A Grouped Genetic Algorithm for Optimizing GPU Kernel Fusion

Technical Report TR-001-04-14
April 2014

Mohamed Wahib, Naoya Maruyama

RIKEN Advanced Institute for Computational Science
Kobe, Japan 650–0047
Email: {mohamed.attia,nmaruyama}@riken.jp

Abstract

Computationally demanding scientific applications designed for GPUs can include tens of kernels and data arrays. Transforming the code of those kernels, using kernel fusion, to expose locality can speedup the applications by reducing off-chip memory traffic. The best fusions are the ones that maximize the exposed locality to reduce off-chip memory traffic. Kernel fusion is approached in this report as a combinatorial optimization problem that requires searching the space of possible kernel fusions. This report proposes a grouped genetic algorithm to solve the kernel problem presented as a variation of the bin packing problem. The proposed algorithm adapts the genetic operators, constraint handling and population initialization to cater for the specific features of kernel fusion. Experimental evaluation using a test suite of benchmarks show high fidelity of the proposed algorithm. Moreover, by implementing fusions suggested by the proposed algorithm, actual speedups of 1.2x and 1.35x for two real-world applications were achieved.

1. Introduction

Real-world applications written for GPU often have tens of kernels[1]. Those kernels share the use of data arrays that have different intentions; input, input-output and output. For some categories of applications, e.g. weather models, most of those kernels are memory-bound due to using finite difference methods. Memory-bound kernels are bound by the bandwidth of memory rather than the processing speed due to a low number of operations (FLOPS) compared to the amount of data (Bytes) operated upon.

GPUs, as well as CPUs, have a memory hierarchy, i.e. memory that is on-chip and have high bandwidth but low capacity vs. memory that is off-chip and have lower band width but higher capacity. A standard practice in High Performance Computing (HPC) is to maximize the use of on-chip memory when possible and minimize the use of off-chip memory. This can be applied on a single kernel level by using on-chip shared memory (SMEM) to reduce the traffic to device global memory (GMEM). With the SMEM bandwidth an order of magnitude higher than that of device global memory (GMEM) in Nvidia GPUs, the performance can potentially improve as the total FLOPS to byte ratio increases.

A potential for improving performance by reducing traffic to GMEM for applications with many kernels exists. This potential can be realized by *kernel fusion*. More specifically, by rearranging kernel calls from the host side such that calls to kernels using the same data array(s) are placed in affinity, they can then be replaced with a single call to a new kernel that aggregates the code segments of the kernels. Such code transformation, i.e. *kernel fusion*, exposes an opportunity to use the on-chip memory (SMEM) to hold the data in-between instructions originally coming from different kernels.

However, identifying which kernels to fuse from tens of kernels is not a trivial task. The search process must consider the complex data dependencies for arrays different in size and access patterns to maintain the order-of-execution in permutations of possible kernel fusions. Moreover, there is the challenge of accurately accounting for the implications by the features specific to the GPU architecture. Architectural constraints such as the number of registers, the capacity of shared memory, and the number of shared memory banks can have significant performance implications in resulting fused kernels, thereby severely limiting the effectiveness of non-architecture-aware algorithms.

[1] A kernel is a function executed on the GPU and invoked by the host CPU

To that end, kernel fusion was formulated in previous work [20] as a combinatorial optimization problem. More specifically, mapping N original kernels to M fused kernels to reduce the memory operations can be described as a variation of the classical *Bin Packing* problem. In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers, each having a pre-defined volume, in a way that minimizes the number of bins used. The problem in hand is similar as kernels of different performances are to be fused into kernels that have a limit on upper bound performance, i.e. a fused kernel should not have more execution time than the sum of those of the kernels which were fused.

To account for the multivariate dependencies, i.e. the data dependencies of kernels fused together. This report proposes a Grouped Genetic Algorithm (GGA) to search the exponentially proportional space of possible fusions. We derive the optimization problem by constructing a data dependency graph and order-of-execution graph for the kernels in the program. The optimization algorithm uses a lightweight and accurate projection method of performance upper bound as an objective function of candidate solutions. This lightweight method is essential to enable fusions for applications with a large number of kernels and data arrays. Finally, the best result of the optimization algorithm can be used to generate the new fused kernels manually or be further improved by an automated source-to-source code transformation.

The main contribution in this report is designing and implementing an adapted grouped genetic algorithm to search for near-to-optimal solutions in the space of possible kernel fusions. Experimental evidence of the quality and effectiveness of the optimization algorithm when applied to a test suite and two real world weather applications with tens of kernels and data arrays is also presented. As was shown in previous work [20], the introduced kernel fusion method was able to identify the optimal kernels to fuse for the weather applications within a reasonable amount of time. The actual kernel fusion transformation resulted in more than 1.35x and 1.2x speedup.

2. Background

This section gives a brief background about GPUs and their execution model. GPUs, and accelerators in general, has attracted much attention in HPC in the last decade. Most computationally demanding areas, e.g. weather and climate modeling, have experimented, at different levels, with relying on GPUs to handle heavy workloads (e.g. [19, 18, 17, 8, 2, 1]). Although GPUs can offer unprecedented

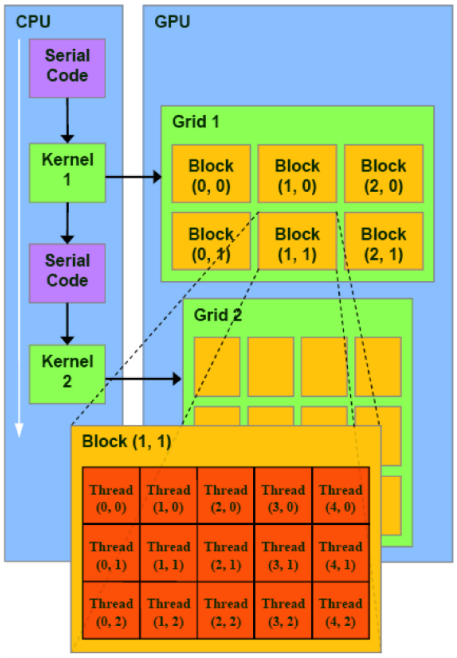

Figure 1: CUDA execution model (with permission of Nvidia)

performance gain, implementation of an algorithm over a GPU to take full advantage of this technology involves a significant complexity of parallelizing across the multiple cores. Figure 1 shows the execution model of GPU using Nvidia's CUDA. The GPU is connected to the CPU over the PCI express. The interconnect is expected to be changed in the future as well as the GPU-CPU proximity. Yet the execution model of a CPU invoking kernels on the GPU remains the same. On the GPU side, the kernels minimum execution unit is a thread. Threads are arranged to thread blocks which are themselves organized into a grid. Details about the mapping of kernels to CUDA are skipped in this report for simplicity. The important point relating to the work in this report is the execution model itself.

