An Automated Approach for SIMD Kernel Generation for GPU based Software Acceleration

Kanupriya Gulati and Sunil P Khatri
Department of ECE, Texas A&M University, College Station TX 77843.

***Abstract*— Graphics Processing Units (GPUs) are highly parallel Single Instruction Multiple Data (SIMD) engines, with extremely high degrees of available hardware parallelism. The task of implementing a software routine on a GPU currently requires significant manual design, iteration and experimentation. This paper presents an automated approach to partition a software application into kernels (which are executed in parallel) that can be run on the GPU. Experimental results demonstrate that our approach correctly and efficiently produces fast GPU code, with high quality. We show that with our partitioning approach, we can speedup certain routines by as high as 71% (on avg. 25%) when compared to a monolithic (unpartitioned) implementation. Our entire technique (from reading a C subroutine to generating the partitioned GPU code) is completely automated, and has been verified for correctness.**

I. Introduction

GPUs were natively designed as graphics accelerators for image manipulations, 3D rendering operations, etc. These graphics acceleration tasks require that the same operations are performed independently on different regions of the display. As a result, GPUs were designed to operate in a SIMD fashion, which is a natural computational paradigm for graphical display manipulation tasks.

In recent times, the power of the GPU has been actively exploited for general purpose scientific computations as well [1], [2], [3], [4]. The growth of the general purpose GPU (GPGPU) applications stems from the fact that GPUs, with their large memories, large memory bandwidths, and high degrees of parallelism are readily available as off-the-shelf devices, at very inexpensive prices. The theoretical performance of the GPU [5] has grown from 50 Gflops for the NV40 GPU in 2004 to more than 900 Gflops for GTX 280 GPU in 2008. This high computing power mainly arises due to a heavily pipelined and highly parallel architecture. GPU memory bandwidths have grown from 42 GB/s for the ATI Radeon X1800XT to 141.7 GB/s for the NVIDIA GeForce GTX 280 GPU. Further, the development of open-source programming tools and languages for interfacing with the GPU platforms has further fueled the growth of GPGPU applications.

There are typically two broad approaches that have been employed to accelerate scientific computations on the GPU platform. The first approach is the most common, and involves taking a scientific application, and *re-architecting* it's code to exploit the GPU's capabilities. This redesigned code is now run on the GPU. Significant speedup has been demonstrated in this manner, for several algorithms. Examples of this approach include the GPU implementations of sorting [6], the map-reduce algorithm [7], database operations [8] etc. A good reference in this area is [4].

The second approach involves identifying a particular subroutine S in a CPU based algorithm (which is repeated multiple times in each iteration of the computation, and is found to take up a majority of the runtime of the algorithm), and accelerating it on the GPU. We refer to this approach as the *porting* approach, since only a portion of the original CPU based code is ported on the GPU (without any re-architecting of the code). This approach requires less coding effort than the re-architecting approach. The overall speedup obtained through this approach is, however, subject to Amdahl's law [9]. The re-architecting approach typically requires a significant investment of time and effort. The porting approach is applicable for many problems in which a small number of subroutines are run repeatedly on independent data values, and take up a large fraction of the total runtime. Therefore, an approach to automatically generate GPU code for such problems would be very useful in practice.

In this paper, we focus on automatically generating GPU code for the porting class of problems. Porting implementations require careful partitioning of the subroutine into kernels which are run in parallel on the GPU. Several factors must be considered in order to come up with an optimal solution:

- To maximize the speedup obtained by executing the subroutine on the GPU, numerous and sometimes conflicting constraints imposed by the GPU platform must be accounted for. In fact, if a given subroutine is run without considering certain key constraints, the subroutine may fail to execute on the GPU altogether.
- The number of kernels, and the total communication and computation costs for these kernels must be accounted for as well.

