CUDA-lite: Reducing GPU Programming Complexity

Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{ueng, mlathara, bsadeghi, hwu}@crhc.uiuc.edu

Abstract. The computer industry has transitioned into multi-core and many-core parallel systems. The CUDA programming environment from NVIDIA is an attempt to make programming many-core GPUs more accessible to programmers. However, there are still many burdens placed upon the programmer to maximize performance when using CUDA. One such burden is dealing with the complex memory hierarchy. Efficient and correct usage of the various memories is essential, making a difference of 2-17x in performance. Currently, the task of determining the appropriate memory to use and the coding of data transfer between memories is still left to the programmer. We believe that this task can be better performed by automated tools. We present CUDA-lite, an enhancement to CUDA, as one such tool. We leverage programmer knowledge via annotations to perform transformations and show preliminary results that indicate auto-generated code can have performance comparable to hand coding.

1 Introduction

In 2007, NVIDIA introduced the Compute Unified Device Architecture (CUDA) [9], an extended ANSI C programming model. Under CUDA, Graphics Processing Units (GPUs) consist of many processor cores, each of which can directly address into a global memory. This allows for a much more flexible programming model than previous GPGPU programming models [11], and allows developers to implement a wider variety of data-parallel kernels. As a result, CUDA has rapidly gained acceptance in application domains where GPUs are used to execute compute intensive, data-parallel application kernels.

While GPUs have been designed with higher memory bandwidth than CPUs, the even higher compute throughput of GPUs can easily saturate their available memory bandwidth. For example, the NVIDIA GeForce 8800 GTX comes with 86.4 GB/s memory bandwidth, approximately ten times that of Intel CPUs on a Front Side Bus. However, since the GeForce 8800 has a peak performance of 384 GFLOPS and each floating point operation operates on up to 12 bytes of source data, the available memory bandwidth cannot sustain even a small fraction of the peak performance if all of the source data are accessed from global memory.

Consequently, CUDA and its underlying GPUs offer multiple memory types with different bandwidth, latency, and access restrictions to allow programmers to conserve memory bandwidth while increasing the overall performance of their applications. Currently, CUDA programmers are responsible for explicitly allocating space and managing data movement among the different memories to conserve memory bandwidth. Furthermore, additional hardware mechanisms at the memory interface can enhance the main memory access efficiency if the access patterns follow memory coalescing rules. Currently, CUDA programmers shoulder the responsibility of massaging the code to produce the desirable access patterns. Experiences show that such responsibility presents a major burden on the programmer. CUDA-lite is designed to relieve such burden. Furthermore, CUDA code that is explicitly optimized for one GPU's memory hierarchy design may not easily port to the next generation or other types of data-parallel execution vehicles.

This paper presents CUDA-lite, an experimental enhancement to CUDA that allows programmers to deal only with global memory, the main memory of a GPU, with transformations to leverage the complex memory hierarchy. For increased efficiency, the programmers provide annotations describing certain properties of the data structures and code regions designated for GPU execution. The CUDA-lite tools analyze the code along with these annotations and determine if the memory bandwidth can be conserved and latency can be reduced by utilizing any special memory types and/or by massaging memory access patterns. Upon detection of an opportunity, CUDA-lite performs the transformations and code insertions needed. CUDA-lite is designed as a source-to-source translator. The output is CUDA code with explicit memory-type declarations and data transfers for a particular GPU. We envision CUDA-lite to eventually target multiple types and generations of data-parallel execution vehicles. If maximum performance is desired, the programmer can still choose to program certain kernels at the CUDA level.

In this paper we present CUDA-lite in detail. We cover the memories and techniques that are leveraged by the tool to conserve memory bandwidth and reduce memory latency. We describe how CUDA-lite identifies the opportunities and the hand transformations that it replaces. We have developed plug-ins for the Phoenix compiler [7] from Microsoft to perform all of the transformations as a source-to-source compiler, and evaluated our results by passing the resulting source code through NVIDIA's tool chain. We show that the performance of code generated by CUDA-lite matches or is comparable to hand generated code.

