Scalable Kernel Fusion for Memory-Bound GPU Applications

Mohamed Wahib
RIKEN Advanced Institute for Computational Science
JST, CREST
Kobe, Japan 650–0047
Email: mohamed.attia@riken.jp

Naoya Maruyama
RIKEN Advanced Institute for Computational Science
JST, CREST
Kobe, Japan 650–0047
Email: nmaruyama@riken.jp

***Abstract*—GPU implementations of HPC applications relying on finite difference methods can include tens of kernels that are memory-bound. Kernel fusion can improve performance by reducing data traffic to off-chip memory; kernels that share data arrays are fused to larger kernels where on-chip cache is used to hold the data reused by instructions originating from different kernels. The main challenges are a) searching for the optimal kernel fusions while constrained by data dependencies and kernels' precedences and b) effectively applying kernel fusion to achieve speedup. This paper introduces a problem definition and proposes a scalable method for searching the space of possible kernel fusions to identify optimal kernel fusions for large problems. The paper also proposes a codeless performance upper-bound projection model to achieve effective fusions. Results show that using the proposed scalable method for kernel fusion improved the performance of two real-world applications containing tens of kernels by 1.35x and 1.2x.**

I. Introduction

GPU-accelerated HPC applications, such as weather models, can have large codebases consisting of dozens of functions offloaded to GPUs. The use of finite difference or finite volume methods with iterative stencils renders most of these functions, called *kernels*, as memory-bound. The analysis of the source code of various weather applications in different states of being ported to GPU and falling into this class of problems showed an opportunity to reduce data traffic to off-chip memory by exploiting data locality across kernel calls [1–6]. 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 (e.g. shared memory, or SMEM, of Nvidia GPUs) to hold the data in-between instructions coming from different kernels. With the SMEM bandwidth an order of magnitude higher than that of device global memory (GMEM) in Nvidia GPUs, performance can potentially improve as the total FLOP-to-byte ratio of the new kernels shifts to the right endpoint of the Roofline model [7].

Applying kernel fusion to real-world large-scale applications on GPUs, however, is difficult. The first main challenge is defining a scalable method to search for the optimal rearrangement and fusion of kernels. This study is intended to be applicable to stencil-based scientific applications having large codebases with tens of kernels and data arrays. Large codebases, as discussed in Section II-B, are the norm, especially in climate and weather modeling applications, one of the most important computational science domains. Thus, exhaustive search used with existing kernel fusion methods (Ex. [8, 9]) would be prohibitive for such applications. Moreover, the search process must take into account the complex data dependencies for arrays different in size and access patterns to maintain the precedence constraints in permutations of possible kernel fusions. The second main challenge is to accurately account for GPU-specific architectural features and constraints. Architectural constraints, such as the number of registers, capacity of shared memory, and number of shared memory banks can have significant performance implications in resulting new kernels; thereby, severely limiting the effectiveness of greedy, non-architecture-aware algorithms.

To address the two challenges, we formulate kernel fusion as a combinatorial optimization problem and propose an approximation heuristic for searching the exponentially sized space of possible fusions. The optimization problem includes constraints that respect data dependencies and execution order requirements of the input program. Our search heuristic uses a light-weight and tight performance upper-bound to evaluate the quality of candidate solutions. The upper-bound on performance is projected for potential fusions without requiring any form of code representation. This light-weight method is essential to enable fusions for applications with a large number of kernels and data arrays. The result of the search heuristic can be used to guide the generation of new kernels manually, as discussed in this paper, or be further leveraged by an automated source-to-source code transformation.

The main contributions in this paper are a) formulation of kernel fusion as an optimization problem, b) use of a scalable search heuristic to search for near-to-optimal solutions in the space of possible kernel fusions, c) use of a highly accurate codeless upper-bound performance projection model to guide kernel fusion and, d) experimental evidence on the effectiveness of the kernel fusion optimization when applied to a test suite and two real world weather applications with tens of kernels and data arrays. As discussed later, the proposed search heuristic identified 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.

SC14, November 16-21 2014, New Orleans
978-1-4799-5500-8/14/$31.00 ©2014 IEEE

II. Background

This section first discusses the relevance of loop fusion, followed by the rationale for GPU kernel fusion, basic terminology, and a motivating example.

A. Loop Fusion

Loop fusion is a classical compiler optimization that can improve performance by reducing off-chip memory traffic [10, 11] (details are omitted due to space limitation. For reference, [10] covers loop fusion extensively). Loop fusion commonly operates by fusing iterations of different loops when those iterations reference the same data. The fusion should take into account loops of different shapes, loops of a different number of dimensions, dependencies from code in-between loops, and non-perfectly nested loops. Next, and due to non-explicit control over the cache, a secondary step of data regrouping is required to populate the cache exclusively with data used by the computation.

Fusion can also be applied to a group of GPU kernels to increase effective bandwidth by data reuse via cache. In fact, it is likely that a large number of kernels can be legitimately fused because GPU kernels are typically modular in the sense of all statements and data being confined in the kernel and having no relation to any statements or local data appearing outside the scope of the kernel. However, similar to loop fusion, kernel fusion does not always result in performance improvements. An effective fusion must take into account limited architectural resources such as the capacity of on-chip memory and the number of available registers, while maximizing the reduction of off-chip memory data traffic. Moreover, it is necessary to have a mechanism of finding fusible kernels that can be used at scale with respect to the number of kernels. Existing proposals on loop fusions, while they are conceptually similar to our method, often operate only with a relatively small number of loops (or individual iterations in polyhedral frameworks [11]) with a greedy incremental method [10], which would be prohibitive when applied to large GPU codebases.