Our kernels are generated to be compiled in the Compute Unified Device Architecture (CUDA), which is an open-source programming and interfacing tool provided by NVIDIA for programming their GPU devices. The GPU device used for our implementation and benchmarking is the NVIDIA GeForce GTX 280. We next briefly discuss the hardware and programming model for the GTX 280 GPU.

A. Hardware Model

The GeForce 280 GTX architecture has 30 multiprocessors (MPs) per chip and 8 processors (ALUs) per multiprocessor. There is no mechanism to communicate *between* the different multiprocessors without the intervention of the host. Each multiprocessor has a set of 16384 32-bit registers and 16KB of shared memory. The total on-board memory in GeForce 280 GTX is 1 GB. The device has about 480 KB of read-only cached memory. The main memory (i.e. the global memory) is readable and writable, and all threads *and* the host can access it. However, global memory is *not* cached and has high latency.

B. CUDA Programming Model

CUDA [10] was released by NVIDIA corporation in early 2007. When programmed through CUDA, the GPU is viewed as a compute device capable of executing a large number of *threads* in parallel. Threads are the atomic units of parallel computation, and the code they execute is called a *GPU kernel*. The GPU device operates as a coprocessor to the main CPU, or host. Data-parallel, compute-intensive portions of applications running on the host can be off-loaded onto the GPU device. Such a portion of code is compiled into the instruction set of the GPU device and the resulting program, called a GPU kernel, is downloaded to the device.

A block (or a thread block) is a batch of threads that can cooperate with each other by efficiently sharing data through fast shared memory, and can synchronize their execution to coordinate memory accesses. Threads are grouped in warps, which are further grouped in blocks. All the threads composing a block are guaranteed to run on the same multiprocessor, and can thus take advantage of shared memory and local synchronization. Blocks have restrictions on the *maximum* number of threads in them. In a GeForce 280 GTX, the maximum number of threads grouped in a block is 512. A set of identical thread blocks is executed on the device by executing one or more blocks on each multiprocessor, using time slicing. However, at a given time, at the most 1024 threads can be active in a single multiprocessor in the 280 GTX device.

II. Our Approach

Our kernel generation engine automatically partitions a given subroutine S into K kernels in a manner that maximizes the speedup obtained by multiple invocations of these kernels on the GPU. Before our algorithm is invoked, the key decision to be made is the determination of which subroutine(s) to parallelize. This is determined by profiling the program and finding the set of subroutines Σ that are invoked repeatedly and independently (with different input data values) and collectively take up a large fraction of the runtime of the entire program. Now each subroutine $S \in \Sigma$ are passed to our kernel generation engine, which automatically generates the GPU kernels for S.

In particular, in our implementation, we assume that S is implemented in the C programming language, and the particular SIMD machine for which the kernels are generated is an NVIDIA GTX 280 GPU. Note that our kernel generation engine is general, and can generate kernels for other GPUs as well. If an alternate GPU is used, this simply means that the cost parameters to our engine need to be modified. Also, our kernel generation engine handles in-line code, nested if-then-else constructs of arbitrary depth, pointers, structures, and non-recursive function calls (by value).

A. GPU Constraints on the Kernel Generation Engine

The use of a GPU based SIMD platform induces some constraints on the kernel generation engine. In this section, we summarize these constraints. In the following section, we describe how these constraints are incorporated in our automatic kernel generation engine.

- As mentioned earlier, the NVIDIA GTX 280 GPU consists of 30 multiprocessors, each of which has 8 processors on it. As a result, there are 240 hardware processors in all, on the GPU IC. For maximum hardware utilization, it is important that we issue significantly more than 240 threads at once. By issuing a large number of threads in

parallel, the data read/write latencies of any thread are hidden, resulting in a maximal utilization of the processors of the GPU, and hence ensuring maximal speedup.

