A Memory Optimization Technique for Software-Managed Scratchpad Memory in GPUs

Maryam Moazeni
Computer Science Department
University of California,
Los Angeles
mmoazeni@cs.ucla.edu

Alex Bui
Department of Radiological Sciences
University of California,
Los Angeles
buia@mii.ucla.edu

Majid Sarrafzadeh
Computer Science Department
University of California,
Los Angeles
majid@cs.ucla.edu

Abstract—**With the appearance of massively parallel and inexpensive platforms such as the G80 generation of NVIDIA GPUs, more real-life applications will be designed or ported to these platforms. This requires structured transformation methods that remove existing application bottlenecks in these platforms. Balancing the usage of on-chip resources, used for improving the application performance, in these platforms is often non-intuitive and some applications will run into resource limits. In this paper, we present a memory optimization technique for the software-managed scratchpad memory in the G80 architecture to alleviate the constraints of using the scratchpad memory. We propose a memory optimization scheme that minimizes the usage of memory space by discovering the chances of memory reuse with the goal of maximizing the application performance. Our solution is based on graph coloring. We evaluated our memory optimization scheme by a set of experiments on an image processing benchmark suite in medical imaging domain using NVIDIA Quadro FX 5600 and CUDA. Implementations based on our proposed memory optimization scheme showed up to 37% decrease in execution time comparing to their naïve GPU implementations.**

Keywords-GPU Computing, Memory Optimization, CUDA

I. INTRODUCTION

Traditional single-core microprocessors have driven rapid performance increase in two decades. However, constraints on power consumption slowed this progress, which have forced CPU vendors to find other ways to meet high performance computing needs in science and engineering. One solution is that of multi-core architectures, which are moving towards integrating tens or hundreds of cores onto a single chip – termed as many-core. Many-core processors can offer higher performance or power efficiency compared to current CPU or multi-core processors [1].

One example of commodity many-core processors is the current programmable graphics processing units (GPUs) such as AMDR580 with CTM [2] as their compute runtime driver or NVIDIA G80 GPU's with CUDA [3] as their programming model, and the future Intel Larrabee [4]. Current GPUs have hundreds of processor cores and high memory bandwidth. For example, G80 consists of 16 *Streaming Multiprocessors* (SMs), each with eight *Streaming Processors* (SPs), 8096 registers, and 16 KB of on-chip memory per SM. The architecture allows efficient data sharing and synchronization among threads in the same thread block [5]. More comprehensive descriptions of the NVIDIA architecture are found in [3, 5, 16].

Processing power of GPUs has been successfully exploited in the GPGPU domain, especially in scientific, imaging and database applications. However, increasing the application performance in such architectures is not a trivial task.

Modern high-performance computer architectures have increasing number of on-chip processing elements. Architects must ensure that memory bandwidth and latency are also optimized to exploit the full benefits of the available computational resources. Utilizing cache hierarchy has been the traditional way to alleviate the memory bottleneck [6]. In contrast, various modern parallel architectures such as NVIDIA G80 [5] and IBM Cell [28] utilize fast explicitly managed on-chip memories, often referred to as *scratchpad* memories, in addition to slower off-chip memory in the system to hide the memory latencies [6]. Scratchpad memories are limited in size since minimization of on-chip memories is important in reduction of manufacturing cost [7].

The introduction of the IBM Cell processor with software-managed per-core memory (local store) led to the development of techniques for utilizing that memory. However, because of the architectural differences between Cell processor and NVIDIA G80, management of the software-managed on-chip memory (shared memory) in NVIDIA G80 architecture has to be specifically studied, and the effect of imposed overheads has to be evaluated based on the architectural organization of G80. In the NVIDIA G80 architecture, shared memory is partitioned among up to 512 thread blocks that are assigned to the same multiprocessor at run-time. The data in shared memory can be shared among all threads in a thread block, enabling inter-thread data reuse. This is in contrast to single thread access to Cell's local store. Moreover, in G80, an incremental increase in the usage of shared memory per thread can result in a substantial decrease in the number of threads that can be simultaneously executed and thus significantly reducing the parallelism. Current G80 architecture offers limited resources (e.g. shared memory) available to each multiprocessor, and conversely, demand for availability of massive number of threads to achieve maximum performance. The limited size of fast-access shared memory available to each multiprocessor and its considerable impact on reducing the parallelism motivates us to develop a method to minimize the usage of shared on-chip memory space in G80. This method should specifically be designed for the properties of the shared memory within the G80 architecture.