B. Rationale for Kernel Fusion: Data Reuse

Data in this paper refers to data arrays used in finite difference methods, e.g., a 3D grid representing the atmospheric space. Kernels, running on GPU, are typically hot spots and would most likely enclose loop(s) (more specifically the inner loop(s) in the original CPU code). The main routines in large-scale applications, e.g., the dynamical core of weather models, can contain tens of memory-bound kernels that represent bottlenecks. *Kernel Fusion* is a code transformation in which new kernels are constructed by aggregating codes of smaller kernels. The target of this study was to fuse the kernels such that data reuse using on-chip memory improves performance.

To improve performance by reducing the off-chip memory operations, the following main steps are essential a) construct a data dependency graph and order-of-execution graph of the kernels in the program, b) rearrange kernel calls from the host side to expose locality by placing kernel calls using the same data array(s) in sequence, c) for every sequence of kernels using the same data, construct a new kernel that aggregates the code segments of the kernels and replace the sequence of calls from the host by a single call to the newly constructed kernel, and d) in the newly constructed kernels, use the on-chip memory to hold the data shared between instructions coming from different code segments.

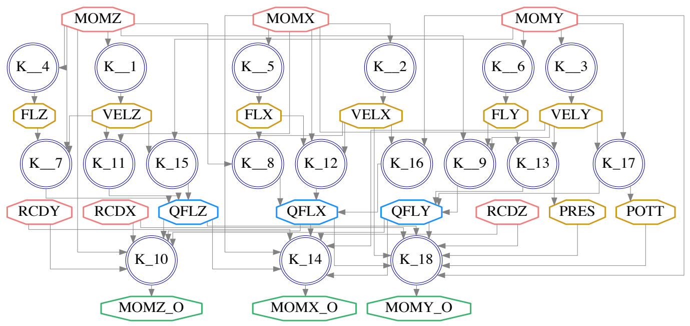


Fig. 1: Simplified data dependency graph of a routine in SCALE-LES. Circles are kernels invoked in order of numbered label. Diamonds are arrays used with following intention: read-only arrays in red, read-write arrays in yellow, expandable read-write arrays in blue, and write-only arrays in green

1) Data Dependency Graph: The potential for optimization by data reuse in GPU kernels is demonstrated by a motivating example extracted from an important routine in the SCALE-LES weather model [1]. The 3^{rd} order Runge-Kutta (RK) routine contains tens of kernels and is a hotspot in SCALE-LES. Figure 1 shows the simplified data dependency graph of RK. The dependency graph is a DAG composed of vertices that can be either kernels or data arrays. The edges connect the kernels with the data arrays depending on the how the data arrays are touched, e.g., an edge from a data array to a kernel means the data array is used as read-only in the kernel. Hence, the graph reveals the inter-dependencies between kernels.

There are four ways for the data arrays in a program to be touched by kernels in the lifetime of a program:
a) Read-only arrays are the most common form. Reuse for this type of arrays is easily exposed by fusing kernels sharing the same read-only arrays. Reusing those arrays does not add implications and is mostly bound by the capacity of the SMEM.
b) Read-write arrays can be reused if both the kernels that write into the array and the dependent kernel that later reads the array are fused together. The problem however would be the need for a barrier between the operations that store and those that load the data array into the SMEM. Additionally, incoherency will occur after the barrier; the threads in a thread block have no access to the SMEM allocated to other thread blocks and the SMEM is incoherent with the GMEM.
c) Expandable read-write arrays are read-write arrays with the exception of having several kernels writing into them. For example in Figure 1, kernel *K_8* writes into array *QFLX*, which kernel *K_10* subsequently reads. Later in the program, kernel *K_12* writes into the same array, *QFLX*, which kernel *K_14* subsequently reads. Array *QFLX* in this case imposes a precedence constraint; instructions from *K_8* must be executed before those of *K_10*. Changing the kernels to write into redundant arrays and adjusting accordingly would relax the dependencies by removing the precedence constraint. Note that this comes at the expense of extra memory capacity.
d) Write-only arrays that cannot be reused.

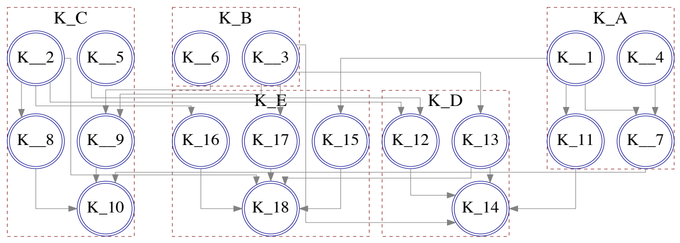


Fig. 2: Order-of-execution graph built from dependency graph in Figure 1. Proposed fusion is shown by enclosing original kernels in dotted lines

2) Order-of-execution Graph: To find possible fusions of the kernels in a program, an *order-of-execution graph* is required to assure the proposed fusions do not violate the logic flow by breaking a precedence constraint. Note that the order-of-execution graph is not a call graph, i.e., the host invokes all the kernels. An order-of-execution graph is a DAG with vertices that represent the kernels and edges that correspond to the inter-kernel precedence that should not be violated. For example, in Figure 2, for a fusion to be valid, the instructions in kernel *K_2* must be invoked before the instructions of kernel *K_8*. The order-of-execution graph is constructed using the data dependency graph in Figure 1. The precedences extracted from data dependencies are relaxed using redundant arrays for expandable read-write arrays. For the next step, the order-of-execution graph is used to identify which original kernels to fuse together into new kernels such that the logic flow is not violated. One of the possible fusions for the kernels in Figure 2 is shown as dotted rectangles. If the fusion shown in the figure is applied, the 18 original kernels would be replaced by 5 new kernels invoked in this order; *K_A*, *K_B*, *K_C*, *K_D* and *K_E*.