Memory management over a GPU makes things even more challenging. Figure 2 shows the memory hierarchy for Nvidia K20X which is generally the same for all Nvidia GPUs. Frequently moving data across the PCI express between the host memory and device global memory can become a bottleneck. Hence, a standard practice is to offload the entire computation to the GPU and have the data arrays reside the lifetime of the program on the global memory. Data allocated in the shared memory have the lifetime of the kernel and is not coherent with global

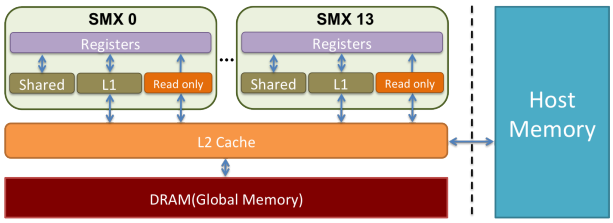

Figure 2: Nvidia K20X memory hierarchy (with permission of Nvidia)

memory. Moreover, threads in a thread blocks have no access to shared memory used by a different thread block. Therefore, thread blocks should be designed to use either the shared memory allocated for the block or the global memory.

3. Problem Description

Kernel fusion problem description and formulation was introduced in pervious work [20]. However, we summarize the important points, i.e. how data arrays are used in kernels, types of dependency between kernels and formulation, to ease the flow of reading. Figure 3 shows the data dependency graph extracted from a routine in SCALE-LES [19], a next-generation weather model. There are generally four intentions for data arrays:

Input-only arrays: The most common form. Redundancies in such arrays are easily exposed in kernels sharing the use of the same arrays. Exploiting such arrays do not add implications and is mostly bound by the capacity of the SMEM.

Strict input-output arrays: Can be reused if both the kernels that read and write into the arrays are fused together. A barrier would be required between the operations that store and those that load the data array into the SMEM. Additionally, incoherency will occur after the barrier; the edge threads in a thread block have no access to SMEM allocated to other thread blocks and SMEM is incoherent with GMEM.

Expandable input-output arrays: Are also input-output arrays. Yet several kernels write them into. So they would enforce an order-of-execution for all kernels writing to the same array. Changing the kernels to write into redundant arrays and adjusting accordingly would relax the dependences by removing the order-of-execution constraint and changing the new arrays to be of the strict type. Note that this comes at the expense of GMEM capacity.

Output-only arrays: That can not be reused. The data dependency graph of the kernels in an application reveals the intention for arrays used by kernels. From the

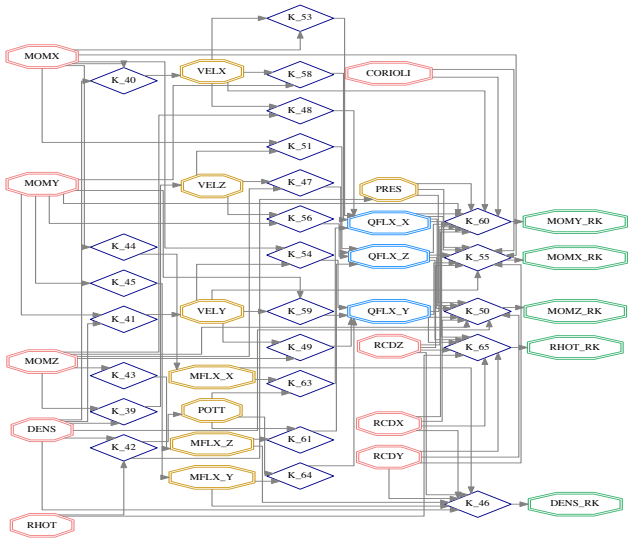

Figure 3: Data dependency graph for the RK routine. Diamonds are Kernels invoked in the order of the numbered label, input-only arrays red octagon, output-only arrays green octagon, expandable input-output arrays blue and strict input-output arrays yellow octagon

dependency graph, an order-of-execution graph for the kernels can be constructed. Note that this graph would be descriptive of the order-of-execution and is not a call graph, i.e. the host invokes all kernels. The precedencies extracted from data dependencies are relaxed by using redundant arrays to change input-output arrays from expandable to strict. The opportunity of improving performance becomes apparent when the order of kernel calls is changed without violating the order-of-execution graph. Hence, allowing fusion of affine kernels to expose intra-kernel locality of data arrays.

Table 1 includes a list of terms that will be frequently used from this point on. In addition to that, the following is a list of architecture-independent constraints when searching for an optimal fusion plan for *original kernel* in an application. First, a kernel can appear in different *sharing sets*. However, an original kernel should appear in only one fused kernel. Resolving the mapping of original kernels to fused kernels is complicated as the other kernels in the *sharing sets* have an effect on the decision. Second, arrays with a low cardinality of *sharing set* can yield better performance if becoming a *kernel pivot* than with higher cardinality if the arrays in the *sharing set* have an order-of-execution. Third, the *original kernels* are not equal in the amount of computation and data operations. Therefore,

Table 1: Summary of terminology related to kernel fusion used in this report

Term	Meaning
Original kernel	A kernel in the original GPU implementation before applying fusion
Fused kernel	A kernel generated from fusing two or more original kernels
Original sum: F^{Σ}	The summation of the measured runtimes of the original kernels that were fused into F
Shared array: $\langle D \rangle$	Array D that is touched by at least two kernels
Sharing set: $\mathbb{K}(D)$	The set of kernels sharing the use of array D
Thread load: $D \xrightarrow{T} K$	Average number of threads in a thread block of kernel K that access the same data element in array D
Kernel pivot: F^{Pivot}	The set of the data array(s) that now have data reuse via SMEM in fused kernel F
Degree of kinship: $(K_1, K_n)^{\circ}$	= n-1, when a chain of kernels having shared data array(s) between k_1 and k_n exists = 0, otherwise

combinations of different kernels to be fused together are affected variably by the limited SMEM capacity and other constraints.