2 CUDA Programming Model

The CUDA programming model is ANSI C extended with keywords and constructs. The GPU is treated as a coprocessor that executes data-parallel kernel functions. The user supplies a single source program encompassing both host (CPU) and kernel (GPU) code. These are separated and compiled by NVIDIA's

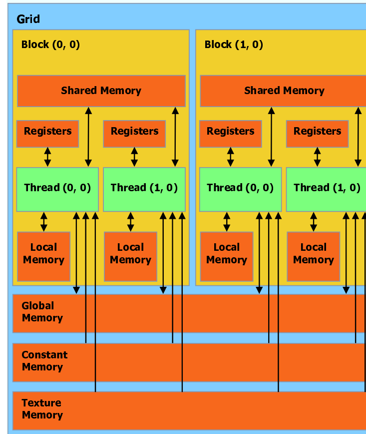


Fig. 1. CUDA Programming Model and Memory Hierarchy

compiler, *nvcc*. The host starts the kernel code with a function call. The complete description of the programming model can be found in [8–10].

Figure 1 depicts the programming model and memory hierarchy of CUDA. Threads are organized into a three-level hierarchy, and are executed on the *streaming multiprocessors* (SMs) on the GPU. At the highest level, each kernel creates a single *grid*, which consists of many *thread blocks* (TBs) arranged in two dimensions. The maximum number of threads per TB is 512, arranged in a three dimensional manner. Each TB is assigned to a single SM for its execution. Each SM can handle up to eight TBs at a time. Threads in the same TB can share data through the on-chip shared memory and can perform barrier synchronization by invoking the `__syncthreads` primitive. Synchronization across TBs can only be safely accomplished by terminating the kernel.

One of the major bottleneck to achieving performance while using CUDA is the memory bandwidth and latency. The GPU provides several different memories with different behaviors and performance that can be leveraged to improve memory performance. However, the programmer must explicitly and correctly utilize these different memories in the source code in order to gain the benefit. In the rest of this section we will examine *shared memory* and desirable memory access patterns to *global memory* that improve memory performance, and show the work required of programmers. Work that CUDA-lite intends to automate.

We focus on memory coalescing for global memory and shared memory in this work since these are the only writable memories in CUDA. We leave the read-only memories, constant and texture, for future work.

2.1 Global Memory

CUDA exposes a general-purpose, random access, readable and writable off-chip global memory visible to all threads. It is the slowest of the available memory spaces, requiring hundreds of cycles, and is not cached. However, its resemblance to a CPU's memory in its generality and size are also what allows more

```
1   #define ASIZE 3000
    #define TPB 256

    __global__ void
5   kernel (float *a, float *b)
    {
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
10    int i;

      if (bki * TPB + thi >= ASIZE)
        return;

15    for (i = 0; i < ASIZE; i++)
        {
          b[(bki*TPB+thi)*ASIZE + i] =
            a[(bki*TPB+thi)*ASIZE + i] * t;
        }
20  }
```

```
int main ()
{
  int num_blocks;
  int size = sizeof (float) * ASIZE * ASIZE;

  /* Allocate a_host and b_host,
   * and initialize a_host with values */

  /* Allocate a_device and b_device */
  cudaMalloc ((void **) &a_device, size);
  cudaMalloc ((void **) &b_device, size);

  /* Copy values from host to device */
  cudaMemcpy (a_device, a_host, size,
              cudaMemcpyHostToDevice);

  num_blocks = ASIZE % TPB == 0 ?
    ASIZE / TPB : (ASIZE / TPB) + 1;

  /* Number of thread blocks in the grid */
  dim3 gridDim (num_blocks);
  /* Number of threads per thread block */
  dim3 blockDim (TPB);

  /* Start executing on the GPU */
  kernel <<<gridDim, blockDim>>>
    (a_device, b_device);

  /* Copy values from device back to host */
  cudaMemcpy (b_host, b_device, size,
              cudaMemcpyDeviceToHost);
}
```

Fig. 2. Example Code: Base Case

general-purpose applications to be ported easily onto the GPU. A straightforward implementation of an application would be to utilize only global memory as a proof of concept for parallelizing the algorithm on CUDA.

Figure 2 shows an example CUDA code. The function `main` sets up the data for computation on the CPU while the function `kernel` contains the code that is actually executed on the GPU. Notice that variables that reside in the global memory of the GPU, like `a_device`, are allocated in `main` and data movement is also performed there via API calls to `cudaMemcpy`.

In the `kernel` function, each thread on the GPU traverses a different row of the 2-D array `a`, scaling each element by a thread specific value before storing into the corresponding location in array `b`. Since each TB must have the same number of threads, depending on the data size and program parallelization there may be excess threads that do not have data to operate on. The conditional check on line 12 that exits the kernel function before the loop handles these cases. This check becomes important as we attempt to utilize memory coalescing (Section 2.3).