- There are 16384 32-bit registers per multiprocessor. Therefore if a subroutine S is partitioned into K kernels, with the i^{th} kernel utilizing r_i registers, then we should have $max_i(r_i) \cdot$ (# of threads per MP) ≤ 16384. This argues that across all our kernels, if $max_i(r_i)$ is too small, then registers will not be completely utilized (since the number of threads per multiprocessor is limited to 1024), and kernels will be smaller than they need to be (thereby making K larger). This will increase the communication cost between kernels.
  On the other hand, if $max_i(r_i)$ is very high (say 4000 registers for example), then no more than 4 threads can be issued in parallel. As a result, the latency of accessing off-chip memory will not be hidden in such a scenario. In the CUDA programming model, if r_i for the i^{th} kernel is too large, then the kernel fails to launch. Therefore, satisfying this constraint is important to ensure the execution of any kernel. We try to ensure that r_i is roughly constant across all kernels,
- The number of threads per multiprocessor must be
  - A multiple of 32 (since 32 threads are issued per warp, the minimum unit of issue),
  - Less than or equal to 1024, since there can be at most 1024 threads issued at a time, per multiprocessor.

  If the above conditions are not satisfied, then there will be less than complete utilization of the hardware. Further, we need to ensure that the number of threads per block is at least 128, to allow enough instructions such that the scheduler can effectively overlap transfer and compute instructions. Finally, at the most 8 blocks per multiprocessor can be active at a time.
- When the subroutine S is partitioned into smaller kernels, the data that is written by kernel k_1 and needs to be read by kernel k_2 will be stored in global memory. So we need to minimize the total amount of data transferred between kernels in this manner. Due to high global memory access latencies, this memory is accessed in a coalesced manner.
- To obtain maximal speedup, we need to ensure that the cumulative computation time over all kernels is as low as possible.
- We need to ensure that the number of registers per thread is minimized such that the multiprocessors are not allotted less than 100% of the threads that they are configured to run with.
- Finally, we need to minimize the number of kernels K, since each kernel has an invocation cost associated with it. Minimizing K ensures that the aggregate invocation cost is low.

Note that the above guidelines often place conflicting constraints on the same variable. Our kernel generation algorithm is guided by a cost function which quantifies these constraints, and hence is able to obtain the optimal solution for the problem.

B. Automatic Kernel Generation Engine

The pseudocode for our automatic kernel generation engine is shown in Algorithm 1. The input to the algorithm is the subroutine S which needs to be partitioned into GPU kernels, and the number N of independent calls of S that are made in parallel.

Algorithm 1 Automatic Kernel Generation(N, S)

```
BESTCOST ← ∞
G(V,E) ← extract_graph(S)
for K = K_min to K_max do
  P ← partition(G,K)
  for P ∈ P do
    Q ← make_acyclic(P)
    for Q ∈ Q do
      if cost(Q) < BESTCOST then
        golden_config ← Q
        BESTCOST ← cost(Q)
      end if
    end for
  end for
end for
generate_kernels(golden_config)
```

The first step of our algorithm constructs the companion control and dataflow graph $G(V,E)$ of the C program. This is done using the Oink [11] tool. Oink is a set of C++ static analysis tools. Each unique line l of the subroutine S corresponds to a unique vertex v of G. If there is a variable written in line l_1 of S which is read by line l_2 of S, then the directed edge $(v_1, v_2) \in E$. Each edge has a weight associated with it, which is proportional to the number of bytes that are transferred between the source node and the sink node. An example code fragment and its graph G (with edge weights suppressed) are shown in Figure 1.

Note that if there are if-then-else statements in the code, then the resulting graph has edges between the node corresponding to the condition being checked and each of the statements in the then and else blocks, as shown in Figure 1.