3.1. Kernel Fusion as an Optimization Problem

This section introduces the reasoning of kernel fusion as a combinatorial optimization problem. The search for the optimal solution for the kernel fusion problem is a permutation. More specifically, mapping N original kernels to M fused kernels to reduce the memory operations can be described as a variation of the classical *Bin Packing* problem. In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers, each having a pre-defined volume, in a way that minimizes the number of bins used. The problem in hand is similar as kernels of different performances are to be fused into kernels that have a limit on upper bound performance, i.e. a fused kernel should not have more runtime than the *original sum*. The differences from the classical bin packing are:

- The objective function in the classical case is to minimize the total number of bins. In kernel fusion, the objective function is minimizing the aggregated performance of the fused kernels, i.e. the number of bins (fused kernels) is irrelevant.

- There is no restriction in the original bin packing problem on which items to pack together. Contrarily, only kernels with a *degree of kinship* ≥ 1 can be fused.

- Bins in the bin packing problem have predefined volumes. In the kernel fusion problem, bins (the newly fused kernels) are created on-the-fly and their volume (upper-performance bound) is dynamically calculated from the original kernels.

3.2. Problem Formulation

For a set D of l arrays shared over the set K of n kernels having runtimes of $P_1, ..., P_n$ that are to be fused into set F of m kernels with projected upper bound performance resulting in runtime $T_1, ..., T_m$, find the m-partition $K_1 \cup ... \cup K_m$ of the set $\{1, ..., n\}$ such that: **a)** Each fused kernel should have lower runtime than the *original sum*, **b)** Each kernel is only fused once, **c)** If two kernels k_a and k_b are fused into the same kernel f and a path exists from k_a to k_b in the dependency graph of array d_l, then all kernels in the path from k_a to k_b must also be fused to f, i.e. preserve the order-of-execution, **d)** No fused kernel should exceed the capacity of on-chip memory per SMX[2] nor the maximum registers allowed per thread, **e)** Every kernel must have a *degree of kinship* more than zero with all other kernels fused with it. A solution is optimal if it has the minimum $\sum_{j=1}^{m} T_j$. Note that for $h \in \{1, ..., m\}$, the T_h value of F_h is a dynamic value, i.e. the value is calculated depending on the subset K_h that is fused to F_h. The canonical form of the problem is:

Input	***Output***
Original kernels $K, \|K\| = n$	Fused kernels $F, \|F\| = m$
Data arrays $D, \|D\| = l$	

Minimize

$$\sum_{j=1}^{m} T_j \tag{1}$$

[2]SMXs are multiprocessors in Nvidia terminology.

Subject to

$$\sum_{i \epsilon F_k} P_i > T_k \tag{1.1}$$

$$\sum_{j=1}^{m} x_{ij} = 1, \forall i \in \{1, ..., n\} \tag{1.2}$$

$$x_{qr} = 1, \forall q \in \overline{k^{ab}}, x_{ar} = 1, x_{br} = 1 \tag{1.3}$$

$$x_{ij} \in \{0, 1\}, \forall i \in \{1, ..., n\} \forall j \in \{1, ..., m\} \tag{1.4}$$

$$DegKin\,(k_x, k_y) > 0, x \neq y, \forall k_x \forall k_y \in F_k, \forall F_k \in F \tag{1.5}$$

$$MEM\,(F_j) \leq Capacity, \forall j \in \{1, ..., m\} \tag{1.6}$$

$$REG\,(F_j) \leq R_{Max}, \forall j \in \{1, ..., m\} \tag{1.7}$$

Where

$x_{ij} = 1$ if kernel k_i is fused to kernel F_j
$\overline{k^{ab}} =$ the set of kernels in the path from k_a to k_b
in the dependency graph of array d_l
$DegKin\,(a, b)$ returns *Degree of kinship* between kernels a, b
$MEM\,(k)$ returns on-chip memory used by kernel k
$Capacity$ is the device on-chip memory capacity
$REG\,(k)$ returns registers per thread used by kernel k
R_{Max} is the maximum registers allowed per thread

4. Proposed Algorithm

4.1. Avoiding redundancy

The offline bin packing and similar derived problems have a prohibitive cost of solving in large problem instances using exhaustive search. SCALE-LES for example has over a hundred of kernels sharing the use of dozens of arrays. The exhaustive search is estimated to have ∼2.6e45 feasible solutions; every possible permutation of kernel fusion, making the exhaustive search prohibitive.

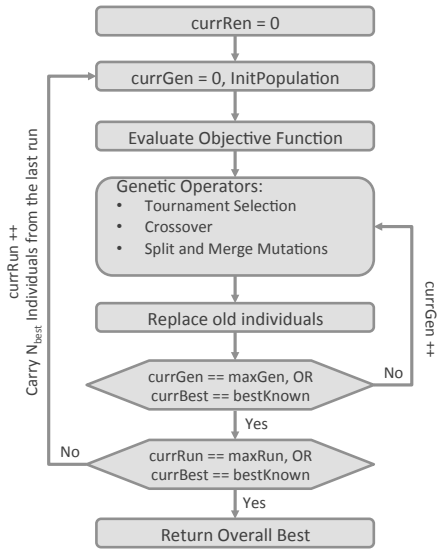

Figure 4: Grouped Genetic Algorithm

A commonly used polynomial-time approximation algorithm [10] for bin packing depends on the first fit decreasing strategy which first sorts the items to be inserted in decreasing order by their sizes, and then inserts each item into the first bin in the list with sufficient remaining space. In the kernel fusion problem, deciding for a criterion to sort the kernels is problematic because the contribution of an original kernel in the performance of a kernel it is fused into is not known beforehand. Thus, a single metric to sort kernels, in analogy to size in the bin packing problem, does not exist. Accordingly, a different approximation algorithm is needed. To decide on the heuristic to use, a specific feature of the kernel fusion problem should be taken into consideration: strong relationship or dependency exists among the decision parameter, i.e. data dependencies among original kernels. Hence a solver able to capture the multi dependencies is the most significant factor.