2.2 Shared Memory

Shared memory is a small (16KB per SM for the GeForce 8800) readable and writable on-chip memory and as fast as register access. Shared memory is uninitialized at the beginning of execution, and resident data is private to each TB

and visible to all threads within the same TB. The intuition is that shared memory should be used for data that is reused, especially if reused across different threads in a TB. However, we found that memory performance improvement from coalesced global memory accesses (Section 2.3) is large enough that shared memory should be leveraged for such purposes even if there is no data reuse.

2.3 Memory Coalescing

Global memory does have a behavior called *memory coalescing* that helps conserve bandwidth while reducing effective latency. Conceptually it is similar to loading an entire cache line from memory versus loading one word at a time. Threads in a TB are numbered along the x direction first and gathered sequentially into *warps*. On the GeForce 8800 a group of 32 threads form a warp. Each warp executes in SIMD (single-instruction, multiple-data) fashion, i.e. all threads in the same warp execute the same instruction at the same time. When the threads of a half-warp execute a global load, the loads are consolidated if they meet constraints necessary for the hardware to perform memory coalescing. Otherwise the loads are serviced individually. We summarize the requirements here and refer interested readers to [10] for full details.

There are four major requirements that memory accesses to global memory have to follow for memory coalescing to happen:

1. Each element of the array has to be 4, 8, or 16 bytes and aligned.
2. The threads in the half-warp have to access consecutive memory addresses in order, e.g. thread number `N` within the half-warp need to access address `BaseAddr + N`.
3. Thread numbering matters only along the first dimension of the thread block, the x dimension.[1]
4. `BaseAddr` must be aligned to a multiple of the element size.

The requirements for memory coalescing are complex. Furthermore, with the exception of access alignment, all of the requirements must be fulfilled or there will be no improvement in the memory performance; a partial improvement usually occurs if alignment is the only requirement missed. The data access pattern to fulfill the memory coalescing requirement is also not natural for all algorithms, e.g. reduction across the rows of an array. When performing a reduction across the rows of an array, it is more natural to have one thread per row, as in Figure 3(a). The different groups of colored arrows represent different TBs. However, traversing one thread per column, shown in Figure 3(b), is needed to fulfill requirement 2 for memory coalescing. The data accessed by threads in a half-warp need to be adjacent to one another in the horizontal direction, not vertical, for the accesses to coalesce. The lack of synchronization across TBs also contributes to making this traversal pattern unnatural for performing a reduction across rows in CUDA. An alternative is to tile the computation, as shown in Figure

[1] Thread blocks are usually created so that the x dimension is a multiple of the number of threads in a warp.

3(c). The tile is first traversed along the column and data is coalesced loaded into a buffer in shared memory, indicated by the grayed arrows. The algorithm then operates on the data along the row from shared memory before moving to the next tile. The performance improvement from doing coalesced loads and using shared memory makes this worthwhile despite the instruction overhead.

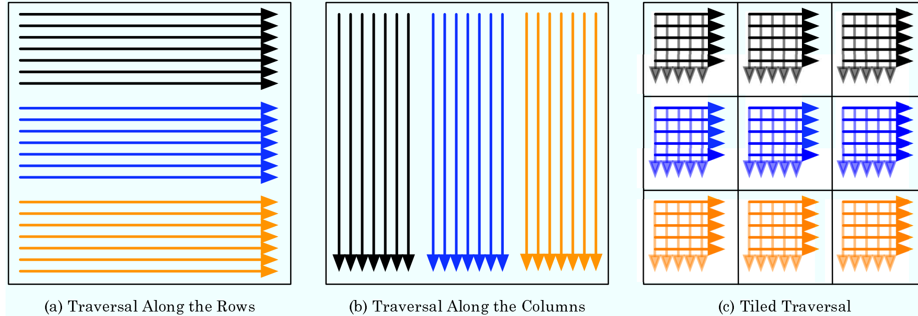


Fig. 3. Graphical View of Data Traversal: (a) Row (b) Column (c) Tiled

For example, the memory access to array `a` on line 18 of Figure 2 does not coalesce because it violates rule number 2. For each iteration of the loop, thread `N` accesses `a[N*ASIZE + i]`; `bki` does not matter since the threads are in the same thread block. This means that each thread is accessing data vertically adjacent to each other, as in Figure 3(a), which does not trigger coalescing.

Figure 4 shows the kernel code from Figure 2 rewritten by hand so the algorithm is tiled and the memory accesses coalesced. The amount of code is roughly doubled. The original loop has been tiled and additional code is inserted to load/store data between global and shared memory. The load from array `a` on line 25 is coalesced since thread `N` accesses `a[k*ASIZE + N]` on each iteration. The computation kernel now operates on the data in shared memory, and the loop around it has included the check on line 12 of the original code as an additional condition. In other words, the excess threads we mentioned back in Section 2.1 may be used to perform memory coalescing accesses, but must not be allowed to perform actual computation.

