Reuse and Refactoring of GPU Kernels to Design Complex Applications

Santonu Sarkar*, Sayantan Mitra*, Ashok Srinivasan†
*Infosys Labs, Infosys Ltd. Bangalore 560100, India
Email: {santonu_sarkar01,sayantan_mitra01}@infosys.com
†Dept. of Computer Science, Florida State University, Tallahassee, FL 32306, USA
Email: asriniva@cs.fsu.edu

***Abstract*—Developers of GPU kernels, such as FFT, linear solvers, etc, tune their code extensively in order to obtain optimal performance, making efficient use of different resources available on the GPU. Complex applications are composed of several such kernel components. The software engineering community has performed extensive research on component-based design to build generic and flexible components, such that the component can be reused across diverse applications, rather than optimizing its performance. Since a GPU is used primarily to improve performance, application performance becomes a key design issue. The contribution of our work lies in extending component based design research in a new direction, dealing with the performance impact of refactoring an application consisting of the composition of highly tuned kernels. Such refactoring can make the composition more effective with respect to GPU resource usage especially when combined with suitable scheduling. Here we propose a methodology where developers of highly tuned kernels can enable application designers to optimize performance of the composition. Kernel developers characterize the performance of a kernel through its "performance signature". The application designer combines these kernels such that such that the performance of the refactored kernel is better than the sum of the performances of the individual kernels.This is partly based on the observation that different kernels may make unbalanced use of different GPU resources like different types of memory. Kernels may also have the potential to share data. Refactoring the kernels, combining them, and scheduling them suitably can improve performance. We study different types of potential design optimizations and evaluate their effectiveness on different types of kernels. This may even involve choosing non-optimal parameters for an individual kernel. We analyze how the performance signature of the composition changes from that of the individual kernels through our techniques. We demonstrate that our techniques lead to over 50% improvement with some kernels. Furthermore, the performance of a basic molecular dynamics application can be improved by around 25.7%, on a Fermi GPU, compared with an un-refactored implementation.**

***Keywords*-gpu, component reuse, kernel composition, refactoring**

I. Introduction

There is growing recognition of the need for good software design support on parallel platforms [1]. With the growing popularity of general purpose computing on Graphics Processing Units (GPU), the need for a suitable design methodology and tool for the creation of complex applications on GPUs is becoming increasingly apparent. Such applications are composed of a large number of kernels. For example, a GPU port of the QMCPack Quantum Monte Carlo software [2] consists of around 120 GPU kernels. A variety of other applications, such as linear flow solvers, are also built by composing several kernel components. Developers of such complex GPU applications spend significant effort and time to fine tune the code for optimal usage of the GPU resources. Often, the kernel components are generic in nature and have a potential of being reused in other application contexts. An application comprising of a set of kernel components may have data dependencies, requiring that they be run in sequence, or they may be run concurrently. A design methodology, that promotes construction of generic kernels which can be composed later in other applications to yield good performance, can significantly reduce the design effort of complex GPU applications.

The key design concern here is to enable an application consisting of a set of kernel components to make optimal use of the hardware resources available in a GPU. Assume that a reusable kernel is well written, and it has some tunable parameters, such as the granularity of parallelization, with a good choice of parameters enabling optimal performance. Even in such a case, the optimal parameters for the application, which is a composition of kernels, does not necessarily correspond to choosing the optimal parameters for each component. To illustrate this point, consider two kernel components A and B, written by GPU experts. Each of them runs optimally with the entire resource at its disposal. The resources could be the amount of shared memory used or the fraction of Streaming Multiprocessor (SM) used. However, these developers are obviously unaware of the context in which these modules will be used in future. It could very well be the case that A and B can be run concurrently, with 60% of the resources allocated to A and 40% to B, for optimal performance on some application. Thus, in a real situation, it will be necessary to optimize functional, architectural and hardware parameters (number of blocks, threads per block, register usage etc) of both A and B to arrive at these magic figures. It has been shown that these parameters are discrete [3] and even a simple version of an optimization problem becomes NP-hard.

Constructing a complex software by composing a set of reusable components has been well studied by the software

engineering research community [4] and many off-the-shelf design tools are available for building and reuse of component libraries. The software is designed to run on a sequential hardware, where the main design concern has been to make a component as flexible and generic as possible for maximal reuse. However, when the underlying hardware becomes massively data parallel like a GPU, the composition from the performance point of view becomes extremely challenging, and the subject has remained relatively less explored.

In this paper we study the composition of GPU kernels, to build an application and subsequent refactoring[1] of the application for performance tuning. Here we do not propose any approach to optimize individual kernels. Rather, we have assumed that kernel developers have provided highly tuned kernels. We propose a design methodology where the responsibility of a kernel developer is to provide a "performance signature" to characterize the performance of a kernel, alongwith its implementation. Assuming that a set of kernel implementations including the optimal implementation, and corresponding performance signatures are available, we show various ways in which an application developer can compose these kernels such that the resulting performance is better than the sum of the performances of the individual kernels. Further, we analyze why particular compositions work better on different GPU architectures – an NVIDIA S1050 Tesla and a GTX 480 (Fermi). Our analysis reveals that different kernels may make unbalanced use of different GPU resources like different types of memory. Kernels may also have the potential to share data. Refactoring the kernels, combining them, and scheduling them suitably can improve performance.