In this report, a Grouping Genetic Algorithm is used to solve the kernel fusion problem (Figure 4). The choice of GGA is motivated by its proven ability [15, 12] to account for dependencies between decision variables. The concept of a Grouping Genetic Algorithm was first introduced by Falkenauer [4]. Falkenauer pointed out that the application of standard item-oriented encoding schemes (e.g. integer encoding or random key-based encoding) to grouping problems would result in two main problems. One is the large proportion of redundancy in the popula-

tion, due to the fact that different chromosomes encode exactly the same solution. The second problem concerns the context insensitivity of the crossover operators, i.e. important information from the chromosome gets lost or is transferred incorrectly. Both aspects will deteriorate the successful application of genetic algorithms to grouping problems. In order to overcome these drawbacks, Falkenauer suggested an unconventional chromosomal representation of solutions that reflects the structure of grouping problems much better. The following section illustrates the unconventional encoding scheme used in GGA along with the genetic operators customized for this encoding.

4.2. Encoding and Genetic Operators

GGAs have an unconventional group-oriented chromosomal representation of solutions that reflects the structure of grouping problems much better. For the kernel fusion problem, this translates into the fact that in a GGA the genes of the chromosome should represent fused kernels instead of original kernels, as it is the case in the standard item-oriented encoding scheme. This implies that the chromosomes can be of variable length. A group-oriented encoding scheme for the kernel fusion, as it is used in our algorithm, is depicted exemplarily for eight original kernel in Figure 5. It shows that original kernels 3 and 7 are fused to one kernel, original kernels 1 and 4 in another one, and so on.

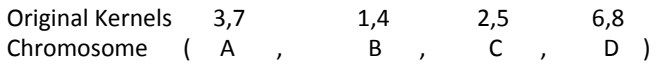

Figure 5: Representation of a solution based on a group-oriented encoding scheme

4.2.1. Crossover

Apart from the encoding scheme, the crossover operator in the GGA is group oriented, too. For this reason the classic GA crossover operator has to be modified for being included in the GGA. Based on [3], we suggest the following procedure:

1. Randomly select two parental chromosomes from the population.
2. Randomly select a crossing section in each of the chromosomes (Figure 6-a).
3. For each of the two chromosomes, insert the genes (fused kernels) of the crossing section of one chromosome into the respective other one. The new genes (fused kernels) will be inserted in front of the genes of the crossing section (Figure 6-b).

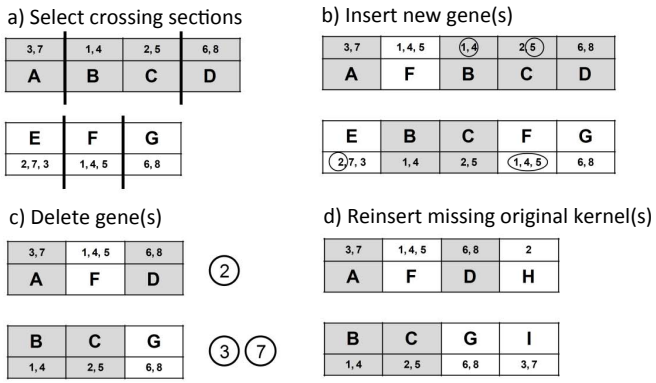

Figure 6: Crossover operator for grouping problems (in analogy to [3])

4. Mark all original kernels of the old genes (fused kernels) which now occur twice in the chromosome (circles in Figure 6-b).
5. Delete all genes (fused kernels) containing at least one of the marked original kernels; note the original that have been deleted completely and are now missing (Figure 6-c).
6. Reinsert missing original kernels into new fused kernels according based on kernel pivot (Figure 6-d).

4.2.2. Mutation

Mutation operator includes small modifications in each individual of the population with a low probability, in order to explore new regions of the search space and also scape from local optima when the algorithm is near convergence. A mutation operator for grouping problems must work with groups rather than items. As for the crossover, the implementation details of the operator depend on the particular grouping problem on hand. Nevertheless, four general strategies can be outlined: *create* a new group, *eliminate* an existing group, *split* a group into two groups or, *merge* two groups together. In this case, we have implemented the two *splitting* and *merging* mutation operators and adapted them to the kernel fusion problem

- Mutation by splitting: it consists of splitting a selected fused kernels into two different ones. The original kernels belonging to the fused kernels are assigned to the new fused kernels with equal probability. Note that one of the new generated fused kernels will keep its label in the group section of the individual, whereas the other will be assigned a new label. The selection

of the initial fused kernel to be split is carried out depending on the fused kernel size, with more probability of split given to larger fused kernels.

- Mutation by merging: it consists of merging two existing fused kernels, randomly selected, into just one. As in mutation by splitting, the probability of choosing the clusters depends on their size. Note that merging two fused kernels is only possible if they have a *degree of kinship* more than zero.

Note that we apply the two mutation operators in a serial fashion (one after the other), with independent probabilities of application. In this case, probability of mutation is smaller in the first generations of the algorithm and larger in the last ones, in order to have more opportunities to scape from local minimums in the last stages of the evolution:

$$P_m(j) = P_{mi} + \frac{j}{G} * (P_{mf} - P_{mi}) \tag{2}$$

where $P_m(j)$ is the probability of mutation used in a given generation j, G stands for the total number of generations of the algorithm, and P_{mi} and P_{mf} are the final and initial values of probability considered, respectively.

4.3. Selection Operator

In this paper we use a rank-based wheel selection mechanism, similar to the one described in [9]. First, the individuals are sorted in a list based on their quality. The position of the individuals in the list is called *rank of the individual*, and denoted $R_i, i = 1, ..., \xi$ with ξ number of individuals in the population of the GGA. We consider a rank in which the best individual x is assigned $R_x = \xi$, the second best y, $R_y = \xi - 1$, and so on. A *fitness* value associated to each individual is then defined, as follows:

$$f_i = \frac{2 * R_i}{\xi * (\xi + 1)} \tag{3}$$

Note that these values are normalized between 0 and 1, depending on the position of the individual in the ranking list. It is important to note that this rank-based selection mechanism is static, in the sense that probabilities of survival (given by f_i) do not depend on the generation, but on the position of the individual in the list. As a small example, consider a population formed by 5 individuals, in which individual 1 is the best quality one ($R_1 = 5$), individual 2 the second best ($R_2 = 4$), and so on. In this case, the fitness associated to the individuals are

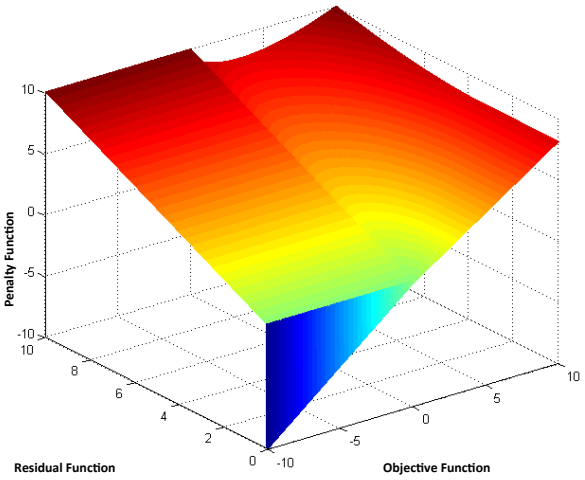

Figure 7: The oracle penalty function [16]

$\{0.33, 0.26, 0.2, 0.13, 0.06\}$, and the associated intervals for the roulette wheel are $\{0$ - 0.33, 0.34 - 0.6, 0.61 - 0.8, 0.81 - 0.93, 0.94 - $1\}$.