This rewriting is a large additional burden on the programmer. Not only must the programmer fulfill the memory coalescing requirements, the programmer also has to maintain correctness. The performance improvement this optimization provides will be the ideal, or oracle, case for CUDA-lite.

3 CUDA-lite

Since the behavior of memory coalescing is complex yet understood, we believe that such transformations are best undertaken by an automated tool. This would reduce the potential for errors in writing memory coalescing code, and reduce the burden upon programmers. In our vision, programmers would provide a straight-

```
1   #define ASIZE 3000
    #define TPB 32

    __global__ void
5   kernel (float *a, float *b)
    {
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
10    int i;

      int j, End, k;
      __shared__ float a_shared[TPB][TPB];     Shared Memory
      __shared__ float b_shared[TPB][TPB];
15
      End = ASIZE % TPB == 0 ? ASIZE / TPB : (ASIZE/TPB)+1;     Loop Tiling
      for (j = 0; j < End; j++)
        {
          /* Coalesce loads */
20        __syncthreads();                                        Coalesced Loads
          for (k = 0; k < TPB; k++)
            {
              if ((j*TPB + thi < ASIZE) &&
                  ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE * ASIZE))
25              a_shared[k][thi] = a[(bki*TPB + k)*ASIZE + j*TPB + thi];
            }
          __syncthreads();

          /* Conditions:
30         * TPB && obey original end && !(early exit condition)
           */
          for (i = 0;
           (i < TPB) && (j*TPB+i < ASIZE) && !(bki * TPB + thi >= ASIZE);
           i++)
35          {
              b_shared[thi][i] = a_shared[thi][i] * t;     Computation Kernel
            }

          /* Coalesce stores */
40        __syncthreads();                                        Coalesced Stores
          for (k = 0; k < TPB; k++)
            {
              if ((j*TPB + thi < ASIZE) &&
                  ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE * ASIZE))
45              b[(bki*TPB + k)*ASIZE + j*TPB + thi] = b_shared[k][thi];
            }
          __syncthreads();
        }
49  }
```

Fig. 4. Example Code: Hand Coalesced Kernel

```
(a) __annotation (L"__global__ <threads per block> <thread blocks per SM>");
(b) __annotation (L"garray <name> <rank> <element size> <rank sizes>");
(c) __annotation (L"BoundChk");
(d) __annotation (L"loop <iterator> <start> <end> <increment>");
```

Fig. 5. CUDA-lite Annotations

forward implementation of the kernel code that utilizes only global memory, and depend on tools to optimize the memory performance.

We have developed tools to automate the transformations previously done by hand to maximize memory performance via memory coalescing. The programmer provides a version of the program that has been parallelized for CUDA using only global memory and the tools output a version with the memory accesses optimized. In other words, the tools transform code like the kernel function in Figure 2 to the memory coalescing version in Figure 4. We rely upon information from the programmer provided via annotations to perform our transformations. We call the software tools and annotations together *CUDA-lite*.

Figure 5 shows the current form of the annotations in CUDA-lite. Part (a) indicates the functions of interest, i.e. kernel functions running on the GPU, and parallelization factors. While some of the information, like threads per TB, can eventually be derived from CUDA code, the last argument gives programmers some control over how much resources a kernel generated by CUDA-lite should take. Part (b) indicates what arrays in global memory are of interest and their properties. This gives control over which memory accesses are targeted for optimization, which uses up resources. The speedup gained from performing memory coalescing needs to be balanced against excessive resource usage that reduces executing parallelism. We will discuss this in detail in Section 4. Part (c) is for annotating exit checks, such as the conditional check on line 12 of Figure 2 mentioned in Section 2.1. While CUDA threads may terminate early, CUDA-lite may need those threads to satisfy memory coalescing and synchronization requirements. Therefore CUDA-lite removes the early termination and places guards around the original computation, as mentioned in Section 2.3. Finally, part (d) conveys information about the control flow of loops in the program. We currently use this information to perform loop transformations.

We recognize that some of the information provided by the annotations is derivable by advanced compiler techniques. However, the point of the annotations was to quickly provide the additional information needed and enable the transformations so that the memory hierarchy optimization automation work can proceed. It is not necessarily the final form.