The paper has been organized as follows. In Section II we summarize relevant features of the GPU architectures and CUDA programming. Next, we discuss our design methodology in Section III. We discuss the experimental study in Section IV. We compare our work with related work in Section VII and finally summarize our conclusions.

II. GPU Architecture

GPU architecture varies across different versions and makes. Here, we provide a brief overview of a typical NVIDIA GPU. A GPU consists of several streaming multiprocessors (SMs) and an SM can execute one of more blocks of threads, where a block of threads can synchronize efficiently through hardware. A GPU has a large global memory which can be accessed by several SMs. Each SM has a shared memory which can be accessed by threads in a thread block. In addition, an SM has several registers which are exclusively owned by a thread. Threads in a given thread block, are partitioned into groups called warps. The GPU scheduler treats a warp as a scheduling unit, where all the threads in a warp run in a lock-step fashion, i.e. each thread in a warp, executes the same instruction in parallel, on different data.

General purpose GPU programming was popularized by CUDA[2] [5], which is an extension to C. In CUDA, the part of the code (and the associated data structure) that needs to run in a data parallel manner is called a kernel function. The process running on the host (CPU) copies relevant data to the device (GPU) and then calls a kernel. A kernel function is executed in a GPU by many threads. Once the kernel completes, its output data needs to be copied back to the host memory.

Programmers need to fine tune resource usage under the architectural constraints (which often turns out to be a complex decision) so as to get an optimal performance [3]. These architectural features include efficient access to different types of memory, such as the global memory, shared memory, and constant memory, efficient synchronization within a thread block in contrast to kernel level synchronization between blocks, enabling data parallelism by reducing branch divergence within a warp, etc.

III. Design Methodology

In order to ensure that the kernel is designed and reused keeping optimal performance in mind, we propose a two pronged approach. We describe two specfic roles, one the kernel developer and the other, the application developer; along with their responsibilities.

A. Kernel Tuning by Kernel Developer

Kernel developers fine tune the kernel for optimal utilization of the GPU resource. As mentioned earlier, we do not propose any optimization approach for the kernel developer. We assume that the kernel developer has tested and fine tuned the kernel such that its performance is optimal. We also assume that the kernel developer is able to write a kernel as a library function and make it generic. The kernel developer often develops multiple implementations of the code, such as a global memory implementation and a shared memory implementation, evaluates them with different algorithmic and hardware related parameters, such as threads per block, and releases the optimal version. We propose that the kernel developer publishes all the different implementations of the kernel, including the optimal one and suboptimal ones. Each implementation is associated with a metadata, called **performance signature**.

1) Performance Signature: A performance signature is a characterization of the performance and resource usage as a function of parameters under the application developers control. For example, for a basic molecular dynamics application shown in Fig.1a, we define the performance

[1]Refactoring involves modifications to the implementation while maintaining the semantics of a module.

[2]While OpenCL is a standard, CUDA is still popular, and so we describe it. In any case, the methodology described in this paper is independent of the programming language used.

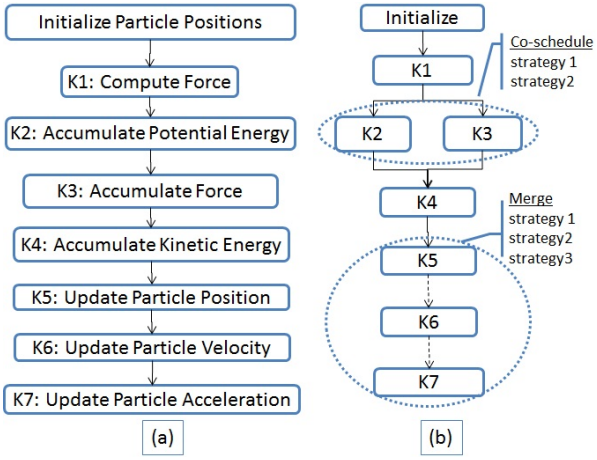

Figure 1. Molecular Dynamics Application:(a)Flow Diagram (b)Design and Refactoring Strategies

signature for kernel K by values of $\mathcal{PS}(K, Arch) =< Reg, T/B, Mem, Time >$ for a given GPU architecture $Arch$. The input $Arch$ takes a nominal value such as "`tesla`" or "`fermi`". Here Reg indicates the number of registers per thread required by the given implementation of K. The next attribute T/B indicates the number of threads per block required, and Mem attribute characterizes the type of memory used in the implementation. This attribute value is also nominal, and is like "`global`", "`shared`" etc. The performance signature corresponding to the implementation that has optimal performance, is denoted by $\mathcal{PS}_{opt}(K, Arch)$.