The process carried out in our algorithm consists of selecting the parents for crossover using this selection mechanism. This process is performed with replacement, i.e., a given individual can be selected several times as one of the parents, however, individuals in the crossover operator must be different.

4.4. Replacement and elitism

In the proposed GGA, the population at a given generation $j+1$ is obtained by replacement of the individuals in the population at generation j, through the application of the selection, crossover, and mutation operators described above. An elitist schema is also applied, the best individual in generation j is automatically passed onto the population of generation $j + 1$, ensuring that the best solution encountered so far in the evolution is always kept by the algorithm.

4.5. Constraint Handling

Penalty methods are used to handle the problem constraints. Such methods transform a constrained problem to an unconstrained problem by adding the weighted sum of constraint violations to the original fitness function. Death or static penalty methods are the most commonly used penalty methods. Although easy to use these methods gets stuck in a single feasible region and therefore, are not able to achieve good performance for tightly constrained problems. We have

used oracle penalty method [16] for constraint handling. Oracle method depends on a single parameter, named Ω, which is selected as the best equivalent or just slightly greater than the optimal (feasible) objective function value for a given problem. As for most real-world problems this value is unknown a priori, we start with a value of $\Omega = 1e^6$. We keep on improving the value of Ω by assigning the best known feasible fitness value of the previous run. Mathematically the oracle penalty function can be represented as:

$$p(x) = \begin{cases} \alpha \cdot |f(x) - \Omega| + (1 - \alpha) \cdot res(x) & \text{, if } f(x) > \Omega \\ & \text{or } res(x) > 0 \\ -|f(x) - \Omega| & \text{, if } f(x) \leq \Omega \\ & \text{and } res(x) = 0 \end{cases}$$

where α is given by:

$$\alpha = \begin{cases} \frac{|f(x)-\Omega| \cdot \frac{6\sqrt{3}-2}{6\sqrt{3}} - res(x)}{|f(x)-\Omega| - res(x)} & \text{, if } f(x) > \Omega \text{ and} \\ & res(x) < \frac{|f(x)-\Omega|}{3} \\ 1 - \frac{1}{2\sqrt{\frac{|f(x)-\Omega|}{res(x)}}} & \text{, if } f(x) > \Omega \text{ and} \\ & \frac{|f(x)-\Omega|}{3} \leq res(x) \leq |f(x) - \Omega| \\ \frac{1}{2}\sqrt{\frac{|f(x)-\Omega|}{res(x)}} & \text{, if } f(x) > \Omega \text{ and} \\ & res(x) > |f(x) - \Omega| \\ 0 & \text{, if } f(x) \leq \Omega \end{cases}$$

Shape of the oracle penalty function is shown in Figure 7. The oracle penalty function is designed to deal with the non-convexities. The following pruning scheme was also introduced to speedup the search process. The kernel fusion problem has many constraints. However, most of the constraints are not active, e.g. if *shared arrays* are not a *kernel pivot* of a fused kernel, the number of constraints can be decreased. Obviously, checking active constraints will be sufficient for the feasibility check of the current mapping and additionally, if one constraint is violated, the current fusion can be skipped and the algorithm can proceed.

Algorithm 1 Algorithm for Avoiding Redundancy.

1: Initialize counter
2: loop until the maximum number of restarts is achieved
3: Run GGA
4: **if** no better solution is found in the current restart
5: Increment the counter
6: **if** counter exceeds the max. number of similar runs (M_r)
7: Evaluate the neighborhood in sub-solution space to make sure that the point is the local optima
8: **if** a better solution is found in sub-solution space
9: Reset counter and continue the loop
10: **else**
11: Insert the solution into a finite size Queue
12: **end If**
13: end loop

4.6. Avoiding redundancy

Avoiding redundant search near already visited local optima is a key to better and faster search. We have used a simple technique inspired from Tabu search [11] algorithm to avoid this redundancy. The algorithm used for avoiding redundancy is shown in Algorithm 1. Whenever a promising individual is located, the algorithm intensifies the search around that individual in order to find an even better solution. When the algorithm reaches the best solution in this small sub-solution space, it is assumed with high probability that the individual found is the best individual in that sub-solution space. The algorithm maintains a finite list of such visited local optima. The neighborhoods of these individuals are penalized in every fitness evaluation. As the list is of finite size the individual ultimately gets removed from the list and gets a second chance of being searched by the algorithm.

4.7. Population Initialization

For the initial population, chromosomes are generated randomly. In our case, only chromosomes corresponding to feasible solutions have been generated, since application of a GGA that starts from a population with too many infeasible chromosomes will not result in high quality solutions. Moreover, determining the suitable population size is an important decision. If the selected number is too

Algorithm 2 Improving Performance by Kernel Fusion

1: Gather metadata of original kernels K_i, $i = 1, , n$

2: Create dependency graph and order-of-execution graph

3: $G_0 \leftarrow$ Generate M individuals (initial population) each having set F of fused kernels not violating order-of-execution

4: For all M, for fused kernel $f \in F$:

  i: Aggregate metadata of original kernels of f

  ii: Project the runtime of f from projected performance bound

5: $G_t^{Se} \leftarrow$ select $N \leq M$ individuals from G_{t-1} according to selection method

6: $G_t^{Se} \leftarrow$ Apply crossover and mutation operators by defined rates

7: $G_t \leftarrow$ Replace N individuals with G_t^{Se} using a replacement policy

8: If termination criteria not met, go to step **4**

9: Use values of best solution as guide for fusing kernels

small, algorithm may not be able to obtain a good solution. On the other hand, if the number is too large, it may use too much CPU time to obtain a better solution. A procedure was developed in this research to generate a random initial population while constraints are satisfied, i.e. only original kernels with a *degree of kinship* more than zero are fused together.