Requirement 2 of the four requirements detailed in Section 2.3 is the most difficult to satisfy and check for. CUDA-lite derives the expression used in global memory accesses by performing a backwards dataflow up to the parameters of the kernel function and thread indices. The expression is first simplified by extracting all references to the thread index in the x direction. We leverage the SIMD execution model to eliminate the need for temporal locality checks, since the execution model guarantees that the expression is the same for all threads in the warp. The desired expression is one where every thread in a half-warp accesses the same location, differing only by their order within the half-warp. Consequently, any instance of $\lfloor thi.x/hwarp \rfloor$ can be safely disregarded, where $thi.x$ is the thread index in the x dimension and $hwarp$ is the number of threads in a half-warp. Mathematically this can be seen as the function f in Equation 1.

As long as the expression fits the form of the function, then the memory access is coalesced.

$$f(thi.x) = thi.x + g\left(\left\lfloor \frac{thi.x}{hwarp} \right\rfloor\right) + C \quad (1)$$

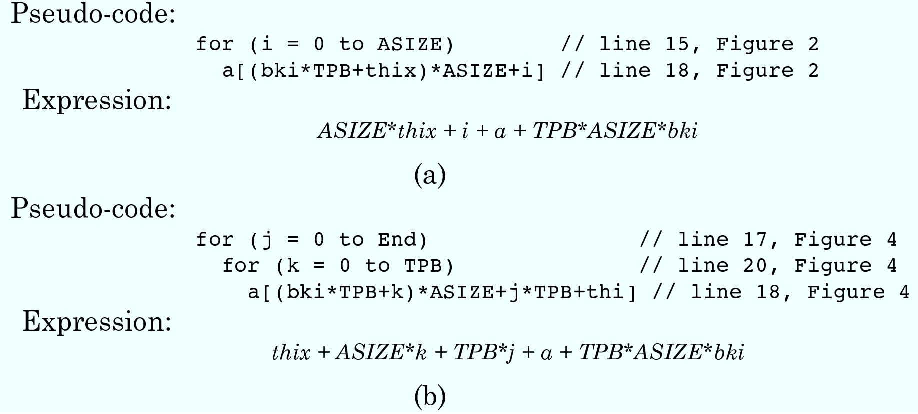


Fig. 6. Array Access and Expression (a) Non-Coalescing (b) Coalescing

Figure 6(a) shows the relevant pseudo-code and expression generated by CUDA-lite for the memory access to array `a` in Figure 2. Due to the `ASIZE` multiplier on the first term, the expression does not fit function f and thus the load is not coalesced. Part (b) shows the memory access to array `a` in Figure 4. Unlike part (a), the expression does fit the form of the function f and therefore the access is coalesced.

If the memory access is not already coalesced, CUDA-lite will attempt to automatically generate a coalescing version. The labels of the additional boxes in Figure 4 outline the majority of the transformations: inserting shared memory variables, performing loop tiling, generating memory coalesced loads and/or stores, and replacing the original global memory accesses with accesses to the corresponding data in shared memory.

The shared memory size and tiling factor are fixed and known for each target GPU, due to the half-warp requirement for memory coalescing. The amount of shared memory allocated can thus be determined by the number of arrays of interest, array dimensions, and array element size. The generation of coalescing loads or stores depends on the relationship between the array dimension and the threading dimension. If they match, then CUDA-lite needs to have each thread load from the appropriate place in global memory into the thread's corresponding position in shared memory. If the array is of higher dimension than the thread organization, two-dimension to one dimension in the running example, then CUDA-lite generates loops that load/store the data. This can be seen in the Coalesced Loads and Stores boxes of Figure 4. These loops must not only

```
1   #define ASIZE 3000
    #define TPB 32

    void
5   kernel (float *a, float *b)
    {
      __annotation (L"__global__ TPB 1");
      __annotation (L"garray a 2 4 ASIZE ASIZE");
      __annotation (L"garray b 2 4 ASIZE ASIZE");
10
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
      int i;
15
      __annotation (L"BoundChk");
      if (bki * TPB + thi >= ASIZE)
        return;

20    for (i = 0; i < ASIZE; i++)
        {
          __annotation (L"loop i 0 ASIZE 1");
          b[(bki*TPB+thi)*ASIZE + i] =
            a[(bki*TPB+thi)*ASIZE + i] * t;
25      }
    }
```

Fig. 7. Example Code: CUDA-lite Kernel

be tiled correctly for correct data movement but they must also obey the array bounds.

Figure 7 shows how the example kernel would be annotated using the current implementation of CUDA-lite. The programmer only needs to insert the five boxed additional lines instead of doubling the amount of code like in Figure 4.