Main results: In response to this challenge, we propose a memory optimization method, which assists in increasing

978-1-4244-4938-5/09/$25.00©2009 IEEE

parallelism in applications with high data dependencies by minimizing the usage of shared on-chip memory (scratchpad memory) and increasing each multiprocessor's utilization (occupancy) in the G80 architecture. We conducted a set of experiments on our image processing benchmark suite in medical imaging domain as a source for real-life and data-intensive applications.

The work we presented in this paper demonstrates the promising result of having such an optimization scheme in the NVIDIA G80 architecture, based on experimental results on our image processing benchmark suite consisting of real-life and data-intensive applications. Our intent is to form the proposed memory optimization technique as automatic transformations for this platform.

Organization: The remainder of this paper is organized as follows. We first discuss related work in Section II. In Section III, we present our motivations for memory optimization for scratchpad memory in the NVIDIA G80 architecture, and propose our solution for the memory optimization scheme. Then we evaluate our method by a set of experiments on an image processing benchmark suite in medical imaging domain in Section IV. Finally, we present some concluding remarks and suggestions for future work in Section V.

II. RELATED WORK

Historically, a motivation for the development of data-parallel languages is strongly related with Single Instruction Multiple Data (SIMD) computer architectures. Data-parallel languages such as OpenMP [8] are explicit parallel programming model to support parallel computing. Fortran 90 [9] was one of the most widely used data-parallel languages that provide constructs for specifying concurrent execution based on data-parallelism. HPF [10] extends FORTRAN 90 with additional parallel constructs and data placement directives. HPF was introduced as a standard data-parallel language to support programs with SPMD. Message passing libraries such as MPI [11] also became a popular programming model for scalable parallel systems. Intel Ct [12] is a programming model developed by Intel to ease the exploitation of its future multi-core chips. It is based on the exploitation of SIMD to produce automatically parallelized programs.

The interest in GPU programming for general-purpose computations has been driven by relatively recent improvements in the programmability and flexibility of graphics hardware. CUDA [3] from NVIDIA, by increasing the flexibility and programmability of GPU's has improved their suitability for high-performance computing. A programming interface alternative to CUDA is available for AMD Stream Processor, using the R580 GPU, in the form of the Close to Metal (CTM) [2] compute runtime driver which, completely exposes ISA to the programmer; thus providing fine-grained control. Intel's C for Heterogeneous Integration (CHI) programming environment [13] extends the OpenMP pragma for heterogeneous multithreading programming that tightly couples specialized accelerator cores with general-purpose CPU cores.

The challenges in the GPGPU community have revolved around the constraints of the programming environment and optimal mapping of applications so to best leverage the highly parallel GPU architecture. There have been several attempts in introducing structured methods and models for optimizing applications for maximum performance in this domain. Ryoo *et al*, present metrics to judge the performance of an optimization configuration based on first-factors of performance [14]. In this work, these optimization metrics are used to prune many optimization configurations; and therefore reducing the optimization space up to 98% without missing the configuration yielding best performance. Govindaraju *et al*, present a memory model in [15] for analyzing and improving the performance of scientific algorithms on graphic processors. Their model is based on texturing hardware and incorporates several characteristics of GPU architectures. There has been few comprehensive performance studies conducted in [16, 17]. [16] studied the performance of a broad range of applications and presents general principles for optimizing applications for this type of architecture. [17] also provides an application study of diverse applications and discusses advantages and inefficiencies of the CUDA programming model and some desirable features that might allow for more readily support a larger body of applications.

There have been several structures proposed in different programming languages deploying memory reuse approaches. The approach in [18] uses array transformations on data and shows that constraints on memory allocation functions can be formulated as one or more ILP problems. Our memory optimization technique is most similar to that of [7] that uses pointer analysis and graph coloring for discovering the chances of memory sharing, incorporated in a behavioral synthesis tool that synthesizes sequential C programs. However, the focus of this work is only sequential programs. In [30] a scratchpad overlay technique for low power embedded processors is presented which analyzes the application and inserts instructions to dynamically copy both variables and code segments onto the scratchpad at runtime. This problem is also an extension of the global register allocation problem. Additionally, graph coloring has also been used for optimizing utilization of stream register files in stream processors [19]. Our memory optimization technique is designed particularly for an architecture in which scratchpad memories are shared among a large number threads, which require us to adopt a reuse pattern to reuse the data that is shared among all threads in a thread block. In general, our memory optimization technique employs similar approaches from register allocation domain [20] that have been previously studied and proved to be practical.