5. Using Kernel Fusion to Speedup Applications

The details of how the output of optimization algorithm is used to transform HPC applications and achieve speedup is beyond the scope of this paper. The process is briefly overviewed in Algorithm 2. The optimization algorithm is shown in steps 3 to 8. Note that components of the system can be changed. For example, the performance projection method (step 4.ii) used as an objective function can be any performance bound model (e.g. Williams:2009:RIV:1498765.1498785). A highly accurate and lightweight performance projection model was derived and used in this report. Note that the last step in the algorithm can either be manual or automated by a source-to-source code transformation.

6. Results and Evaluation

The algorithm is controlled and configured by various input parameters. All the parameters are preconfigured to an appropriate value. However, an advanced user can modify the parameters through the parameter's configuration file. Nomenclature of the parameters is given in Table 2. With so many parameters for the

Table 2: Parameters to tune the optimization algorithm

Parameter	Meaning	Value Used
G, P, R	Maximum number of allowed generations, population and restarts	$G = 10$, $P = 10$, $R = 30000$
P_{con}	Penalty function and penalty configuration	0.7
P_c, P_{mi}, P_{mf}	Crossover probability, initial mutation probability and final mutation probability	$P_c = 0.15$, $P_{mi} = 0.1$, $P_{mf} = 0.3$
T	Tournament size for the selection operator	3
Pr	Proximity parameter for sampling of population in consecutive runs	0.1
η_c, η_m	Crossover & mutation probability distribution index	$\eta_c = 2, \eta_m = 100$
Ω	Initial value of the oracle for oracle penalty method	1e6
I_a	Number of restarts for performing grouped genetic algorithm	8
I_r	Number of iterations of each round of grouped genetic algorithm	8
St	Stopping criteria	"No improvement"
M_f, M_t	Maximum allowed fitness evaluations & execution time (hours)	$M_f = 1.0e8$,$M_t = 4$
M_G, M_R	Maximum allowed generations & runs without any improvement in fitness	$M_G = 3000$,$M_R = 8$
Q	Size of the queue for the Tabu list	30
N	Size of the neighborhood that needs to be penalized	4
E	Allowed error between the calculated and the best known result	1%
E_p	Residual accuracy	0.01

algorithm calibration, it is vital to carefully study the effect of each and every important parameter on the total execution time and solution quality. The preset values of the parameters used for the experimentation is shown in the table. The values of these parameters are selected empirically by studying the effect of the different parameters on the output.

6.1. Systems and Specifications

Results for the optimization algorithm given in this section were collected over a system with Intel®Core™Xeon X5670 2.93GHz CPU. GCC v.4.4.7 was used as a compiler configured to highest possible compiler optimization. The search heuristic was run 10 times for each application to confirm the same best solution is repeatedly found. The best solution was then used as the basis for manually fusing kernels.

Tsubame2.5 supercomputer was used for GPU kernel fusion experimentation. As for the GPU software, PGI 14.3 CUDA Fortran and CUDA 5.5 were used for the Nvidia Kepler K20x and K40 GPUs. The test suite kernels are written in CUDA C and the real-world applications are written in CUDA Fortran. GPU Performance results are averaged for 10 runs compiled at the highest possible compiler optimization.

Table 3: Attributes of the test suite built from CloverLeaf

Attribute	***Min***	***Max***	Δ	**Attribute**	***Min***	***Max***	Δ
# Kernels	10	100	10	*Size Sharing set*	2	8	2
# Arrays	20	200	20	*Avg. Thread Load*	4	12	4
# Data Copies	2	10	2	*Kinship*	2	5	1

6.2. Benchmarking and Applications

6.2.1. Test Suite From Mini-App: CloverLeaf

CloverLeaf [7] is a Lagrangian-Eulerian hydrodynamics mini-app that solves the compressible Euler equations on a 2D Cartesian grid. The computation in CloverLeaf is broken down into several kernels. A test suite based on the kernels from CloverLeaf was constructed to help understand the factors affecting the kernel fusion optimization algorithm through a controlled study of benchmarks with different features. Table 3 shows the test suite evaluated in this paper. The test suite includes different benchmarks for all attribute values from *Min.* to *Max.* with a step of Δ. Note that most important attributes are the number of kernels and number of data arrays; an increasing number of kernels means a larger search space and an increasing number of data arrays means increasing complexity in the search space due to the increasing number of *sharing sets*.

6.2.2. Real-world Applications: SCALE-LES and CAM-HOMME

The ability of the kernel fusion optimization algorithm to generate effective fusions was evaluated using two weather models: SCALE-LES [19] and CAM-HOMME [2]. SCALE-LES is a next generation weather model requiring a huge amount of computational resources beyond the current existing systems. SCALE-LES is calculated to have 41% reducible GMEM traffic; the bound of reduction in SCALE-LES's runtime using the best solution found by the optimization algorithm is 41%. For reference, [19] includes full details about the GPU version of SCALE-LES. SCALE-LES includes over a hundred kernels. The majority of those kernels have a low FLOP to byte ratio as they have a stencil-like form.

HOMME is the dynamical core within the Community Atmospheric Model (CAM). It is calculated to have 21% reducible GMEM traffic. HOMME has a much smaller search space compared to SCALE-LES (43 kernels compared to 142 kernels in SCALE-LES). However, HOMME a search space that is highly complex, i.e. more non-convexities compared to SCALE-LES.

6.3. Algorithm's Efficiency

6.3.1. Test Suite

The table below explains the abbreviations used for reporting the results in the following tables.