3) Prospect of Kernel Fusion in Real-world Applications: Table I lists the relevant features for different weather applications at different stages of being ported to GPU. The features were collected through static analysis using the ROSE compiler [12] in addition to manual analysis in some cases. For the applications that have not been fully ported to a GPU yet, the number of anticipated kernels was conservatively estimated by assuming every main Fortran subroutine (function in C) to be mapped to a GPU kernel. Assuming the fusion of kernels that use data arrays, which have more than one thread in a thread block accessing the same data element, the maximum fusion that does not invalidate the order-of-execution represents the upper-bound for data traffic reduction. For example, SCALE-LES [1] is calculated to have almost 41% reducible GMEM traffic.

C. Assumptions and Restrictions

The techniques discussed in this paper are based on the following assumptions about the targeted applications. Stencil kernels typically have low arithmetic intensity; hence, new kernels generated from fusion are expected to be memory-bound despite improved data locality. All kernels, original and new, have the same number of threads per block and blocks per grid at invocation time. Data accesses are coalesced (a common feature due to uniform memory access in stencil codes). The proposed method takes into account existing data transfer between the host and device in-between kernel

TABLE I: Features of Different Weather Applications

Application	No. of Kernels	No. of Arrays	Reducible Global Memory Traffic
SCALE-LES [1]	142	64	41%
WRF [13]	122	46	24%
ASUCA [3]	115	58	17%
MITgcm [14]	94	31	22%
HOMME [6]	43	27	21%
COSMO [4]	35	24	38%

invocations as a constraint when constructing the order-of-execution graph. Existing CUDA streams are also handled in a similar manner. Finally, it is assumed with the proposed method that each original kernel has a single call site. Hence, each original kernel appears once in new kernels. To address the scenario of several invocations of a single kernel in the original code, the proposed method can be extended to treat different invocations to the same original kernel as if they are invocations of different kernels, i.e., the same approach as ”expandable arrays” but for kernels and not arrays.

The following are assumptions related to the GPU architecture features. Data arrays considered for traffic reduction are in the GPU memory space. Note that regardless of the future physical/logical proximity between the host and GPU memories, the proposed method would still address the same fundamental problem, i.e., how to fuse independent functions (GPU kernels) to expose data locality and reduce traffic to the lower-level memory. We assume that in the case of different loop bounds in the original CPU codebase, the GMEM data index is offset by the difference in value in the loop indices. As a consequence, when fusing kernels, the indices for the arrays in GMEM are adjusted between fused portions to align the threads to the bounds of the loops to generate non-penalizing warp divergence. The penalty of thread divergence is avoided by padding the arrays in the horizontal spatial direction. The following assumption is related to on-chip memory. Starting from the Kepler architecture, an on-chip *read_only cache* of 48KB per thread block is addressable but limited only to read only data as the name implies. Hence, for applications with data intended as read-only throughout the program, the read-only cache could be a viable asset for relaxing the on-chip memory capacity limit.

Varying the data arrays sizes at runtime has no impact on the effectiveness of kernel fusion, assuming each thread block's workload remains the same. This is because kernel fusion exploits locality across kernel calls by the threads in the same thread block. For the workload per thread block, the GPU single instruction, multiple threads (SIMT) execution model assigns all the threads to execute the same kernel. Therefore, the amount of work per thread, and consequently thread block, can be kept constant despite not knowing the data arrays sizes. Keeping the amount of work per thread constant is done by changing the CUDA grid and block sizes used across all kernels, original and new, to maintain the setting of each CUDA thread loading a single stencil site to the SMEM.

D. How Kernels are Fused: Motivating Example

The method on how to decide which kernels to fuse using a scalable approach is discussed in Section III. This section demonstrates how kernels are actually fused after a decision on which kernels to fuse together is reached. Table II includes a list of terms that will be frequently used. References to Figure 3 are included as examples.

TABLE II: Summary of Terminology Related to Kernel Fusion Used in this Paper

Term	Meaning	Example in Figure 3
Original kernel	Kernel in original GPU implementation before applying fusion	Kernel A
New kernel	Kernel generated from fusing two or more original kernels	Kernel X
Original sum: F^{Σ}	Summation of measured runtimes of original kernels that were fused into F	X^{Σ}=Time(Kernel A) + Time(Kernel B)
Shared array: $\langle D\rangle$	Array D touched by at least two kernels	Array Q touched by kernels D and E
Sharing set: $\mathbb{K}(D)$	Set of kernels sharing use of array D	$\mathbb{K}(Q) = \{Kern\_D, Kern\_E\}$
Thread load: $D \xrightarrow{T} K$	Average number of threads in thread block of kernel K that access same data element in array D	$Q \xrightarrow{T} C = 3$
Kernel pivot: F^{Pivot}	Set of data array(s) that now have data reuse via SMEM in new kernel F	$Y^{Pivot} = \{T, Q, V\}$
Degree of kinship: $(K_1, K_n)^{\circ}$	= 1 if two kernels k_1 and k_n directly share at least one data array = $n-1$ if there is at least one data array shared between each two consecutive kernels in the chain of kernels $k_1, k_2, \ldots, k_n$ = 0 otherwise	$(Kern\_C, Kern\_D)^{\circ} = 2$

The fusion of kernels K_1, K_2, …, K_n is done by aggregating the codes of the kernels to make a new kernel K_{New}. Figure 3 illustrates a synthetic motivating example showing data reuse via fusing GPU CUDA kernels operating on 3D data arrays. The example introduces the two possible types of fusion: simple and complex. The example also provides an insight into a limitation of using a simple performance model, as discussed later in Section IV. The two possible types of fusions are as follows:

1) Simple Fusion: Simple fusion may occur when $K_1, K_2, \ldots, K_n \in \mathbb{K}(D)$, but no order-of-execution constraints between the kernels exist. In such a case, the codes of the kernels are aggregated in a single new kernel. If $D \xrightarrow{T} K_1$ is more than one, then K_1 might already have D loaded to the SMEM for optimization in the original GPU implementation. In this case, the codes of $K_2, \ldots, K_n$ are changed to load D from the SMEM. If $D \xrightarrow{T} K_1$ is equal to one, i.e., a single data value is shared between $K_1, K_2, \ldots, K_n$, then a register is used to hold the value. The process is done repeatedly for every array D, such that $K_1, K_2, \ldots, K_n \in \mathbb{K}$ for $\mathbb{K}(D)$. Kernel Y in Figure 3 Listing 7 is an example of this type of fusion.