The notion of multiple implementations, conforming to an interface is well-known in the domain of modular software design. This concept, pioneered by Parnas and others [6], suggests that a software is a composition of modules where a module has an interface, with one or more implementations of the interface. Here we extend this concept, from the performance point of view. Here the kernel developer not only provides the optimal version, but also provides a sub-optimal version of these kernels and their performance signatures. For example, if a kernel performs optimally when the data resides in *shared memory*, the developer also provides an alternate *global memory* based implementation of the kernel. In the next section we demonstrate that multiple kernel implementations become different design choices for application developers for the optimal design of an application.

B. Design Choice Exploration by Application Developer

Given a set of kernels with their performance signatures, we now describe how an application developer will compose these kernels to build an application. The application developer begins with a flow graph to define a composite application created out of a set of predefined kernels such as the one shown in Figure 1b for molecular dynamics, which is explained later. Here, K2 and K3 do not have any

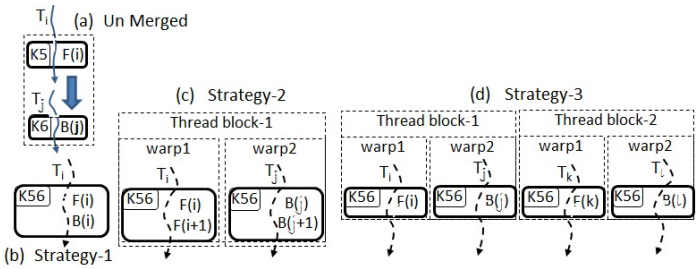

Figure 2. Merge Strategy

dependency whereas K5, K6, and K7 have dependencies. The dotted line in this figure denotes anti-dependence [7], [8], that is, the input set to the first kernel intersects the output set to the next kernel. Similarly the developer can specify that the kernel K4 in this figure is a reduction kernel (a design hint, indicating that a shared memory implementation will yield better performance).

1) Refactoring through Kernel Merging: Merging kernels can be useful if each kernel makes unbalanced use of GPU resources, such as registers or threads permitted per block, so that the combination can make better use of them. Such merging can also be useful when kernels share data; it may happen that shared memory is not useful for a particular kernel because there is not enough data reuse, but the combined kernel has enough data reuse to make shared memory useful. If the output of one kernel is the input to the next, then merging can also reduce data movement. The performance signature and flow graph can help identify kernels to be merged. For example, if the optimal implementation of two kernels, when combined, will not exhaust the resources available, such as registers and threads per block, then they may be merged. More interestingly, there may be a benefit to merging even if non-optimal parameters will not exhaust the resources, if the performance for non-optimal parameters is not very different from the performance for optimal parameters. We later show how the performance signature for the merged kernels differ from that of a direct combination of the original kernels. We next describe three strategies for merging kernels.

Strategy 1: In this strategy kernel K_i's instructions are executed before K_j's. When K_j has a strict data-dependency on K_i this is the only feasible merge strategy, if the dependence can be enforced in each thread block. The idea is illustrated in Figure 2b.

Next, let's consider a more complex scenario. The data flow analysis reveals an anti-dependence [7], [8] between K5 and K6 which can be removed through the standard technique of variable cloning and renaming. The following two merge strategies can be used. Of course, these strategies can also be used if there is no dependence, though co-scheduling, which is described later, could be more effective, unless the kernels share common input which will make merging beneficial.

Strategy 2: As shown in Figure 2c, we keep the number of thread blocks same, and different warps perform the computations of the different original kernels. This implies that each thread performs double the computation of the standalone scenario (Fig 2a).

Strategy 3: Like strategy 2, here we let different warps of the same thread block do computations on different kernels. However, we keep the workload of each thread to be the same, which increases the number of blocks, providing the potential to decrease global memory access latency.

2) Co-Scheduling: Design decisions can be influenced by hardware architecture. Since a Fermi GPU allows concurrent execution of kernels, the designer can opt for concurrent execution of kernels which do not have data dependencies. In the experiment section we show performances of two independent kernels K2 and K3 in Figure 1. We also investigate the use of this strategy for pseudo parallel execution of two kernels K7 and K1 in Figure 1, as explained below. The computation in this flow graph is repeated in a loop. Such loops are fairly common in scientific applications. In this particular application, computation of K7 in iteration i is independent of the computation of K1 in iteration $i + 1$. They can, consequently, be executed concurrently.

3) Optimal Design Choice: Given above strategies, what could be the optimal design choice? Given a set of performance signatures for each kernel, including the optimal signature, which implementation of kernels, say, K5 and K6 should one choose during merging? Is it always the case that selection of optimal implementations of two kernels yield optimal result when they are merged? We show in the following section that this is always not the case and explain the reason.

The next question is, which merge strategy will work best in this case? Furthermore, in a flow graph like 1b, should one merge K5 & K6 or K6 & K7 or K5, K6, & K7? In the following section we investigate these issues through experimental study.

IV. Experimental Study

For our analysis, we have taken a simple example of molecular dynamics simulation which works as follows, as shown in Figure 1a.