III. MEMORY OPTIMIZATION

Global memory bandwidth can limit the throughput of the system as described in Section 1. In G80, alleviating the pressure on global memory bandwidth generally involves using additional registers and shared memory to reuse data, which in turn can limit the number of simultaneously executing threads. Balancing the usage of these resources is often non-intuitive and some applications will run into resource limits. This section presents a memory optimization technique in the G80 architecture to further alleviate the constraints of using shared memory for data-intensive applications in the G80 architecture.

Data-intensive applications that have high usage of shared memory in their CUDA implementations are limited by low SM occupancy when ported to the G80 architecture. In the G80, as each thread's resource usage (e.g. shared memory and register count) increases, the total number of threads that can occupy the SM decreases, which results in reduction of SM occupancy that results in significant performance loss. Occasionally this decrease in thread count occurs in a dramatic fashion because threads are assigned to an SM at the granularity of thread blocks; this makes the situation very critical in a sense that a small increase in thread's resources could have a dramatic effect on performance.

For example, consider a data-intensive application with 128 threads per block and 8KB of shared memory per thread block. This application can schedule 2 thread blocks on each SM. However, if each thread's shared memory usage increases from 8KB to 10KB (an increase of 25%), the number of blocks per SM will decrease from 2 to 1 (a 50% decrease). In other words, the G80 can only assign one thread block (128 threads) to an SM because a second block would increase the amount of shared memory usage above the SM limit. This results in significant performance reduction. Therefore, allocating memory space in limited scratchpad-like memories in such data-intensive applications is highly costly in modern parallel architectures such as G80.

In order to maximize the performance, it is better to allow for two or more thread blocks to simultaneously execute. For this to happen, not only should there be at least twice as many thread blocks as there are multiprocessors in the device, but also the amount of allocated shared memory per thread block should be at most half the total amount of shared memory available per multiprocessor [3]. Therefore, it is crucial to have a mechanism to minimize the usage of shared memory. We are aiming at achieving this by reusing allocated memory spaces and avoiding the allocation of further unnecessary resources for each thread block with the goal of maximizing the performance. In our vision, by having this transformation, developers will provide a straightforward implementation of the kernel code that utilizes shared memory, and depend on this transformation to optimize the memory usage.

In the following section, we propose a *memory reuse scheme* particularly designed for scratchpad memory in GPU architectures. In Section IV, we demonstrated the effectiveness of our approach on our image processing benchmark suite.

A. Memory Reuse Scheme

The architectural constraints of shared memory in the G80 architecture was described in Section I and III. In this section, we present the rationale behind our proposed memory minimization approach.

Consider a motivational simple example of the shared memory reuse in Figure 2(a), where memory blocks `sA`, `sB` and `sC` are shared among all threads in a thread block, and need to be allocated to certain memory areas in shared memory. A naive allocation, as performed by almost all the software compilers, is to map each of the blocks to distinct memory locations, as shown in Figure 2(b). A careful inspection of the program reveals that memory block `sA` and memory block `sC` can in fact be shared, leading to the allocation in Figure 2(d), which can be obtained by the modified program in Figure 2(c). We refer to the `sA_shared_sC` memory block as the "reused memory block" in our scheme.