2) Complex Fusion: Complex fusion may occur when the kernels have order-of-execution constraints. This fusion is like a *simple fusion*, yet barriers are inserted in between the codes of kernels. In addition to the barrier, this type of fusion introduces a type of problem related to the architecture of GPUs. The SMEM in GPU is not coherent with L2 and GMEM. For K_1 writing to array(s) D, after the barrier, K_2 would use D that is stored in the SMEM. However, incoherency will cause boundary threads in all thread blocks that load data directly from the GMEM to be unaware of the change in the value(s) of D by K_1 operations. This would certainly affect the fidelity of the output. We resolve this problem using the common approach of temporal blocking in GPU stencil applications [15]. An extra layer(s) of values, or *halo layer(s)*, beyond the ones used in the boundary blocks, is initially loaded to the SMEM of each block. The number of layers depends on the radius of the stencil neighborhood. Therefore, the operations of K_1 will be applied to an extra layer(s) of stencil sites at each thread block by using specialized warps [16] to compute the halo layer(s). After the barrier, operations of K_2 will be only applied to the number of stencil sites originally intended per thread block. However, using this method introduces a tradeoff in the size of thread blocks. A larger size would mean a smaller number of redundant halo layer(s) computations and less SMEM bytes used for the total number of stencil sites. By contrast, the larger size would add more strain on the already limited SMEM capacity. Kernel X in Figure 3 is an example of this type of fusion using a halo layer.

III. Scalable Kernel Fusion

The previous sections introduced the rationale for kernel fusion and how to fuse kernels once a decision is made on which kernels to fuse together. This section discusses how to identify the most effective fusions for large-scale programs. To that end, this section formalizes kernel fusion as a combinatorial optimization problem and summarizes a heuristic solver based on evolutionary search techniques.

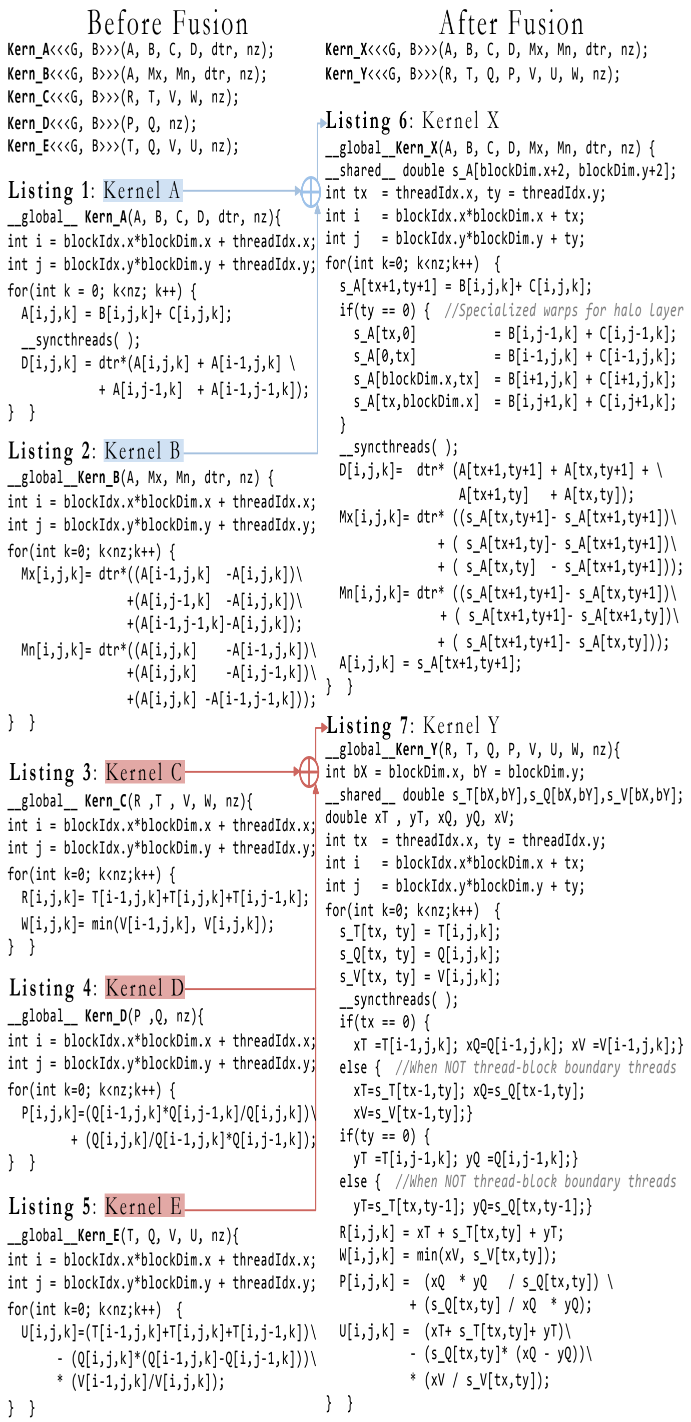
A. Kernel Fusion as a Combinatorial Optimization Problem

The search for the optimal solution for the kernel fusion problem is a permutation that maps original kernels to new kernels. Consider a set K of n original kernels and a cost function (projected runtime bound) $T : \mathcal{P}(K) \rightarrow \mathbb{R}$ that maps the power set of K to the set of reals. The problem is to find m subsets $K_1, \ldots, K_m \subseteq K$ such that a) the subsets are pairwise disjoint and complete (i.e., their union is exactly K), and b) the total cost of the subsets, $\sum_j T(K_j)$, is minimized. Note that m is itself not known. The kernel fusion problem has features similar to classical combinatorial problems (e.g., Bin packing). However, problem-related features, such as data dependencies, render kernel fusion more than just a variation of classical combinatorial problems.