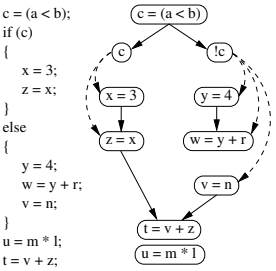

Fig. 1. CDFG Example

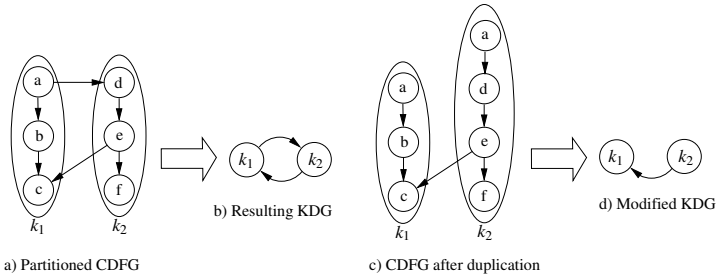

Fig. 2. KDG example

Now our algorithm computes a set $\mathcal{P}$ of K-way partitions of the graph G, which is inherently directed. We use hMetis [12] for this purpose. Since hMetis (and other graph partitioning tools) operate on undirected graphs, there is a possibility of hMetis' solution being infeasible for our purpose. This is illustrated in Figure 2. Consider a companion CDFG G which is partitioned into 2 partitions k_1 and k_2 as shown in Figure 2 a). Partition k_1 consists of nodes a, b and c, while partition k_2 consists of nodes d, e and f. From this partitioning solution, we induce a *kernel dependency graph (KDG)* $G_K(V_K, E_K)$ as shown in Figure 2 b). In this graph, $v_i \in V_K$ iff k_i is a partition of G. Also, there is a directed edge $e_k(v_i, v_j) \in E_K$ iff $\exists n_p, n_q \in V$ s.t. $e(n_p, n_q) \in E$ and $n_p \in k_i$, $n_q \in k_j$. Note that a cyclic kernel dependency graph is an infeasible solution for our purpose, since kernels need to be issued sequentially. To fix this situation, we selectively duplicate nodes in the CDFG, such that the modified KDG is acyclic. Figure 2 c) illustrates how duplicating node a ensures that the modified KDG that is induced (Figure 2 d)) is acyclic.

In our kernel generation engine, we explore several K-way partitions. K is varied from K_{min} to a maximum value K_{max}. For each of the explored partitions of the graph G, a cost is computed. This estimates the cost of implementing the partition on the GPU. The details of the cost function are described in the next section. The lowest cost result *golden_config* is stored. Based on *golden_config*, we generate GPU kernels (using a PERL script). Suppose that *golden_config* was obtained by a k-way partitioning of S. Then each of the k partitions of *golden_config* yields a GPU kernel, which is automatically generated by our PERL script.

Data that is written by a kernel k_i and read by another kernel k_j (k_i, $k_j < k$) is stored in the global memory in an array of length equal the number of threads issued, and indexed at a location which is always aligned to a 32 byte boundary. This enables coalesced write and read accesses by threads executing kernel k_i and k_j, respectively. Since the cached memories are read-only memories, we cannot use them for communication between kernels. Also, since the given subroutine S has N *independent calls*, our generated kernels do not create any memory access conflicts when accessing global memory.

1) Cost of a Partitioning Solution

The cost of each partitioning solution is computed using several cost parameters, which are described next. In particular, our cost function C considers 4 parameters $\mathbf{x} = \{x_1, x_2, \cdots, x_4\}$. We consider a linear cost function, $C = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4$.

1) (Parameter x_1): The first parameter to our cost function is the number of partitions being used. The GPU runtime is significantly modulated by this term, and hence it is included in our cost model.
2) (Parameter x_2): This parameter measures the total time spent in communication to and from the device's global memory $x_2 = [\frac{\sum_{i=1}^{K}(B_i)}{BW}]$. Here B_i is the number of read or write transfers that are required

for the kernel i, and BW is the peak bandwidth for coalesced global memory transfers. Therefore the term x_2 represents the total amount of time that is spent in communicating data, when any one of the N calls of the subroutine S is executed.