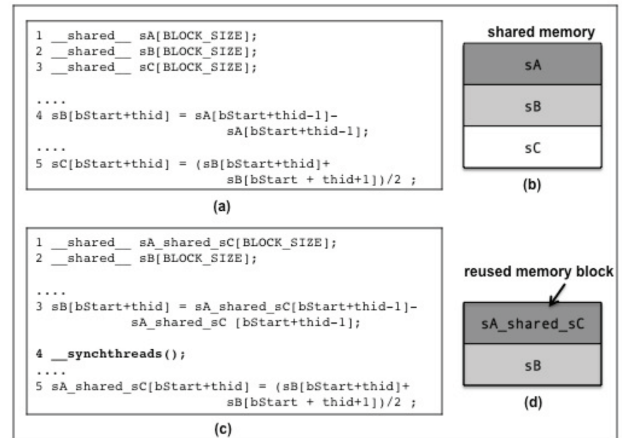

Fig. 1. A Motivational Example in CUDA

Our ultimate goal in the memory reuse scheme is to minimize the usage of memory space without changing the structure of the program. One might argue that the programmers should identify such opportunities of memory reuse and enforce them manually in the program. We believe this requirement is unrealistic for the following reasons: (1) the primary goal of a programmer is to specify functionality; for a programmer, readability and maintainability has higher priority than implementation details; (2) as the application complexity increases (i.e. consisting of data structures with different sizes) automated optimization tools have a better chance to find an optimal solution than the programmers; (3) in a multithreaded context it is harder for the programmer to enforce the memory sharing while maintaining the correctness of the program; (4) eventually productivity of programmers will increase by taking the burden of memory management off their shoulder.

The idea of having a memory reuse scheme is very similar to the register allocation problem in traditional compiler optimization [20]. The goal in the memory reuse problem is to achieve memory minimization by discovering the chances of memory reuse with the goal of maximizing the application performance. We propose a solution for the memory reuse problem based on graph coloring described in the following section.

B. Solution Approach

1) Reuse Pattern

As described in previous sections, our goal is to achieve memory minimization by discovering the chances of memory reuse. Since our solution is proposed for optimization in the GPU shared memory space, the desired reuse pattern in applications should be suited to the architecture of GPUs and shared memory in particular.

In the G80 architecture, shared memory is shared among all threads in a thread block and we intend to leverage a reuse pattern to reuse shared memory spaces across all threads in the thread block as illustrated in the example of Figure 2. Therefore, execution of all threads in the thread block needs to be synchronized to coordinate shared memory accesses to provide means of correct and safe memory reuse, as illustrated by use of *__synchthreads* primitive in Figure 2(c), line 4. This is due to the fact that each active thread block on a multiprocessor is split into SIMD groups of threads (warps) executed in an SIMD fashion, and all the SIMD groups from all active thread blocks on the multiprocessor are time-sliced. Therefore, there is no explicit guaranteed ordering in the accesses to shared memory in different SIMD groups in a thread block. As a result, in order to make memory reuse a viable solution in such SIMD architecture, it is crucial to enforce synchronization after or before the *points of reuse*. We define *points of reuse* as any use or definition point of a "reused memory block" in the program. For example lines 3 and 5 in Figure 2(c).

Previous methods [6, 19], discussed in the Section II, were not designed for scratchpad memories that are shared among i.e. 512 threads; therefore, synchronization of threads for coordinating the accesses to shared memory was not an issue in those studies. In our problem, memory blocks are shared among all threads in the thread block; thus, coordination of memory accesses according to the underlying threading model is critical, and needs to be explicitly added to the kernel code at points of reuse – an example of this case is demonstrated in Figure 2(c). In section IV, we evaluate the effect of synchronization overhead on the overall performance results.

2) Discovering the Chances of Memory Reuse

Regardless of the desired reuse pattern in our scheme, we describe our solution to discovering the chances of memory reuse in this section.

In our proposed solution, we define a *memory block* to be an arbitrary sized array of data shared among all threads in a thread block. A *memory partition* is a partition of shared memory to which one or more memory blocks will be assigned. Figure 3 demonstrates our memory reuse scheme. Figure 3(a) shows the placement of memory blocks without the memory reuse scheme; Figure 3(b) depicts the live range conflicts between memory blocks $B_1..B_5$ in the given interference graph in which two nodes each representing memory blocks are connected if their live ranges overlap. Figure 3(c) demonstrates the reduction in memory space in each thread block by leveraging the memory reuse scheme based on the given interference graph. As an example, there is no edge between memory block B_4 and B_1 in the interference graph, and hence, B_4 and B_1 are both assigned to memory partition P_1. As it is shown in the figure, B_4 is placed inside B_1's memory space in order to reuse available memory spaces. It should be noted that memory partitions should not necessarily be of the same dimension.