It is usually prohibitive to solve combinatorial problems by exhaustive search. SCALE-LES for example has over a hundred kernels using dozens of arrays. The exhaustive search is estimated to have ∼2.6e45 feasible solutions; every possible permutation of kernel fusion, making exhaustive search prohibitive. Another option is to use a polynomial-time approximation algorithm, e.g., first fit decreasing strategy which first sorts the items to be mapped in decreasing order by their sizes [17]. However, it is not clear how to sort kernels in the kernel fusion problem. There is no clear notion of "size" as in bin packing, for instance. Accordingly, a different approximation algorithm is needed.

B. Problem Formulation

For a set D of l arrays shared over the set K of n kernels, having runtimes of $P(K_1), \ldots, P(K_n)$, applying kernel fusion results in set F of m kernels with projected upper-bound performance resulting in runtime $T(F_1), \ldots, T(F_m)$. The target is to find the m-partition $F_1 \cup \ldots \cup F_m$ of the set $\{K_1, \ldots, K_n\}$ such that a) each new 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 to generate a new kernel F_j and a path exists from K_a to K_b in the



```
Kern_A<<<G, B>>>(A, B, C, D, dtr, nz);
Kern_B<<<G, B>>>(A, Mx, Mn, dtr, nz);
Kern_C<<<G, B>>>(R, T, V, W, nz);
Kern_D<<<G, B>>>(P, Q, nz);
Kern_E<<<G, B>>>(T, Q, V, U, nz);
```

Listing 1: Kernel A

```
__global__ Kern_A(A, B, C, D, dtr, nz){
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.y*blockDim.y + threadIdx.y;
for(int k = 0; k<nz; k++) {
  A[i,j,k] = B[i,j,k]+ C[i,j,k];
  __syncthreads( );
  D[i,j,k] = dtr*(A[i,j,k] + A[i-1,j,k] \
           + A[i,j-1,k]  + A[i-1,j-1,k]);
} }
```

Listing 2: Kernel B

```
__global__Kern_B(A, Mx, Mn, dtr, nz) {
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.y*blockDim.y + threadIdx.y;
for(int k=0; k<nz;k++) {
  Mx[i,j,k]= dtr*((A[i-1,j,k]  -A[i,j,k])\
              +(A[i,j-1,k]  -A[i,j,k])\
              +(A[i-1,j-1,k]-A[i,j,k]);
  Mn[i,j,k]= dtr*((A[i,j,k]    -A[i-1,j,k])\
              +(A[i,j,k]    -A[i,j-1,k])\
              +(A[i,j,k] -A[i-1,j-1,k]));
} }
```

Listing 3: Kernel C

```
__global__ Kern_C(R ,T , V, W, nz){
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.y*blockDim.y + threadIdx.y;
for(int k=0; k<nz;k++) {
  R[i,j,k]= T[i-1,j,k]+T[i,j,k]+T[i,j-1,k];
  W[i,j,k]= min(V[i-1,j,k], V[i,j,k]);
} }
```

Listing 4: Kernel D

```
__global__ Kern_D(P ,Q, nz){
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.y*blockDim.y + threadIdx.y;
for(int k=0; k<nz;k++) {
  P[i,j,k]=(Q[i-1,j,k]*Q[i,j-1,k]/Q[i,j,k])\
       + (Q[i,j,k]/Q[i-1,j,k]*Q[i,j-1,k]);
} }
```

Listing 5: Kernel E

```
__global__Kern_E(T, Q, V, U, nz){
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.y*blockDim.y + threadIdx.y;
for(int k=0; k<nz;k++)  {
  U[i,j,k]=(T[i-1,j,k]+T[i,j,k]+T[i,j-1,k])\
      - (Q[i,j,k]*(Q[i-1,j,k]-Q[i,j-1,k]))\
      * (V[i-1,j,k]/V[i,j,k]);
} }
```

```
Kern_X<<<G, B>>>(A, B, C, D, Mx, Mn, dtr, nz);
Kern_Y<<<G, B>>>(R, T, Q, P, V, U, W, nz);
```

Listing 6: Kernel X

```
__global__Kern_X(A, B, C, D, Mx, Mn, dtr, nz) {
__shared__ double s_A[blockDim.x+2, blockDim.y+2];
int tx  = threadIdx.x, ty = threadIdx.y;
int i   = blockIdx.x*blockDim.x + tx;
int j   = blockIdx.y*blockDim.y + ty;
for(int k=0; k<nz;k++)  {
  s_A[tx+1,ty+1] = B[i,j,k]+ C[i,j,k];
  if(ty == 0) {  //Specialized warps for halo layer
    s_A[tx,0]           = B[i,j-1,k] + C[i,j-1,k];
    s_A[0,tx]           = B[i-1,j,k] + C[i-1,j,k];
    s_A[blockDim.x,tx]  = B[i+1,j,k] + C[i+1,j,k];
    s_A[tx,blockDim.x]  = B[i,j+1,k] + C[i,j+1,k];
  }
  __syncthreads( );
  D[i,j,k]=  dtr* (A[tx+1,ty+1] + A[tx,ty+1] + \
                   A[tx+1,ty]   + A[tx,ty]);
  Mx[i,j,k]= dtr* ((s_A[tx,ty+1]- s_A[tx+1,ty+1])\
                 + ( s_A[tx+1,ty]- s_A[tx+1,ty+1])\
                 + ( s_A[tx,ty]  - s_A[tx+1,ty+1]));
  Mn[i,j,k]= dtr* ((s_A[tx+1,ty+1]- s_A[tx,ty+1])\
                 + ( s_A[tx+1,ty+1]- s_A[tx+1,ty])\
                 + ( s_A[tx+1,ty+1]- s_A[tx,ty]));
  A[i,j,k] = s_A[tx+1,ty+1];
} }
```