Abbreviation	Explanation
Problem (#K,#A)	Benchmark with number of kernels K and number of data arrays A
$\text{Restarts}_{\text{mean}}$	average number of restarts
$\text{Eval}_{\text{mean}}$	average number of evaluations over all runs with a feasible solution
$Feasible_{\%}$	Percentage of feasible solutions found out among the benchmarks of 10 test runs
$Optimal_{\%}$	Percentage of optimal solutions found out among the benchmarks of 10 test runs
f_{best}	best (feasible) objective function value found out of 30 test runs
f_{worst}	worst (feasible) objective function value found out of 30 test runs
f_{mean}	average objective function value over all runs with a feasible solution
$\text{Time}_{\text{mean}}$	average CPU-time over all runs with a feasible solution

Table 4 shows the performance results obtained by applying the proposed algorithm on different benchmarks from test suite. Note that the time required in the algorithm, in the magnitude of hours, is considered reasonable considering that kernel fusion is done offline and that real-world applications benefiting from this method typically require months and years of implementation. Table 5 shows the quality of solutions found for benchmarks with variation in the *thread load* and *sharing set* cardinality according to Table 3. The optimal solutions for benchmarks of small sizes were verified using a deterministic method. Note the high percentage of feasible solutions signifying the effectiveness of the oracle penalty method for the type of non-convexities in the kernel fusion search space.

6.3.2. Applications

Table 6 shows performance metrics and parameters of the search algorithm for: a) SCALE-LES (142 original kernels and 65 *sharing sets*, estimated to have $\sim$2.6e45 feasible solutions) and b) HOMME dynamical core (43 original kernels and 29 *sharing sets*). Only three out of the main fused kernels in SCALE-LES and two in HOMME did not utilize the SMEM at maximum capacity. Note the relatively long runtime compared to the total number of evaluations of the objective function. Using a performance model that requires code representation would increase the runtime significantly. For example, using the exhaustive search method in GROPHECY[13] to evaluate the feasible solutions to fuse SCALE-LES original kernels would have required more than $2.1e39$ hours as a single evaluation using the MWP performance model adopted in GROPHECY is measured at 3ms.

Table 4: Performance results obtained by applying the proposed algorithm on different benchmarks

Problem (#k, #A)	Restarts_{mean}	Eval_{mean}	f_{best}	f_{worst}	f_{mean}	Time_{mean} (H)
(10,20)	53	9586	2.8772	2.8772	2.8772	0.13 ± 0.25m
(20,40)	43	14381	3.4132	3.4132	3.4132	0.31 ± 0.25m
(30,60)	37	23253	4.0247	4.0247	4.0247	0.52 ± 0.09m
(40,80)	104	26092	4.9002	4.9117	4.9042	0.80 ± 1.00m
(50,100)	58	41626	5.3328	5.3572	5.3472	0.92 ± 0.70m
(60,120)	237	73581	5.8436	6.0105	5.9215	1.01 ± 0.73m
(70,140)	141	92646	8.8865	8.7162	8.5583	1.02 ± 1.42m
(80,160)	26821	26.04e6	11.0821	12.5829	11.3666	1.12 ± 1.40m
(90,180)	27871	36.25e6	11.0375	18.3966	15.2302	1.23 ± 1.64m
(100,200)	24092	43.91e6	13.6967	23.9085	17.1293	1.61 ± 2.44m

6.4. Speedup Achieved by Applying Best Solutions for Kernel Fusion

For the standard problem size of SCALE-LES set to 1280x32x32 and HOMME set to (4, 26, 101), the GPU version after kernel fusion for SCALE-LES and HOMME shows over 1.35x and 1.2x speedup for a single node, respectively. SCALE-LES and HOMME are reported to have an almost linear weak scaling [19, 2] over a spectrum of different systems and hence the decrease in runtime for a single node would yield almost the same decrease in runtime when using multi nodes assuming overlapped computation and communication.

The main goal of this work to speedup GPU applications via kernel fusion to reduce GMEM traffic. However, there is a gap between the calculated reducible traffic and the actual reduced traffic after fusion, e.g. SCALE-LES had a 41% reducible traffic, 26% reduction in runtime was only achieved leaving 15% of wasted opportunity for improvement. It is essential to identify the reason behind the missed portion, i.e. is it he inability of the search algorithm to find the best solutions that can yield the calculated reducible traffic or is due to the constraints of resources, i.e. SMEM capacity and register used per thread, that limit the search algorithm from finding those best solutions since they will be unfeasible.

Therefore, we used the projection model to study potential performance improvements with hypothetical architecture configurations. One interesting scenario is the effect by the SMEM capacity on the performance improvement combined with the data reuse with kernel fusion. By running the model with 128KB and 256KB for SCALE-LES, compared to the actual 64KB SMEM capacity, we found that the best solution would yield 1.56x and 1.65x improvements, which

Table 5: Quality of results obtained by applying the proposed algorithm on different benchmarks

Problem (#k, #A)	$Eval_{mean}$	$Feasible_{\%}$	$Optimal_{\%}$	f_{best}	f_{worst}	f_{mean}
(20,20)	14241	100	100	3.4624	3.4624	3.4624
(20,40)	14381	100	100	3.4132	3.4132	3.4132
(20,60)	14701	100	100	3.4069	3.4069	3.4069
(30,20)	22790	100	100	4.674	4.76	4.905
(30,40)	22821	99.4	98.5	4.472	4.558	4.703
(30,60)	23253	100	99.4	4.031	4.117	4.262
(40,20)	25381	99.8	99.7	5.7038	5.6955	5.688
(40,40)	25412	99.3	98.7	5.5088	5.5005	5.493
(40,60)	25437	99.1	97.3	4.9418	4.9335	4.9260
(50,20)	40262	99.3	98.2	5.5122	5.5366	5.5286
(50,40)	40959	99.3	98.1	5.4781	5.5025	5.4945
(50,60)	41216	99.5	99.1	5.4069	5.4313	5.4233

is within 2% of the calculated GMEM reducible traffic. Although the increased capacity would also imply architectural trade-off, the speculative study with our modeling highlights the fidelity of the optimization algorithm.

7. Related Work

Attempts for improving GPU performance by fusing kernels to exploit locality is introduced in different contexts. However, to the authors knowledge, no kernel fusion methods were applied at the scale of large applications, i.e. dozens of kernels and data arrays. Therefore, extending the existing approaches would be challenging due to those approaches using basic methods to search the space of possible fusions (e.g. exhaustive search). This renders the kernel fusion optimization prohibitive in large problem sizes (i.e. large number of kernels and data arrays). The following is a review of notable GPU kernel fusion related work.