1. Initialize: Start with a set of particles, placed in a 3-dimensional box, with initial positions, velocities, and accelerations drawn from a specified distribution.

2. Compute: Compute the potential energy or force of each particle using a Lennard-Jones potential. Also compute the kinetic energy of each particle and then compute total energies of all particles.

3. Update: Update the position, velocity and acceleration of each particle.

4. Advance the time-step and go to Step 2 again.

We have converted the above example to CUDA from the original source[3]. In Figure 1b we have shown their data dependencies in a flow graph, where we have used symbolic names $\{K1, K2, \cdots K7\}$ for these kernels. Kernel K1 to K7 has been run repeatedly for each time step. Note that in this application, the time taken per iteration is small; however, the total time for the computation is large because an extremely large number of iterations are required for a realistic computation.

In the first part of the experiment, presented in the following subsection, we have explained the process of collecting optimal as well as sub-optimal performance signatures by a *kernel developer*. Next we have switched to the role of an *application developer*, where the kernel components, their implementations and performance signatures have been assumed to be present. Here we have elaborated the selection of appropriate kernel implementation and composition of kernels through experimental analysis.

A. Infrastructure Used

We report our results both on an NVIDIA S1050 Tesla and on an NVIDIA GTX 480 Fermi GPU. For both these devices, the host is an Intel Xeon 2.5 GHz CPU running Linux kernel version 2.6.18. The host code has been compiled using gcc version 4.3.3 and the kernel code has been compiled with NVCC version 3.2 and 4.0 on the Tesla and Fermi GPU respectively. We have reported the performance using a RDTSC timer on the host that has a resolution of $1ns$. Each kernel execution has been repeated for a sufficiently large number of times. We ignore the time for the first iteration, in order to discount the GPU warmup overhead from our measurement, and then report the average execution time for the subsequent iterations. We have noted the kernel launch overhead on the Tesla to be 21μs, while on the Fermi it is 7μs.

V. Experiments by Kernel Developer

We have run the application by varying the number of threads per block for a given number of particles. We have observed the time taken by each kernel. We have also noted the number of registers being used per thread by each kernel. Table I shows the performance results of each kernel when they have been executed on a Tesla (T) and Fermi (F) respectively. The first part of the table shows the results when the data resides in global memory.

We have re-implemented kernels K2 to K7 as shared memory variants and reported their results in the second part of Table I. We have renamed the shared memory variant of these kernels as K2sh, K3sh etc.

A. Performance Signature

From Table I the kernel developer derives $\mathcal{PS}(K, Arch) =< Reg, TB, Mem, Time >$ for the kernels K1$\cdots$K7. Table II shows the optimal signature

[3]http://people.sc.fsu.edu/~jburkardt/f_src/md_open_mp/md_open_mp.f90

Kernel	#reg./ thread	Execution time for kernels for varying #threads/block when data is in ***GLOBAL*** memory				
		32	64	128	256	512
K1(T)	32	75.907	73.761	**73.593**	86.532	86.985
K1(F)	32	24.054	22.907	**21.343**	21.385	25.651
K2(T)	5	129.885	104.247	105.63	103.763	109.35
K2(F)	10	40.793	39.956	51.670	40.119	50.234
K3(T)	5	980.849	849.206	845.075	842.143	875.417
K3(F)	10	345.88	345.296	333.389	361.69	341.983
K4(T)	4	218.244	168.593	150.256	130.614	117.129
K4(F)	12	86.073	74.858	56.519	49.868	44.868
K5(T)	10	42.763	33.796	33.165	33.524	33.505
K5(F)	12	18.570	13.652	**12.014**	12.806	12.643
K6(T)	10	44.224	31.396	31.695	31.681	31.601
K6(F)	12	18.001	12.621	**10.938**	11.729	12.148
K7(T)	2	30.706	23.868	**23.362**	24.538	24.261
K7(F)	8	12.162	10.173	**8.985**	9.361	9.259
		Execution time for kernels for varying #threads/block when data is in ***SHARED*** memory				
		32	64	128	256	512
K2sh(T)	4	26.011	**23.883**	23.936	25.114	25.394
K2sh(F)	8	21.001	19.905	**10.363**	10.450	11.164
K3sh(T)	4	149.61	142.946	**113.348**	122.57	130.503
K3sh(F)	8	50.903	47.363	**30.918**	32.221	36.068
K4sh(T)	4	54.469	51.107	50.842	**29.046**	29.781
K4sh(F)	8	29.882	25.018	22.991	**15.191**	16.209
K5sh(T)	8	33.258	**32.979**	33.696	36.405	36.949
K5sh(F)	13	14.756	13.207	13.081	14.024	13.846
K6sh(T)	8	31.254	**30.651**	33.121	34.692	36.089
K6sh(F)	10	14.430	12.684	12.244	13.519	13.162
K7sh(T)	3	27.2328	25.858	25.003	26.880	26.485
K7sh(F)	8	13.043	10.909	9.9881	10.272	10.382

Table I
PERFORMANCE OF GLOBAL AND SHARED MEMORY IMPLEMENTATIONS OF KERNELS ON A TESLA (T) AND FERMI (F) GPU FOR 10000 MOLECULES. OPTIMAL PERFORMANCE IS SHOWN IN **BOLD** AND THE CORRESPONDING KERNEL IMPLEMENTATION IS IN *ITALICS*.