Listing 7: Kernel Y

```
__global__Kern_Y(R, T, Q, P, V, U, W, nz){
int bX = blockDim.x, bY = blockDim.y;
__shared__ double s_T[bX,bY],s_Q[bX,bY],s_V[bX,bY];
double xT , yT, xQ, yQ, xV;
int tx  = threadIdx.x, ty = threadIdx.y;
int i   = blockIdx.x*blockDim.x + tx;
int j   = blockIdx.y*blockDim.y + ty;
for(int k=0; k<nz;k++)  {
  s_T[tx, ty] = T[i,j,k];
  s_Q[tx, ty] = Q[i,j,k];
  s_V[tx, ty] = V[i,j,k];
  __syncthreads( );
  if(tx == 0) {
    xT =T[i-1,j,k]; xQ=Q[i-1,j,k]; xV =V[i-1,j,k];}
  else {  //When NOT thread-block boundary threads
    xT=s_T[tx-1,ty]; xQ=s_Q[tx-1,ty];
    xV=s_V[tx-1,ty];}
  if(ty == 0) {
    yT =T[i,j-1,k]; yQ =Q[i,j-1,k];}
  else {  //When NOT thread-block boundary threads
    yT=s_T[tx,ty-1]; yQ=s_Q[tx,ty-1];}
  R[i,j,k] = xT + s_T[tx,ty] + yT;
  W[i,j,k] = min(xV, s_V[tx,ty]);
  P[i,j,k] =  (xQ  * yQ   / s_Q[tx,ty]) \
           + (s_Q[tx,ty] / xQ  * yQ);
  U[i,j,k] =  (xT+ s_T[tx,ty]+ yT)\
           - (s_Q[tx,ty]* (xQ - yQ))\
           * (xV / s_V[tx,ty]);
} }
```

Fig. 3: CUDA example of Kernel fusion. Left side is before fusion and right side is after fusion. Kernels A and B (Listings 1,2) are fused to kernel X (Listing 6). Kernels C, D, and E (Listings 3, 4, 5) are fused to kernel Y (Listing 7). Note a) Kernel X adds halo layer of SMEM to resolve the coherency problem, and b) arrays in uppercase and scalars in lowercase

dependency graph of array D_i, then all kernels in the path from K_a to K_b must also be fused to F_j, i.e. preserve the order-of-execution, d) no new kernel should exceed the capacity of on-chip memory per SMX[1] nor the maximum registers allowed

[1]SMXs are multiprocessors in Nvidia terminology.

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

Minimize
$$\sum_{j=1}^{m} T(F_j) \tag{1}$$

Subject to
$$\sum_{i \epsilon F_k} P(K_i) > T(F_k) \tag{1.1}$$
$$\sum_{j=1}^{m} x_{ij} = 1, \forall i \in \{1, \ldots, n\} \tag{1.2}$$
$$x_{qr} = 1, \forall q \in K_{a \to b}, x_{ar} = 1, x_{br} = 1 \tag{1.3}$$
$$x_{ij} \in \{0, 1\}, \forall i \in \{1, \ldots, n\} \forall j \in \{1, \ldots, m\} \tag{1.4}$$
$$DegKin\,(k_x, k_y) > 0, x \neq y, \forall k_x \forall k_y \in F_k, \forall F_k \in F \tag{1.5}$$
$$MEM\,(F_j) \leq Capacity, \forall j \in \{1, \ldots, m\} \tag{1.6}$$
$$REG\,(F_j) \leq R_{Max}, \forall j \in \{1, \ldots, m\} \tag{1.7}$$

Where

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

Fig. 4: Kernel fusion formulation as an optimization problem

per thread, and e) every kernel must have a *degree of kinship* of more than zero with all other kernels fused with it. A solution is optimal if it has the minimum $\sum_{j=1}^{m} T(F_j)$. Note that for $h \in \{1, \ldots, m\}$, $T(F_h)$ is a dynamic value, i.e., the value is calculated depending on the subset of K that is fused to generate F_h. The canonical form of the problem is shown in Figure 4.

C. Search Heuristic

To decide on the search heuristic to use, two specific features of the kernel fusion problem should be taken into consideration a) strong relationship or dependency existing among the decision parameter, i.e., data sharing among original kernels fused together (Equation (1.5)), and b) the restriction of having all kernels existing in a path in the data dependency graph to be fused together (Equation (1.3)). Hence a solver amenable to multi dependencies and precedences is crucial. Otherwise, infeasible solutions would excessively pollute the search population. Existing generic heuristic solvers (Ex. NEOS Solvers [18]) are naturally not adapted to the above specific features. Hence, a custom search heuristic for kernel fusion becomes necessary.

The search heuristic we developed is an evolutionary computation solver adapted from the Hybrid Grouping Genetic Algorithm (HGGA) [19]. The HGGA is an outgrowth of Genetic Algorithms (GAs), which maintain a population of feasible solutions for which they select a set of individuals,

TABLE III: Metadata of Original Kernel

Parameter	Description
$Blocks_{SMX}$	No. of active blocks per SMX
T_B	No. of active threads per block
Thr	No. of threads per block
B	No. of blocks
R_T	No. of registers allocated per thread
R_{Adr}	No. of registers allocated for indices and addresses
Fl	No. of floating-point operations in kernel
$ThrLD(x)$	Thread load for array x
$Flop(x)$	No. of floating-point operations in relation to array x
$ShrLst$	List of shared arrays in kernel
Hal	Size of the halo region of a thread block (Bytes)

apply genetic operators like mutation and crossover and finally replace old individuals with new ones to create the next generation. This process continues until the algorithm finds an acceptable solution. Note that the most time-consuming step is the evaluation of the objective function, i.e., the performance projection, for every individual in all generations. Hence, a light-weight performance projection model is necessary to run the algorithm in reasonable time.