3) (Parameter x_3): The total computation time is estimated in this parameter. Note that due to node duplication, the total computation time is not a constant across different partitioning solutions. Let C_i be the number of GPU clock cycles taken by kernel i. We estimate C_i based on the number of clock cycles for various instructions like integer and floating point addition, multiplication and division, library functions for exponential, square root, etc. This information is available from NVIDIA. Also let F be the frequency of operation of the GPU. Therefore, the time taken to execute the i^{th} kernel is $\frac{C_i}{F}$. Based on this, $x_3 = \frac{\sum_{i=1}^{K}(C_i)}{F}$.
4) (Parameter x_4): We also require that the average number of registers over all kernels is a small number. As discussed earlier, this is important to maximize speedup. This parameter (for each kernel) is provided by the *nvcc* compiler that is provided along with the CUDA distribution.

III. Experiments

Our kernel generation engine handles C programs. It handles non-recursive function calls (by value), pointers, structures, and if-else constructs. The kernel generation tool is implemented in perl, and it uses hMetis [12] for partitioning, and Oink [11] for generating the CDFG.

A. Evaluation Methodology

Our approach consists of two steps. In the first step, we compute the weights $\alpha_1, \alpha_2, \cdots, \alpha_4$. This is done by taking a set of code benchmarks. For all these C-code examples, we generate the GPU code for the examples with 1, 2, 3, 4, $\cdots$ 20 partitions (kernels). The code is then run on the GPU, and the values of runtime as well as all the $\mathbf{x}$ variables are recorded in each instance. From this data, we fit the cost function $C = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4$ in MATLAB. We take the actual runtime of any partitioning solution on the GPU as its cost.

In the second step, we would use the values of the α_i weights computed in the first step, and run our kernel generation engine on the code which is to be parallelized on the GPU. In particular, we select the best 5 partitions (those which produce the 5 smallest values of the cost function). The kernel generation engine would also generate the GPU kernels for these partitions, and determine the best solution among the 5 (i.e. the solution which has the fastest GPU runtime) after executing them on the GPU.

Our experiments were conducted over a set of 3 benchmarks. These were:

- MM. This code performs matrix multiplication. We experiment with MM for matrices of various sizes (4x4 and 8x8). We experiment with fixed point (MMI) as well as floating point (MMF) versions of MM.
- LU. This code performs LU-decomposition, required during the solution of a linear system. We experiments with systems of varying sizes (matrices of size 4x4 and 8x8).

For our experiments, in the first step of the approach, we use the LU benchmarks (for both 4x4 and 8x8 matrices) and determined the values of α_i. The values of these parameters were determined to be $\alpha_1 = 5.1868$, $\alpha_2 = 0.3485$, $\alpha_3 = -0.0368$ and $\alpha_4 = 1.2991$.

Now in the second step, we tested the usefulness of our approach on the remaining benchmarks (MMI and MMF, for matrices of size 4x4 and 8x8).

The results which demonstrate the fidelity of our kernel generation engine are shown in Table I. In this table, the first column reports the number of partitions being considered. Columns 2, 4, 6 and 8 indicate the 5 best partitioning solutions based on our cost model, for the MMI4, MMI8, MMF4 and MMF8 benchmarks respectively. If our approach had perfect prediction fidelity, then these 5 partitioning solutions would have the lowest runtimes on the GPU.

Columns 3, 5, 7 and 9 report the GPU runtimes (in milliseconds) for the MMI4, MMI8, MMF4 and MMF8 benchmarks respectively. The 5 solutions that actually had the lowest GPU runtimes are highlighted in bold font, with the fastest solution marked with a "$\star$".

From these results, we can see the need for partitioning these subroutines. For instance in MMI8 benchmark, the fastest result obtained is with partitioning the code into 10 kernels, which makes it 71% faster compared to the runtime obtained using one monolithic kernel. Similarly for MMI4, MMF4 and MMF8 the speedup obtained by our partitioing approach is 18%, 3% and 8%, respectively.