Kernel	$\mathcal{PS}_{opt}(K$,Tesla)			$\mathcal{PS}_{opt}(K$,Fermi)		
	Reg	TB	Mem	Reg	TB	Mem
K1	32	128	G	32	128	G
K2	4	64	S	8	128	S
K3	4	128	S	8	128	S
K4	4	256	S	8	256	S
K5	10	64	S	12	128	G
K6	8	64	S	12	128	G
K7	2	128	G	8	128	G

Table II
PERFORMANCE SIGNATURE OF KERNELS ON A TESLA AND FERMI GPU FOR 10000 MOLECULES. S(DATA RESIDES IN SHARED MEMORY), G(DATA RESIDES IN GLOBAL MEMORY)

$\mathcal{PS}_{opt}$ for these kernels in Tesla and Fermi GPUs for 10000 molecules.

Note that the *shared memory* versions of K5 and K6 perform optimally for Tesla, but for Fermi, it is the *global memory* versions of these two kernels that perform the best.

VI. DESIGN CHOICE EXPLORATION- EXPERIMENTS BY APPLICATION DEVELOPER

Using performance signatures and implementations of K1$\cdots$K7 from kernel developers, we now study the efficacy of various design strategies introduced in Section III-B, and present the performance of the overall application.

A. Merge Strategies

Table III shows the performance of different combinations of kernels and different merge strategies. For K5, K6, and K7, we have merged K5 and K6 (K56), then K6 and K7 (K67) and finally K5, K6 and K7 (K567). For each combination, we have used three merge strategies introduced in Section III-B. While merging, we have taken both global and shared memory implementations of these kernels. To distinguish, we have used the suffix "Mg" to denote global memory version and "Msh" to denote the shared memory version of the merged kernel.

The table is divided into four parts. The first part shows the execution time of the best performing versions of K5, K6 and K7 as per their $\mathcal{PS}_{opt}$, shown in Tables I, and II. We have not run our experiment for block size 32 in case of the 2nd and 3rd kernel merging strategies because that would certainly lead to a branch divergence within a warp leading to poor performance.

Optimal Merging: Table III shows that all strategies improve performance. We find that when the global memory implementation of K5 and K6 are merged together using merge strategy 1, the merged kernel K56Mg1 gives the best performance on Tesla as well as on Fermi GPU.

Next, from Table III, we observe that when the shared memory versions of K6 and K7 are merged, the merged kernel K67Msh gives the best performance on Tesla. However, when the global memory version of K6 and K7 are merged, the merged kernel gives optimal performance on Fermi. Once again, strategy 1 gives the best performance. The best performance is obtained when K5, K6, K7 are merged all together using strategy 1, as compared to merging two kernels at a time.

Register Usage: Register usage is an important factor in merging kernels, because the merged kernel has a higher register usage than either kernel. Some previous works [9], [10] have assumed that merged kernels have register usage which is the sum of the register usage of each individual kernel. However, a comparison of the register usage of the original kernels and the merged kernels from their performance signatures above shows that the register usage for the merged kernels is much less than the sum.

1) Analysis of Merging K5 and K6: The merged kernel having its data in global memory performs better than merged kernels with data in shared memory in this case. The version using merging strategy 1, K56Mg1, is 38% faster than when optimally tuned K5sh and K6sh run in sequence one after another on a Tesla GPU. On a Fermi GPU, this merged kernel is 40% faster compared to optimally tuned K5 and K6 executed in sequence. While merging of kernels has an obvious advantage of kernel launch overhead reduction (reported in Sec. IV-A), we observe the following additional performance benefits in favor of global memory. Through