The HGGA maintains the same working model as conventional GAs but replaces the conventional solution representation of direct assignment such that individual genes represent groups, not the individual items. Hence, every group corresponds to a new kernel or an unfused original kernel; both the groups and candidate solutions would be of variable length. The crossover and mutation operations are accordingly changed so they are aware of groups, i.e., groups are swapped and mutated without disruption. The choice of HGGA is motivated by its proven ability to account for dependencies between decision variables [20]. The crossover and mutation operations acting at the group granularity will not disrupt groups; hence, will pass meaningful information and good groups to offspring chromosomes. The algorithm used in this paper adapts the original HGGA to take into account the dependencies of original kernels inside each group (new kernel). Therefore, multivariate dependencies of original kernels in different *sharing sets* are not violated.

The following pruning scheme was also introduced to the HGGA 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 new kernel, the number of constraints can be decreased. Obviously, checking active constraints will be sufficient for the feasibility check of the current mapping; additionally, if one constraint is violated, the current fusion can be skipped and the algorithm can proceed. The details of the HGGA algorithm and the introduced adaptations are beyond the scope of this paper. The reader can find more details about the algorithm elsewhere [21].

IV. Performance Projection

The objective function of the search heuristic is based on the projected performance of potential new kernels. A straightforward method is to use a cycle-accurate GPU performance prediction model for modeling the new kernels and decide if it is feasible or not to proceed with those fusions. Many have attempted to provide analytical and quantitative methods to predict/project the performance of CUDA kernels such as [22, 23]. The problem with such approaches is the dependency on some level of code representation (code skeleton, PTX code, assembly code, etc.). The need for generating some form of code representation then using the representation generated to predict performance for each solution adds a large overhead to the search of the solution space of fusible kernels rendering the process unscalable for applications with a large number of kernels.

TABLE IV: Features of K20X, K40 and Maxwell GTX 750Ti

Parameter	Description	K20X	K40	GTX750Ti
R_{SMX}	Register resource per SMX	64KB	64KB	64KB
Sh_{SMX}	Max. SMEM per SMX	48KB	48KB	64KB
SMX	No. SMX/SMM	14	15	5
R_{Max}	Max. no. of registers per thread	255	255	255
$P_{Theoritical}$ (TFLOPS)	Theoretical peak performance (Kepler DP, GTX 750 Ti SP)	1.31	1.43	1.38
$GMEM_{BW}$ (GB/Sec)	GMEM bandwidth (STREAM)	202	214	69

Consequently, an appropriate method would be using a codeless performance upper-bound projection for all new kernels at each generation in the search process. Using the popular Roofline model is the simplest option. However, the model is not designed to take into account the effect of pressure on resources due to fused codes, i.e., effect of lower occupancy leading to failure in hiding latency, register pressure, and bank conflicts. A more effective method is to use a *simple model* based on empirical measurement of performance and effective BW of *original kernels*, i.e., measure the time required for data access of *shared arrays* in the *original kernels* and subtract it from the *original sum*. This simple-to-compute model would intuitively be more accurate than the Roofline model. However, the limitation still exists; change in behavior of *new kernels* due to resources pressure not accounted for in the *original kernels*.

To achieve accurate modeling without relying on concrete code representations, we extend the approach for projecting bounds on performance developed by Lai et al. [24] for the memory-bound stencil kernels. Lai et al.'s approach does not project the possible performance from a certain implementation. It projects the performance upper-bound that an application cannot exceed. This approach is focused on compute-bound kernels doing matrix multiplication and much of the analysis goes into identifying the potential performance for compute-bound kernels. On the contrary, we are interested in memory-bound kernels; thus, the analysis is adapted to be focused on having the new kernels use enough active threads to allow the CUDA runtime to effectively hide memory latency. Therefore, the proposed projection model implicitly deduces the practical performance bound depending on CUDA runtime's ability of hiding the latency in a specific kernel.

A. Performance Upper-Bound Projection

For a new kernel F composed from original kernels K_i and $i = 1, \ldots, n$, the goal is to identify the practical upper-bound on the performance of F. This is done on-the-fly for all new kernels in all individuals for each generation in the search process. Initially, the relevant metadata for every original kernel is extracted to be used throughout the search process. The metadata of original kernels are listed in Table III. The model also requires characteristics of the architecture that are collected on real hardware and are independent of the application. Table IV includes the characteristics of the GPUs used in this paper; Nvidia's Kepler K20X, K40, and Maxwell

GTX 750 Ti. The K20X has 192 SPs per SMX. Each SMX has four warp schedulers, each of which has two dispatch units. The K40 differs by increasing one SMX, a slight increase in memory BW and doubling the GMEM capacity. The GTX 750 Ti is not expected to have practical use in HPC due to its greatly reduced double-precision processing speed. However, experiments were done on this device to gain an early insight into the effect of Maxwell's architecture features when fusing kernels of stencil-like workloads. The results are reported for single precision for GTX 750 Ti to avoid the effect of abnormal machine balance if using double precision. The relevant features for Maxwell are a) the increase in SMEM capacity to 64KB by adding the L1 functionality to texture caches, b) doubling the number of maximum active thread blocks leading to the potential for higher occupancy, and c) the registers spilling to L2 cache instead of L1, which increases the performance penalty for kernels with high register pressure.