We can further observe that our kernel generation approach correctly predicts 4, 3, 3 and 3 of the 5 best solutions (for the MMI4, MMI8, MMF4 and MMF8 benchmarks respectively). In 3 of the 4 benchmarks, our 5 best partitioning solutions included the solution that was actually the best when run on the GPU. In the fourth benchmark (MMF8), the GPU runtime of the best solution among the 5 predicted best solutions was only 4.17% slower than the solution that was actually the fastest when run on the GPU.

Prts.	MMI4		MMI8		MMF4		MMF8	
	Best pred.	GPU time	Best pred.	GPU time	Best pred.	GPU time	Best pred.	GPU time
1	√	**92.82**		289.16	√	**65.63**	√	**110.82**
2	√	**83.97**		353.53	√	324.47		114.36
3	√	**91.31**	√	298.50	√	**63.70**$\star$		1988.37
4		177.40	√	476.86		**66.96**	√	957.00
5	√	89.40		1118.72	√	81.01		1027.36
6	√	**75.81**$\star$		784.21		163.27		2249.39
7		**78.98**	√	**143.45**	√	**79.86**		1654.47
8		182.73		784.22		**77.64**	√	**105.85**
9		105.78		1121.62		202.61		798.85
10		133.14	√	**82.12**$\star$		80.70	√	156.84
11		447.58		521.53		235.98		401.71
12		90.65		433.84		91.79		**101.65**$\star$
13		102.39	√	**132.70**		94.15		**104.92**
14		101.65		**163.29**		100.13		233.51
15		122.09		266.63		96.16		148.29
16		101.84		543.54		102.93	√	**108.01**
17		99.40		508.95		91.14		1020.10
18		94.92		559.10		105.15		357.71
19		102.40		**133.97**		176.45		596.27
20		98.18		664.43		99.08		614.99

TABLE I
VALIDATION OF THE AUTOMATIC KERNEL GENERATION APPROACH

In general the GPU runtimes tend to be noisy, and hence it is hard to obtain 100% prediction fidelity. We are currently conducting further studies to explain the variations in the GPU runtimes. We also plan to test our approach on a larger and more general set of benchmarks.

IV. Conclusions

GPUs are highly parallel SIMD engines, with extremely high degrees of available hardware parallelism. These platforms have received significant interest for accelerating scientific software applications in recent times. The task of implementing a software application on a GPU currently requires significant manual intervention, iteration and experimentation. This paper presents an automated approach to partition a software application into kernels (which are executed in parallel) that can be run on the GPU. The input to our algorithm is a subroutine which needs to be accelerated on the GPU. Our approach automatically partitions this routine into GPU kernels. Various partitions are explored, and each is given a cost which accounts for GPU hardware and software constraints. Based on the least cost partition, our approach automatically generates the resulting GPU code. Experimental results demonstrate that our approach correctly and efficiently produces fast GPU code, with high quality. Our results show that with our partitioning approach, we can speedup certain routines by as high as 71% (on avg. 25%) when compared to a monolithic (unpartitioned) implementation. Our entire flow (from reading a C subroutine to generating the partitioned GPU code) is completely automated, and has been verified for correctness.

References

[1] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47.

[2] J. Owens, "GPU architecture overview," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. New York, NY, USA: ACM, 2007, p. 2.

[3] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: general-purpose computation on graphics hardware," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 208.

[4] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[5] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, and T. Chiueh, "Graphic-card cluster for astrophysics (GraCCA) – performance tests," in *Submitted to NewAstronomy*, July 2007.

[6] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1381–1388, 2008.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.

[8] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 215–226.

[9] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," vol. 30, 1967.

[10] "NVIDIA CUDA Homepage," `http://developer.nvidia.com/object/cuda.html`.

[11] "Oink: a collaboration of C++ static analysis tools," `http://www.cubewano.org/oink`.

[12] G. Karypis and V. Kumar, *hMETIS : A Hypergraph partitioning package*, University of Minnesota, Dept of Computer Science and Engineering, Nov 1998.