It is important to note that CUDA-lite does not affect parallelization and threading decisions, and operates under the constraints of how the program has been parallelized. This was a deliberate decision to make the problems that CUDA-lite is tackling more tractable. CUDA-lite can be folded into a more comprehensive programming framework for GPU computing system as the part that handles memory optimization.

4 Experimental Results

We have implemented CUDA-lite using the Phoenix compiler [7] as a source-to-source compiler using two Phoenix plug-ins: one to perform the necessary analysis and code transformations, and another to generate source code back from the IR. The regenerated source code is then fed into NVIDIA's compiler nvcc to generate binaries for execution. We used CUDA version 1.0 for all of our experiments. The CPU was an Opteron 248 system running at 2.2GHz with 1GB of memory. The GPU was a GeForce 8800 GTX. The source codes for Phoenix are straightforward CUDA implementations that use only global memory, with slight manipulations so the CUDA extensions not recognized by Phoenix can be passed through and regenerated correctly.

We present three applications as our benchmark: MRI-FHD, TPACF, and the running example of this paper. These three applications display differences in the arrays to be optimized (e.g. 1-D and 2-D) and the level of control flow sophistication (e.g. loop nesting) that CUDA-lite had to handle. MRI-FHD is one of the compute intensive portions of three-dimensional MRI Reconstruction,

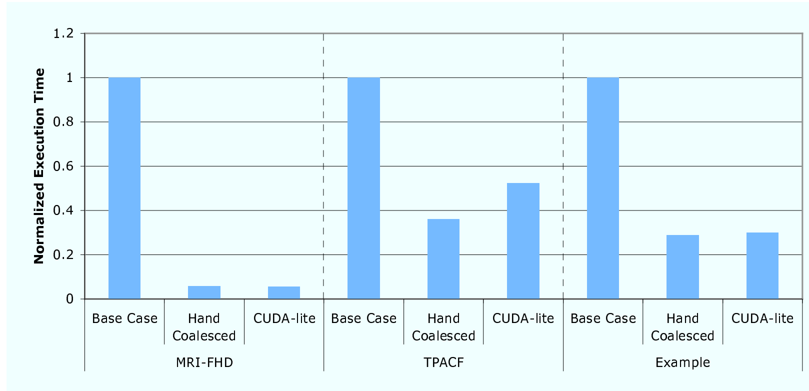
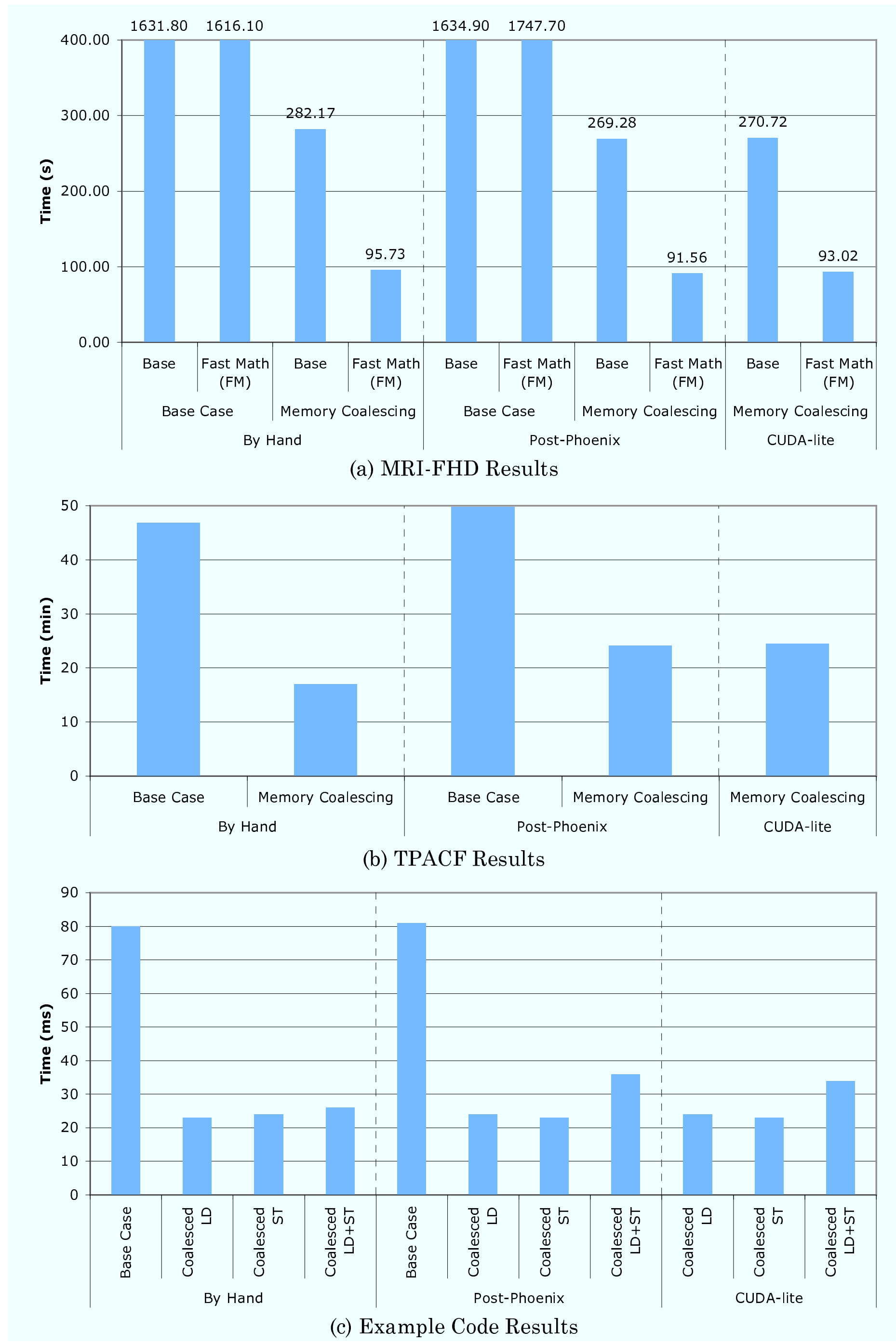