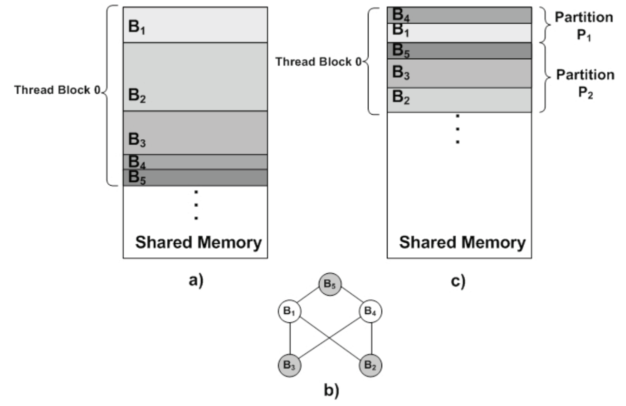

Fig. 2. A Memory Reuse Scheme for Shared Memory

The difference between the proposed memory reuse technique against the well-known register allocation problem is that: inputs to the memory reuse problem are arbitrary sized memory blocks. This makes the problem different than register allocation in the sense that finding the optimal sizes of memory partitions are now added to the problem, which makes it more of a placement problem as seen in the bin-packing problem [21].

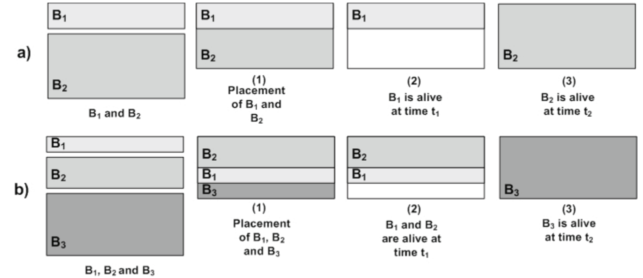

Fig. 3. Configuration of memory blocks B_i in their memory partitions

There are two possible configurations for the placement of memory blocks in memory partitions in the memory reuse scheme. In the first configuration shown in Figure 4(a), only one of the memory blocks in a memory partition is alive at a time. In Figure 4(a) memory blocks B_1 and B_2 are sharing a memory partition where B_1 is placed inside B_2 space; B_1 is showed to be alive at time t_1 and B_2 is alive at time t_2. In the second configuration, it is possible to have more than one memory blocks that are simultaneously alive in a memory partition as depicted in Figure 4(b) in which B_1 and B_2 are alive at time t_1. In this placement B_1 and B_2 are both placed with no physical overlap inside B_3 space. Therefore, in this configuration B_1 and B_2 may have conflict in their lifetimes, but neither should have conflicts with B_3.

In order to relax the problem, we solve the problem assuming only one memory block b_i from memory partition P_j can be alive at a time. Therefore, our expected configuration for the relaxed problem is as illustrated in Figure 4(a), which our proposed algorithm is based upon. Thus, two memory blocks that are simultaneously alive cannot share a partition. Therefore, our ultimate goal is to assign memory blocks with non-overlapping lifetimes to memory partitions so that usage of

memory space is minimized without changing the structure of the program. Given a program with arbitrary sized memory blocks b_i, our goal is to allocate memory partitions P_j in shared memory to fit as many memory blocks with non-overlapping lifetimes as possible in P_j.

Algorithm1 Memory Reuse

Input:

- $\{b_1, b_2, \ldots, b_n\}$: existing allocated memory blocks in shared memory

Output:

- K: number of memory partitions to be created
- $\{P_1, P_2, \ldots, P_k\}$: memory partitions to be allocated and their related meta-data

1: Determine the live ranges of all memory blocks b_i
2: Build the Interference Graph $G(V, E)$
  - $V=\{b_1, b_2, \ldots, b_n\}$
  - Undirected edge connects two memory blocks (b_i, b_j), if live ranges of b_i and b_j overlap in time
3: $B' \leftarrow \{b_0\}$

4: **for all** nodes b_i adjacent to nodes in B' **do**
5: $B' \leftarrow B' \cup \{b_j\}$
6: Assign color K_j such that adjacent colored nodes has different colors
7: $K = Max(K_j, K)$
8: $P_j \leftarrow P_j \cup \{b_j\}$
9: $Partition_Size(P_j) = Max(Partition_Size(P_j), Block_Size(b_j))$
10: **end for**

11: Allocate $P = \{P_1, P_2, \ldots, P_k\}$ in shared memory {Physically assign members of P to the corresponding memory partition}

Algorithm 1 is proposed as a solution for memory reuse problem. In Step 1 of the algorithm, the live ranges of memory blocks are determined to identify memory blocks with non-overlapping lifetimes. Step 2, constructs an interference graph based on the output of Step 1 in which two nodes each representing memory blocks are connected if their live ranges overlap (Figure 3(b)). Steps 4-6 cluster the memory blocks with non-overlapping lifetimes by coloring the interference graph; memory blocks assigned to the same memory partition are represented by the same color after coloring the interference graph. By adding each color K_j, a new memory partition P_j is added to the solution, and memory blocks colored with color K_j are assigned to P_j in Step 8. In Step 7, number of memory partitions K is calculated. Size of memory partitions are calculated based on the maximum size of assigned memory blocks based on the configuration described in Figure 4(a) (Step 9).