Kernel	#reg./ thr.	Exec. time(μs) for kernels for varying #thr/ blk				
		32	64	128	256	512
K5sh(T)	8	33.258	32.979	33.696	36.405	36.949
K5(F)	12	18.570	13.652	12.014	12.806	12.643
K6sh(T)	8	31.254	30.651	33.121	34.692	36.089
K6(F)	12	18.001	12.621	10.938	11.729	12.148
K7(T)	2	30.706	23.868	23.362	24.538	24.261
K7(F)	8	12.162	10.173	8.985	9.361	9.259
Merging K5 and K6						
K56Mg1(T)	14	42.426	40.134	**39.691**	40.640	40.879
K56Mg1(F)	18	16.603	13.873	**13.727**	14.107	14.237
K56Msh1(T)	13	42.859	40.614	40.055	41.035	41.382
K56Msh1(F)	16	17.333	14.744	14.069	14.737	15.13
K56Mg2(T)	13	NA	54.820	54.173	55.071	56.008
K56Mg2(F)	16	NA	15.233	14.694	15.918	17.946
K56Msh2(T)	13	NA	54.621	54.596	55.059	56.445
K56Msh2(F)	18	NA	14.812	14.731	17.569	19.161
K56Mg3(T)	10	NA	44.641	44.282	45.480	45.166
K56Mg3(F)	12	NA	16.359	14.311	15.615	15.793
K56Msh3(T)	10	NA	50.421	49.884	50.659	50.676
K56Msh3(F)	14	NA	14.965	14.718	15.566	16.180
Merging K6 and K7						
K67Mg1(T)	11	37.126	36.382	36.124	37.286	37.776
K67Mg1(F)	14	13.868	11.760	**11.438**	11.565	12.108
K67Msh1(T)	11	36.804	35.379	**35.361**	37.367	37.776
K67Msh1(F)	13	15.133	12.375	11.843	12.476	12.726
K67Mg2(T)	10	NA	46.961	47.986	48.181	47.750
K67Mg2(F)	14	NA	13.375	12.310	12.503	14.419
K67Msh2(T)	11	NA	44.915	45.827	45.724	45.859
K67Msh2(F)	16	NA	13.348	12.841	13.367	14.317
K67Mg3(T)	10	NA	37.148	36.848	37.858	37.573
K67Mg3(F)	12	NA	13.964	12.193	12.394	13.014
K67Msh3(T)	10	NA	37.494	37.528	38.093	38.113
K67Msh3(F)	12	NA	13.419	12.675	13.289	13.885
Merging K5, K6 and K7						
K567Mg1(T)	14	49.791	48.168	47.844	50.998	50.292
K567Mg1(F)	18	18.840	15.351	14.203	15.426	15.524
K567Msh1(T)	13	48.363	**47.267**	47.681	50.122	50.297
K567Msh1(F)	20	17.033	14.573	**14.084**	15.125	15.282

Table III
PERFORMANCE RESULTS AFTER MERGING KERNELS K5, K6, AND K7 A TESLA (T) AND FERMI (F) GPU FOR 10000 MOLECULES. THE FINAL SUFFIX DENOTES THE MERGE STRATEGY USED. OPTIMAL PERFORMANCE IS SHOWN IN **BOLD**

CUDA Visual Profiler we have found that storing data in the shared memory does not bring down the number of global memory load instructions significantly for K56Mg1. This implies that there is not enough data to be re-used when K5 and K6 are merged. Furthermore, a Fermi GPU caches global memory reads. Using a CUDA Visual Profiler, we have seen that the ratio of L1 cache hits to misses for K56Mg1 is higher than that of K56Msh1.

Next, we have investigated the reason for strategy 1 to work better here. To begin with, K5 and K6 are not computation-heavy. To ensure that K5 and K6 run in different warps, the application developer has to write extra operations in the merged kernel involving modulo operations having a low instruction throughput. CUDA Visual Profiler has shown that additional number of instructions added due to such operations is significantly high in case of strategy 2 and 3, compared to the strategy 1. Next, recall as per strategy 2 (Fig 2c), K56Msh2 performs double the computations per thread. This leads to two threads of the same half warp on a Tesla GPU and two threads in the same warp on a Fermi GPU to access the same memory bank, resulting in bank conflicts and thus degrading the performance of this merged kernel. Finally, as per strategy 3 (Fig 2d), the number of blocks executed by a merged kernel is twice than that created by strategy 1. Doubling the number of thread blocks adds to the thread creation overhead, which makes an impact on performance because K5 and K6 are not computationally intensive.

2) Analysis of Merging K6 and K7: Table III shows that at 128 threads per block (T/B) on the Tesla GPU, K67Msh1 gives the best performance and is 37.4 % faster than when optimally tuned K6sh (64 T/B)and K7 (128 T/B) are executed in sequence. However, on the Fermi GPU, the best performing merged kernel K67Mg1 runs 42.57% faster than optimally tuned K6 and K7. On both the Tesla and the Fermi GPU, the reasons for kernels merged using strategy 1, to perform better than those kernels merged using strategies 2 and 3; are similar to the reasons that we had discussed earlier for K56Mg1 and K56Msh1.

Unlike merging of K5 and K6, the reason for K67Msh1 where the data resides in the shared memory, to perform marginally better than K67Mg1 is as follows. Using CUDA Visual Profiler, we have found that though K67Mg1 executes 10% fewer instructions than K67Msh1, the number of global memory load instructions for K67Mg1 is 25% more than K67Msh1. So any performance improvement obtained by K67Mg1 by executing fewer instructions than K67Msh1 is offset by a performance degradation due to more global memory access. This leads to a marginal performance improvement for K67Msh1 over K67Mg1.