Fig. 8. Overall Results

of which details can be found in [16]. TPACF stands for the two-point angular correlation function, which is used to characterize the probability of finding a cosmological object at a given distance from another cosmological body. A more detailed description of the algorithm can be found in [3]. Both of these programs experienced terrific speedup moving from CPU to GPU [14].

Figure 8 shows the overall results for our benchmarks. The run times are normalized to the base case of the application implemented in CUDA utilizing only global memory. For each application we show base, hand-coalesced, and CUDA-lite results. It is obvious how important improving the memory performance can be, providing between 2 to 17x performance difference in these studies. CUDA-lite, despite being generated from an automated tool, provided performance comparable to the hand-generated versions for all of the applications. We explain the discrepancy of the results between hand-generated and CUDA-lite-generated code in Section 4.1.

Figure 9 shows the detailed results of our experiments. Part (a) is MRI-FHD. Fast math is a compiler option in nvcc to utilize the hardware special function units (SFUs) on the GeForce 8800. This is very beneficial for MRI-FHD because its sine and cosine calculations can be performed on the SFUs. We present three sets of data: Code generated by hand, passing hand-generated code through Phoenix (Post-Phoenix), and CUDA-lite. The second set of data gives an idea of the overhead for going through a translation tool, and provides a more appropriate comparison for CUDA-lite. Note that Post-Phoenix and CUDA-lite memory coalescing code out-perform hand-generated. Fewer registers per thread were allocated by nvcc for the Post-Phoenix and CUDA-lite codes than the hand-generated code, which allowed two TBs to run concurrently on an SM. Otherwise the register allocation allowed only one. Part(b) shows the details of the TPACF results. Although the performance of CUDA-lite is the same as Post-Phoenix, they are both worse than hand-generated. Only one TB could run on an SM at a time in all cases due to shared memory usage.



(a) MRI-FHD Results

(b) TPACF Results

(c) Example Code Results

Fig. 9. Detailed Results: (a) MRI-FHD (b) TPACF (c) Example Code

Figure 9(c) shows the results for the running example code in this paper. We compare the benefits of load coalescing, store coalescing, and both. There are two arrays in the example code. Array `a` is read while array `b` is stored to. Using the annotations in CUDA-lite, we control which accesses are coalesced by the tool. The results indicate that coalescing either the load or the store is better than coalescing both. When only one access to one array is coalesced, up to three TBs can run concurrently on an SM. When accesses to both arrays are coalesced, the amount of shared memory used is doubled and the number of TBs running is reduced to one. Consequently, automatically coalescing all memory accesses is not always a good policy. Resource usage and overall performance need to be taken into account, perhaps in a performance optimization search like in [15].

4.1 Post-Phoenix Overhead

Intuitively, regenerating source code from a compiler should add some amount of overhead. Curiously, our results show that this does not always translate into performance loss. Going through Phoenix showed no ill effect for MRI-FHD, a visible slowdown in TPACF, and mixed results in the example code. We narrowed down the problem to a combination of control flow and executing parallelism.