Note that this algorithm, arbitrarily selects memory blocks to be added to the solution, and therefore, the final solution may not be optimal. However, we show the effectiveness of this algorithm through experimental evaluations. It is worth mentioning that finding the optimal size of memory partitions and placing memory blocks in a memory partition when having more than one live memory block per memory partition at a time is NP-Complete, which can be proved by reducing this problem to the bin-packing problem [21], which is not discussed in this study.

IV. EXPERIMENTAL EVALUATION

This section presents the experimental results of manual evaluation of our memory optimization scheme introduced in Section III. We based our experiments on our image processing benchmark suite as a source of real-life and data-intensive applications to demonstrate how real-life applications can benefit from the introduced optimization method. These real-life applications are more interesting and useful than microbenchmarks because of their larger code sizes and data sets, and variety of instructions and control flow [22].

We used CUDA version 2.0 for our experiments. Experiments were performed on Core2 Duo running at 2.33 GHz with 8 GB of main memory and NVIDIA Quadro FX 5600 as a commodity GPU.

A. Benchmarks

Our benchmark suite[1] consists of denoising, segmentation and registration algorithms particularly designed for medical imaging. In this section we describe the most important characteristics of the three benchmarks that are mostly relevant to our experimental evaluations of the memory reuse scheme. The denoising benchmark is a local nonlinear iterative denoising algorithm called Total Variation Regularization [23]; the segmentation benchmark is a curvature-based segmentation algorithm called Active Contour [24] based on geometric PDE's, and the registration benchmark is based on Biharmonic Regularization [25].

Measurement of *curvature* exists in both denoising and segmentation benchmarks, and has high usage of shared memory in its CUDA implementation; curvature is a common measurement in image processing and computer vision algorithms [23, 24, 26, 27]. The denoising benchmark uses a 3D computation of curvature (Curvature3D), and the segmentation benchmark uses a 2D calculation of the curvature (Curvature2D). Detailed explanation on the measurement of GPU-based Curvature3D is presented in [29]. We implement Curvature3D and Curvature2D as independent kernels.

The Curvature kernel consists of three measurements: (1) partial derivatives, (2) gradient norm and (3) divergence where measurements at each step have dependency to the previous measurements. For example, measurement of partial derivates is dependent on two neighboring pixel values; measurement of gradient norm is dependent on four neighboring pixels' corresponding partial derivates; measurement of the divergence is dependent on two neighboring pixels' corresponding gradient norms [23]. Therefore, there is significant data reuse;

[1] Benchmarks' code and data are available by emailing the authors.

thus, shared memory is utilized for storing these arrays. The pixel data is loaded from global memory collaboratively by each thread, where accesses to global memory are coalesced. It is important to note that the computation of Curvature3D is heavier than Curvature2D and involves higher synchronization overhead. The preferred thread block size is 16×16 for both implementations and the SM occupancy is 66%.

The registration benchmark consists of two computational steps, which update a "displacement" array in vertical and horizontal directions within each iteration; the two displacement arrays are placed in shared memory, as there is significant data reuse in computation of each pixels' displacement from displacement of neighboring pixels' in the previous iteration. The pixel data is loaded from texture memory in order to utilize the on-chip texture cache. The preferred thread block size is 16×24 in our implementation and the SM occupancy is 38%.

In all experiments, we consider each benchmark in its preferred configuration; for instance, in the denoising and the segmentation benchmarks, the thread block size is set to be 16×16, while in the registration benchmark, it is set to be 16×24.

B. Evaluation of the Memory Reuse Scheme

We performed a set of experiments applying our memory reuse scheme manually on our image processing benchmark suite. We compare the results of straightforward GPU-based implementations of the three benchmarks with our GPU-based implementations optimized for shared memory space based on our memory optimization technique.