Kernel fusion/fission is proposed in [22] to reduce the impact of the limited PCIe bandwidth for codes utilizing relational algebra operators. The work in [22] emphasized the effect of the number of operations on the fusion process with less regard to fusing multiple kernels. GROPHECY [13] is a GPU performance projection tool that uses CPU code skeletons. The tool is further used in [14] to fuse kernel. GROPHECY uses a high level abstraction to model the CPU codes

Table 6: Performance & parameters of the search algorithm

Application	*Number Generations*	*Population Size*	*Total # Evaluations*	*Average Runtime*
SCALE-LES	2000	100	5.4e6	1.23 H
HOMME	1000	100	2.7e6	0.79 H

and an unscalable brute-force search for code layouts. It showed to be effective for the reported fusion of three or less kernels.

Fousek el al. [6] used a customized branch and bound method to search the space of possible fusions. The kernels were represented as elementary mapping functions that share the same template-like design separating the load, store and computation routines, thus simplifying the performance predication to benchmarking routines which in turn simplifies the branching procedure. The approach is reported to be effective, yet not appropriate for modeling applications having kernels of different sizes and data access patterns. The work introduced in [5] uses a branch and bound Map-Reduce approach to fuse kernels composed of linear algebra operations. Representing BLAS functions as Map and Reduce calls simplified the search process at the expense of flexibility, i.e. having to create kernels from a fixed set of problems-specific elementary functions.

Fusing independent kernels was also proposed in [21] to optimize for power. The motivation of fusing kernel was to achieve higher utilization and a much more balanced demand for hardware resources. The problem was represented as a dynamic programming problem of fusing a small number of kernels based on their power-related features. Reported fusion of two kernels was shown to be rewarding for power optimization.

8. Conclusion & Future Work

Kernel fusion, aggregating GPU kernels sharing data arrays, can speedup applications by exposing data locality and allowing for data reuse. The real challenge with achieving an actual speedup is identifying the best fusions. The next challenge is the actual implementation of the fusion. This report is concerned with the first challenge, i.e. designing and implementing an optimization algorithm to solve the kernel fusion problem which is formulated as a variation of the bin packing problem.

Many of the real-world applications that would benefit from kernel fusion, e.g. weather models, include dozens of kernels and data arrays. Hence, this report proposed a scalable method for kernel fusion to improve the performance of this

class of applications. A grouped genetic algorithm that well fits the nature of dependencies in kernel fusion was used to represent the problem at the fused kernel granularity. Adapted genetic operator, constraint handling method and population initialization methods were introduced.

The proposed algorithm was evaluated by a test suite and two next-generation weather models: SCALE-LES and CAM-HOMME. The proposed method was proven scalable to large sized benchmarks in the test suite, and applicable to SCALE-LES and HOMME yielding 1.35x and 1.2x speedup, respectively. As a future work, calibrating (optimizing) the input parameters via auto-tuning is of interest.

References

[1] M. Bianco, T. Diamanti, O. Fuhrer, T. Gysi, X. Lapillonne, C. Osuna, T. Schulthess, A GPU Capable Version of the COSMO Weather Model, ISC '13, 2013.

[2] I. Carpenter, R. Archibald, K. Evans, J. Larkin, P. Micikevicius, M. Norman, J. Rosinski, J. Schwarzmeier, M. Taylor, Progress Towards Accelerating HOMME on Hybrid Multi-core Systems, Int. J. High Perform. Comput. Appl. 27 (2013) 335–347.

[3] E. Falkenauer, A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems, Evolutionary Computation 2 (1994) 123–144.

[4] E. Falkenauer, A Hybrid Grouping Genetic Algorithm for Bin Packing, Journal of Heuristics 2 (1996) 5–30.

[5] J. Filipovic, M. Madzin, J. Fousek, L. Matyska, Optimizing CUDA Code By Kernel Fusion—Application on BLAS, CoRR 1305.13.

[6] J. Fousek, J. Filipovič, M. Madzin, Automatic Fusions of CUDA-GPU Kernels for Parallel Map, SIGARCH Comput. Archit. News 39 (4) (2011) 98–99.

[7] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, S. A. Jarvis, Accelerating Hydrocodes with OpenACC, OpeCL and CUDA, SCC '12, 2012.

[8] C. Hill, C. DeLuca, V. Balaji, M. Suarez, A. d. Silva, The Architecture of the Earth System Modeling Framework, Computing in Science and Eng. 6 (2004) 18–28.

[9] T. L. James, M. Vroblefski, Q. Nottingham, A Hybrid Grouping Genetic Algorithm for the Registration Area Planning Problem, Computer Communications 30 (2007) 2180–2190.

[10] R. E. Korf, A New Algorithm for Optimal Bin Packing, in: Eighteenth national conference on Artificial intelligence, 2002.

[11] S. Kulturel-Konak, B. A. Norman, D. W. Coit, A. E. Smith, Exploiting Tabu Search Memory in Constrained Problems, INFORMS J. on Computing 16 (2004) 241–254.

[12] E. Lopez-Camacho, H. Terashima-Marin, G. Ochoa, S. E. Conant-Pablos, Understanding the Structure of Bin Packing Problems Through Principal Component Analysis, International Journal of Production Economics 145 (2013) 488 – 499.

[13] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, T. D. Uram, GROPHECY: GPU Performance Projection from CPU Code Skeletons, SC '11, 2011.

[14] J. Meng, V. A. Morozov, V. Vishwanath, K. Kumaran, Dataflow-driven GPU Performance Projection for Multi-kernel Transformations, SC '12, 2012.

[15] K. Praditwong, Solving Software Module Clustering Problem by Evolutionary Algorithms, in: Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on, 2011.

[16] M. Schlueter, M. Gerdts, The Oracle Penalty Method, Journal of Global Optimization 47 (2) (2010) 293–325.

[17] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, S. Matsuoka, An 80-Fold speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code, SC '10, 2010.

[18] W. C. Skamarock, J. B. Klemp, A Time-split Nonhydrostatic Atmospheric Model for Weather Research and Forecasting Applications, J. Comput. Phys. 227 (2008) 3465–3485.

[19] M. Wahib, N. Maruyama, Highly Optimized Full GPU-Acceleration of Non-hydrostatic Weather Model SCALE-LES, CLUSTER'13, 2013.

[20] M. Wahib, N. Maruyama, Scalable Kernel Fusion for Memory-Bound GPU Applications, in: Submitted, SC '14, 2014.

[21] G. Wang, Y. Lin, W. Yi, Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU, in: GreenCom'10, 2010.

[22] H. Wu, G. F. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, S. T. Chakradhar, Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission, in: IPDPS Workshops, 2012.