The output of Phoenix uses only GOTO statements to express the control flow of the program. This results in poor performance on CUDA. We verified this by manually generating versions that consist of only GOTO statements for control flow and observed similar degradations in performance. This explains the slowdown of TPACF and coalesced LD+ST in the example code. Multiple TBs executing on an SM provides additional parallelism to mask this overhead in MRI-FHD.

5 Related Work

Techniques have been proposed to allow array-dominated applications to benefit from scratch-pad memories [5, 12]. In [2], the authors used the polyhedral model to detect data locality and copy the portion of data that is going to be used in a tile into the shared memory (or “scratch-pad memory”) of a GPU. Our motivation and approach is different as we copy data from global memory to shared memory even if there is no data reuse. This is due to the significant performance benefit of coalescing global memory accesses on the GPU architecture.

Related techniques have also been developed to manipulate data accesses for SIMD devices [13, 18]. SIMD units typically operate on short vectors, as opposed to the large massively parallel arrays that CUDA prefers. Also, memory coalescing has to be linear access since that is the requirement from the programming model. Data permutation and rearrangement would apply to setting up the data outside of the GPU kernel, or detecting that the data usage within the kernel covers data in such a way that interaction with the array should be coalesced.

We performed loop transformations such as tiling to properly reorganize the execution pattern. Wolf et al. [17] covered the loop transformations that enhance

data locality in loop nests. We decided not to automate the tiling transformations and rely upon programmer annotations instead since that was not the focus of our work. Our approach is not the same as the multi-level tiling schemes presented in [4, 6], but we share the view that having a global knowledge of data access patterns facilitates improving locality in higher levels of memory hierarchy and increases global memory bandwidth performance.

There exists a body of work that incorporates programmer knowledge in performing transformations. Among these, the Spec# system by Microsoft [1] is closest to our work. It utilizes annotations to allow a separate verifying compiler to check for program correctness. Our annotations are information that feed directly into compiler analyses and transformations, usually information that would otherwise be missing or difficult to infer automatically by the compiler.

6 Conclusion and Future Work

In this paper we introduced CUDA-lite to help relieve programmers of the burden of optimizing the memory performance of code developed under the CUDA programming environment for GPU, which offers a complex memory hierarchy that needs to be leveraged to best match memory bandwidth with compute throughput. This is an important task due to the large effect memory performance has on overall performance (2-17x).

We show that CUDA-lite produces code with performance comparable to hand-coded versions. The coding requirements for CUDA-lite are lower than performing the same transformations by hand and provides a layer of abstraction from the definition of warps in CUDA, which could change in the future. Since CUDA-lite does not handle the parallelizing aspects of GPU programming, we foresee CUDA-lite as the memory optimizing module of an eventual overall framework for facilitating GPGPU programming that encompasses parallelization and resource usage decisions to maximize performance.

For future work we plan to broaden the application set and to extend CUDA-lite to leverage constant memory. We also hope to simplify the annotations in CUDA-lite, some of which can be replaced by compiler analyses currently not in our infrastructure.

Acknowledgment

We would like to thank David Kirk and NVIDIA for generous hardware loans and support. We also thank the anonymous reviewers for their feedback. The authors acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Experiments were made possible by NSF CNS grant 05-51665. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. This work was performed with software donations from Microsoft.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. of Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, volume 3362 of *LNCS*. Springer, March 2004.
2. M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.
3. R. J. Brunner, V. V. Kindratenko, and A. D. Myers. Developing and deploying advanced algorithms to novel supercomputing hardware. In *Proceedings of NASA Science Technology Conference - NCTC'07*, 2007.
4. J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. Programming with tiles. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.
5. M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, 2002.
6. T. J. Knight, J. Y. Park, M. Ren, Mike H., M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.
7. Microsoft. Phoenix compiler. http://research.microsoft.com/Phoenix/.
8. J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum, May 2007.
9. NVIDIA. NVIDIA CUDA. http://www.nvidia.com/cuda.
10. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide: Version 1.0*. NVIDIA Corporation, June 2007.
11. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
12. P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97: Proceedings of the 1997 European Conference on Design and Test*, 1997.
13. G. Ren, P. Wu, and D. A. Padua. Optimizing data permutations for SIMD devices. In *PLDI*, pages 118–131, 2006.
14. S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.
15. S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO*, April 2008.
16. S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, Z. Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs. In *Proceedings of the 2008 International Conference on Computing Frontiers*, May 2008.
17. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.
18. P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *ICS*, pages 169–178, 2005.