TABLE I
SHARED MEMORY SAVING FOR THE IMAGE PROCESSING BENCHMARK

Benchmark	**Size w/o optimization (byte)**	**Size w/t optimization (byte)**	**Memory Savings**
Denoising	6220	3916	37%
Segmentation	4940	3084	37.5%
Registration	13596	13596	0%

Table I shows the memory savings we can achieve for the benchmark suite. The first column gives the shared memory usage per thread block without optimization. The second column gives the shared memory usage per thread block after applying the memory optimization. The third column gives the percentage of memory saving we are able to achieve per thread block. For denoising and segmentation benchmarks the achieved memory saving is 37% in the optimized implementations, allowing the number of active thread blocks to increase from 2 to 3 (increasing the number of active threads from 512 to 768) on each multiprocessor, which increases the multiprocessor occupancy from 66% to 100%. This increase in the number of active threads increases the parallelism, which results in increasing the performance.

In the registration benchmark we cannot achieve memory saving directly from this optimization technique. However, by changing the order of computations we can apply the memory reuse technique to this benchmark as well. In spite of this, we ignore this benchmark for optimization since reordering the computations is out of the scope of this paper. Nevertheless, it is worth pointing out the existing potential.

We observe that benchmarks that involve multiple steps of dependent processing benefit from our memory reuse technique to the most. For example, measuring the curvature constitutes the measurement of partial derivates, gradient norm – dependent on partial derivates – and finally, measurement of the divergence – dependent on gradient norm [23].

TABLE II
GPU EXECUTION TIME FOR DENOISING

Data size	**Exec time w/o optimization (μsec)**	**Exec time w/t optimization (μsec)**	**Percentage**
16^3	25.2	19.53	22.5%
32^3	74.89	60.96	18.6%
64^3	390.7	319.59	18.2%
128^3	3033.3	2532.85	16.5%

TABLE III
GPU EXECUTION TIME FOR SEGMENTATION

Data size	**Exec time w/o optimization (μsec)**	**Exec time w/t optimization (μsec)**	**Percentage**
64x64	35.3	22.14	37%
128x128	39	27.39	30%
256x256	70.5	61.2	13%
512x512	185	181.76	2%

Table II and Table III show the execution time before and after applying the optimization to denoising and segmentation benchmarks. In both tables, the first column gives the GPU execution time without optimization. The second column gives the GPU execution time after applying the memory optimization. The third column shows the reduction of GPU execution time in percentage. It is observed that by increase in data size, the performance increase is reduced. This demonstrates that high multiprocessor occupancy has less effect on performance as the load is increased on GPU. We observe that our optimization technique maintains more stable results particularly in the denoising benchmark with higher computational load and higher synchronization overhead compared to the segmentation benchmark. Particularly in the denoising benchmark, it is also observed that although there is a synchronization overhead involved in our optimization scheme, the overall performance results are promising.

V. CONCLUSION AND FUTURE WORK

In this work, we proposed a memory reuse scheme to minimize the usage of shared memory space in applications with high data dependencies and increasing the parallelism as a result. In the G80, alleviating the pressure on global memory bandwidth generally involves using additional registers and shared memory to reuse data, which in turn can limit the number of simultaneously executing threads, which significantly decreases the application performance. Balancing

the usage of these resources is often non-intuitive and some applications will run into resource limits. Therefore, our proposed memory optimization technique in the G80 architecture further alleviates the constraints of using shared memory for applications with high data dependencies in the G80 architecture. We evaluated our proposed memory optimization technique by a set of experiments on our image processing benchmark suite in medical imaging domain using NVIDIA Quadro FX 5600 and CUDA. Implementations based on our proposed memory reuse scheme showed up to 37% decrease in execution time comparing to their naïve GPU implementations.

Future work for our study is directed towards evaluating the memory reuse scheme on a broader range of applications. In this paper, we present the promising result of having such an optimization scheme in the G80 architecture, based on experimental results from our image processing benchmark suite. We aim at forming the memory reuse scheme as an automated transformation for this platform. Adding this memory optimization technique as an automatic transformation method will improve the productivity of developers by taking the burden of managing shared memory off their shoulders.

REFERENCES

[1] D. Manocha, M. C. Lin, N. Govindaraju. GPGPU to Many-Core Processing: Higher Performance for Mass Market Applications. Manycore Computing Workshop, 2007.

[2] AMD Stream Processor. http://ati.amd.com/products/streamprocessor/index.html

[3] NVIDIA Corporation. NVIDIA CUDA Programming Guide, version 1.1, 2007.

[4] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. 2008. Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph. 27, 3 (Aug. 2008).

[5] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum, May 2007.

