

[14] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 151–162, ACM, 2006.

[15] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*, ch. Spiral. Springer, 2011.