On a Fermi GPU we find the reverse. Using CUDA Visual Profiler, we have found that K67Mg1 executes 12% lesser number of instructions compared to K67Msh1. The number of global memory load instructions for K67Mg1 is 25% more than K67Msh1. The ratio of L1 cache hits to misses for K67Mg1 is higher than K67Msh1. Hence we see that i) *a higher ratio* of L1 cache hits, combined with ii) *lesser number* of instructions to execute, offsets any performance degradation due to more global memory access by K67Mg1. *This leads to a marginal performance improvement for K67Mg1 over K67Msh1.*

3) Analysis of Merging K5, K6 and K7: Table III shows that kernel K567Msh (data in shared memory) gives a better performance consistently than K567Mg. At 64 threads per block (T/B), K567Msh is 45.66% faster than when optimized version of K5sh, K6sh (both at 64 T/B) and K7 (128 T/B) are run in sequence on the Tesla GPU. On the Fermi GPU and at 128 T/B, K567Msh is 57% faster than when optimized K5, K6 and K7 are run in sequence. The result is quite expected as there is enough reusable data in the merged kernel and hence storing such data in the shared memory hides global memory latency better; the shared data in this case is *twice*

Kernel	Exec. time(μs) for kernels for varying #thr/ block				
	32	64	128	256	512
K2sh	21.001	19.905	10.363	10.450	11.164
K3sh	50.903	47.363	30.918	32.221	36.068
K23Psh	60.514	58.607	**37.290**	38.333	42.806

Table IV
PERFORMANCE COMPARISON BETWEEN K2 AND K3 AND CONCURRENT EXECUTION OF THE SAME KERNELS (K23PSH) ON A FERMI GPU FOR 10,000 MOLECULES. OPTIMAL PERFORMANCE IS IN **BOLD**

Kernel	Exec. time(μs) for kernels for varying #thr/ block				
	32	64	128	256	512
K7, i^{th} itr	12.162	10.173	8.985	9.361	9.259
K1, (i+1)th itr	24.054	22.907	21.343	21.385	25.651
K71P	28.643	26.253	**23.939**	24.297	28.109

Table V
PERFORMANCE COMPARISON BETWEEN K7 AND K1 AND PSEUDO PARALLEL EXECUTION OF THE SAME (K71P) ON A FERMI GPU FOR 10,000 MOLECULES. OPTIMAL PERFORMANCE IS IN **BOLD**

the amount of shared data in the K5-K6 or K6-K7 merger cases.

4) Merge Strategies: Merge strategies 2 and 3 too lead to significant improvement in performance over the unrefactored code, even though strategy 1 performs better than them with in the application we have considered here. We can expect them to perform better than strategy 1 in other applications. For example, strategy 3 has greater parallelism and so can potentially hide global memory access latency better than strategies 2 or 3. However, in this particular application, the extra computation for the modulus proved to be a significant overhead because the amount of computation was very small.

B. Concurrent Execution of Kernels

As discussed in Section III-B2, it is possible to run kernels K2 and K3 concurrently since they don't have any data dependencies. It is also possible to interleave the execution of i^{th} time-step of K7 with $(i+1)^{th}$ of K1 as they don't have dependencies either. As Tesla GPU does not support concurrent kernel execution, we present the results from the next set of experiments only for the Fermi GPU.

1) Concurrent execution of K2 and K3: Both K2 and K3 are data independent kernels that execute the same number of thread blocks for a given number of molecules. Table IV shows the performance comparison of concurrent kernel K23Psh with that of standalone kernels K2 and K3 that have been implemented with the data in shared memory.

Clearly, at 128 threads/block K23Psh performs 9.67% better over the standalone kernels when executed one after another. We explain the reason later.

2) Executing kernels K7 and K1 in parallel: For any given iteration, we execute kernel K7 of that iteration with K1 of the next iteration. Let K71P represent such a pseudo kernel whose performances have been shown in Table V. In this table we compare the performance of K71P with that of standalone kernels K7 and K1 of the next time-step, both of which have been implemented with their data in global memory. At 128 threads per block, K71P performs 21% better over the standalone kernels when executed one after another.

Each of these four kernels (K1, K2, K3 K7) is able to run 8 blocks per SM. In our experiment, for 10,000 molecules each of these kernels execute 79 blocks leading to 67% GPU occupancy. Thus, with CUDA stream based concurrent execution, execution of kernels K2 and K3 are overlapped as also that of K7 and K1 with pseudo parallelism, leading to a better GPU utilization and overall performance improvement over sequential execution of kernels.

C. Optimal Design of the Application

Based on performance results of various merge and concurrent scheduling strategies, the application designer can now build a complete application. From the complete application perspective, we still have two design choices. We have to see if merging of K5, K6, K7 is better than merging K5, K6 and then execute K7, K1 concurrently. We have compared execution times[4] of three versions of the application using Unix wall clock time on the Fermi GPU, as discussed below.

1) The first version uses optimized kernels, K1, K2sh, K3sh, K4sh, K5, K6, and K7 without any further design optimization. The application takes 12.442s.
2) The second version uses K1, K23P, K4sh, K567Msh (data in shared memory) as per the optimal design choice. The application takes **9.243**s.
3) The third version uses K1, K23P, K4sh, K56Mg1 (data in global memory), K7K1P. The application takes 9.368s.