[6] Kandemir, M. and Choudhary, A. 2002. Compiler-directed scratch pad memory hierarchy design and management. In Proceedings of the 39th Conference on Design Automation (New Orleans, Louisiana, USA, June 10 - 14, 2002). DAC '02. ACM, New York, NY, 628-633.

[7] Zhu, J. 2001. Static memory allocation by pointer analysis and coloring. In Proceedings of the Conference on Design, Automation and Test in Europe (Munich, Germany). W. Nebel and A. Jerraya, Eds. Design, Automation, and Test in Europe. IEEE Press, Piscataway, NJ, 785-790.

[8] OpenMP Architecture Review Board. OpenMP application program interface, May 2005.

[9] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 handbook: complete ANSI/ISO reference*. Intertext Publications, Inc.,/McGraw-Hill, Inc., 1992.

[10] D. B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Technology, 1(1):25–42, 1993.*

[11] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. HussLederman. *MPI: The Complete Reference*. MIT Press, 1995.

[12] ArchitecturesA Gholoum, E Sprangle, J Fang, G Wu, X Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale - Intel Whitepaper, October, 2007.

[13] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 156–166, 2007.

[14] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu, “Program optimization space pruning for a multithreaded gpu,” in CGO ’08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization. New York, NY, USA: ACM, 2008, pp. 195–204.

[15] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, “A memory model for scientific algorithms on graphics processors,” in SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM, 2006, p. 89.

[16] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W. 2008. 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 (Salt Lake City, UT, USA, February 20 - 23, 2008). PPoPP '08. ACM, New York, NY, 73-82.

[17] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. 68, 10 (Oct. 2008), 1370-1380.

[18] D. Wilde and S. V. Rajopadhye, “Memory reuse analysis in the polyhedral model,” in Euro-Par ’96: Proceedings of the Second International Euro-Par Conference on Parallel Processing. London, UK: SpringerVerlag, 1996, pp. 389–397.

[19] Xuejun Yang, Li Wang, Jingling Xue, Yu Deng and Ying Zhang, “Comparability Graph Coloring for Optimizing Utilization of Stream Register Files in Stream Processors", The 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009.

[20] P. Briggs, K. D. Cooper, and L. Torczon, “Improvements to graph coloring register allocation,” ACM Trans. Program. Lang. Syst., vol. 16, no. 3, pp. 428–455, 1994.

[21] V. V. Vazirani, Approximation algorithms. New York, NY, USA: Springer-Verlag New York, Inc., 2001.

[22] Shane Ryoo. Program Optimization Strategies for Data-Parallel Many-Core Processors. Ph.D. Dissertation, University of Illinois at Urbana Champaign.

[23] O. Christiansen, T.-M. Lee, J. Lie, U. Sinha, and T. F. Chan, “Total variation regularization of matrix-valued images,” International Journal of Biomedical Imaging, vol. 2007, 2007.

[24] Active Contour and Segmentation Models Using Geometric PDE's for Medical Imaging Tony F. Chan and Luminita A. Vese, in Malladi, R. (Ed.), ``Geometric Methods in Bio-Medical Image Processing", Series: Mathematics and Visualization, Springer, 2002, pp. 63-75.

[25] B. Fischer and J. Modersitzki, "Curvature based image registration," J.Math. Imaging Vis., vol. 18, no. 1, pp. 81–85, 2003.

[26] Kindlmann, G.; Whitaker, R.; Tasdizen, T.; Moller, T., "Curvature-based transfer functions for direct volume rendering: methods and applications," Visualization, 2003. VIS 2003. IEEE , vol., no., pp.513-520, 24-24 Oct. 2003.

[27] Wang, W., Pottmann, H., and Liu, Y. 2006. Fitting B-spline curves to point clouds by curvature-based squared distance minimization. ACM Trans. Graph. 25, 2 (Apr. 2006), 214-238.

[28] C. R. Johns and D. A. Brokenshire, ”Introduction to the Cell Broadband Engine Architecture”, IBM Journal of Research and Development, Vol 51, Number 5, 2007, pp 503-520.

[29] Maryam Moazeni, Alex Bui, Majid Sarrafzadeh, "Accelerating Total Variation Regularization for Matrix-Valued Images on GPUs", In the Proceedings of the 2009 ACM International Conference on Computing Frontiers, May 2009.

[30] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low-power embedded processors. IEEE Transactions on Very Large Scale Integration Systems, 4(8):802815, Aug. 2006.