The reason why choice 3 is not as good as 2 is because K7 takes $O(N)$ while K1 takes $O(N^2)$, where N is the number of molecules. As a result, we do not get the desired performance benefit when we run them in parallel. Overall, the design choice 2, which is the optimal composition, gives us a 25.7% improvement compared to the trivial composition of optimized kernels in choice 1.

VII. RELATED WORK

Several papers [11], [12], [13], [14], [15] discuss porting applications to GPUs and improving performance through an optimal assignment of architectural parameters to achieve overall execution efficiency. Our goal, in contrast, deals with a design methodology that improves performance of an application that consists of a composition of kernels. Merging kernels plays an important role in this. Merging kernels has been considered by a couple of other works. Guevara et.al. [9] consider scheduling independent kernels, corresponding to independent jobs, on a pre-Fermi GPU. Due to the absence of concurrent execution on that GPU, they merge the kernels such that different blocks perform

[4]The number of iterations was fewer than in a production run

the work of different kernels. Wang et.al. [10] use merging to optimize power consumption by GPU. While both the above works in [9], [10] merge kernels, the main purpose of their work is quite different. In our work, we perform an experimental study on composition of kernels and their impact on the performance, which involves different issues, such as better performance through reuse of data in shared memory. Of course, the design methodology that we present is also an important focus of our work.

VIII. Conclusion

In this paper we have studied the composition of GPU kernels, and subsequent refactoring for performance tuning to build an application. We have proposed a design methodology where kernel developers focus on optimal kernel development and provide performance signatures of a kernel alongwith different versions of implementations for subsequent reuse. From an application developer's perspective, we have analyzed various kernel merging techniques and have shown under what circumstance a particular technique performs the best. We have also analyzed two different ways to schedule kernels concurrently when there is no data dependency between them. Finally we have shown how an application developer selects the best design strategy to compose a complete application.

We notice from our study that standalone kernels might be tuned to give an optimal performance using shared memory or by using higher number of threads per block so as to hide global memory latency better. But when refactoring techniques such as kernel merging is applied to such optimized kernels, sometimes sub-optimal strategies when applied to the merged kernel, give us a better result. We observe that in certain cases the merged kernel produces a better performance when either the data is moved to the global memory (in those cases where there is not much scope for data to be re-used in the merged kernel), or for smaller number of threads per block or both. In a different context, the authors [16] had shown that a sub-optimal data layout for DFTs could help the application, on the whole, to perform better. The work present here further substantiates the argument that when designing a complex application, optimal choices may not correspond to making optimal choices for each individual component separately.

Our work can be extended in several directions. Our eventual goal is to incorporate different design decisions that we have described here, into a design tool, where a part of the decision selection strategy can be automated. We intend to investigate if the attributes from a performance signature is sufficient to predict the performance when the kernel is reused in a larger application context. Another worthwhile extension of our work could be partial automation of application optimization through the composition of kernels.

References

[1] D. Patterson and J. Hennessy, *Graphics and Computing GPUs: Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2009.

[2] K. Esler, J. Kim, L. Shulenburger, and D. Ceperley, "Fully Accelerating Quantum Monte Carlo Simulations of Real Materials on GPU Clusters," *Computing in Science and Engineering*, vol. 99, no. PrePrints, 2010.

[3] S. Ryoo, C. I. Rodriguesy, S. S. Baghsorkhiy, S. S. Stoney, D. Kirk, and Wen-Mei W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proc. ACM SIGPLAN Symp. on PPoPP*. 2008.

[4] G. T. Heineman and W. T. Councill, *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] D. Kirk and Wen-Mei W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[6] D. Parnas, P. Clements, and D. Weiss, "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, pp. 259–266, 1985.

[7] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," in *Proc of the ACM SIGPLAN Symp. on Compiler construction*, 1986, pp. 162–175.

[8] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence Analysis for Pointer Variables," in *ACM SIGPLAN Conf. on PLDI*, 1989, pp. 28–40.

[9] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling Task Parallelism in the CUDA Scheduler," in *Proceedings of the Workshop on PMEA*, 2009, pp. 69–76.

[10] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion : an Effective Method for Better Power Efficiency on Multithreaded GPU," in *IEEE/ACM Intl Conf. on Green Computing and Communications*, 2010, pp. 344–350.

[11] 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 *Proc. ACM SIGPLAN Symp. on PPoPP*. 2008.

[12] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan., "A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs," in *Proc. International Conference on Supercomputing*, 2008.

[13] S. Ueng, M. Lathara, S. S. Baghsorkhi, and Wen-Mei W. Hwu, "Cuda-lite: Reducing GPU Programming Complexity." in *Proc. Workshops on Languages and Compilers for Parallel Computing*, 2008.

[14] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and Wen-Mei W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPUs." in *IEEE/ACM Intl Symp. on Code Generation and Optimization*. 2008.

[15] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *ACM SIGPLAN Conference on PLDI*. 2010.

[16] S. Mitra and A. Srinivasan, "Small Discrete Fourier Transforms on GPUs," in *11th IEEE/ACM Intl Symp. on CCGrid*, 2011, pp. 33–42.