B. Analysis of Potential Peak Performance

Lai et al.'s approach progresses in the following sequence: identifying the active threads on an SMX, a step-by-step identification of the register and SMEM blocking factors, then end by identifying the potential peak performance. The blocking factors are used in this adaptation, i.e., stencil operation, to ensure no degradation in performance due to lower occupancy for the SMXs that would limit the latency hiding. The same sequence is followed in the adaptation to identify the blocking factors for stencil. However in this adaptation, the original kernels' metadata are used to estimate the performance bound for new kernels, including the penalty for halo layer(s).

$$R_T \leq R_{Max} \tag{2}$$

$$Blocks_{SMX} * Thr * R_T \leq R_{SMX} \tag{3}$$

where R_{Max} is 255 registers. The registers used by active warps are bound by R_{SMX}. The resource constraint for register blocking was defined in the original model as the registers holding the value of sub-matrices multiplied together in each thread. It is changed here to each thread computing the value of one stencil site. The registers required for blocking in a new kernel F are

$$RegFac * max(ThrLD(x)) + c * H_{TH} + 1 < R_T \leq R_{Max}, x \in F^{Pivot} \tag{4}$$

The value c is equal to one if a halo layer(s) was used in the new kernel to maintain coherency; otherwise, equal to zero. For the sake of simplification, we assume $H_{TH} = \lceil \frac{Hal}{Thr} \rceil$ such that the registers required for the halo layer(s) are divided evenly over the threads in the block; therefore, the value is rounded. It should be noted that Hal is calculated according to the stencil operation with the widest radius in the new kernel.

The number of registers needed per thread to hold the stencil neighborhood can range from one register to the total size of the stencil neighborhood. This depends on how the compiler reuses the registers in a specific kernel. Understanding the register reuse in the *nvcc* compiler is futile due to the complication of the register allocation process and the numerous factors involved. We define a reuse factor $\frac{1}{max(ThrLD(x))} \leq RegFac \leq 1$ to refrain the model from being limited to a specific kernel, compiler, or ISA. Micro-benchmarks of relevant stencils showed a low reuse for Kepler (CUDA 5.0, compute capability 3.5 and *nvcc* V0.2.1221 generating $RegFac \approx 0.85$) with a slight improvement in Maxwell. Low reuse is likely because reusing registers would prevent memory load pipelining by the runtime. There are no direct data lanes between the GMEM and SMEM memory spaces. To overlap the data transfer and computation, the extra registers to fetch data from GMEM to SMEM are

$$R_{fetch} = 1 + c * H_{TH} \tag{5}$$

Each thread loads one value in each iteration, and registers are assumed to be reused for GMEM to SMEM data fetching; a single register for all the data fetched by a single thread. All threads in a block divide registers needed for the halo layer(s). Considering R_{adr} registers for storing the address of matrices in the SMEM and GMEM as the original model proposed and using Equations (4,5), the constraint for registers (satisfying the constraint in Equation (1.7)) is

$$R_{fetch} + RegFac * max(ThrLD(x)) + c * H_{TH} + R_{adr} + 1 < R_T \leq R_{Max} \tag{6}$$

For $Blocks_{SMX}$ active blocks, $(1 + c * H_{TH}) * T_B * Blocks_{SMX} * Size(ShrLst)$ is needed to store GMEM data and is bounded by the SMEM capacity, i.e. capacity constraint in Equation (1.6)

$$(1 + c * H_{TH}) * T_B * Blocks_{SMX} * Size(ShrLst) + B_{conf} \leq Sh_{SMX} \tag{7}$$

Note that the number of active threads T_B is equal to the least T_B in the original kernels. Due to erratic SMEM bank conflicts resulting from fusion, the number of bytes required for padding B_{conf} is taken into consideration to prevent new kernels from using the maximum available SMEM without leaving enough space for padding. The Kepler architecture for example requires a factor of $1/32$ of the used memory for padding (32 banks at 8 bytes of access granularity). Detecting the SMEM access patterns in the code that can trigger bank conflicts follows the model in [25]. For the next equation, the SMEM blocking factor B_{Sh} is a direct function of the number of active threads, halo size, and number of shared arrays

$$B_{Sh} = T_B * Blocks_{SMX} / ((1 + c * H_{TH}) * Size(ShrLst)) \tag{8}$$

For $B_{eff} = B_{Sh} * SMX / (Thr * B)$, the performance bounded by GMEM bandwidth is estimated to be

$$P_{MemBound} = \frac{B_{eff} * GMEM_{BW}}{8} \text{ GFLOPS} \tag{9}$$

The potential peak performance depends on the effectiveness of blocking B_{eff} and not the operational intensity, as in the Roofline model. B_{eff} conservatively projects the performance bound by adapting the blocking factor to maintain the count of active threads. Note that for single precision, the used SMEM (in bytes) and number of registers should be adjusted and the denominator in the last equation is changed. The following is an example extracted from an actual fusion using K20X: a fusion of three kernels sharing two arrays with a single halo layer needed for one of the kernels. For a min. T_B of 86 out of Thr at 128, Hal of 32 points and $Blocks_{SMX}$ of 32 at B of 64. B_{SH} would equal 688 according to Equation (8). Knowing the operational intensity of the new kernel by profiler measurement to be 0.195, the projected performance is estimated at 75.8% $\left(\frac{688*14*202}{8*128*64}\right)$ of the theoretical peak of

Algorithm 1 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 new kernels not violating order-of-execution
4: For all $M$, for new kernel $f \in F$:
      i: Aggregate metadata of original kernels fused to $f$
      ii: Project 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 are not met, go to step 4
9: Use values of best solution as guide for fusing kernels
```

39.39 GFLOPS computed according to the Roofline model.

Linking back to the motivating example in Figure 3, using the simple model or Roofline for projecting the performance bound would favor fusing Kernels C, D, and E to Kernel Y, as shown in the figure. However, running the code in the listings on K20X as a micro-benchmark showed degradation. The runtime of Kernel Y measured at 554μs compared to the aggregate 519μs of kernels C, D and E despite the Roofline projecting a bound yielding 336μs and the simple model at 410μs. The proposed model on the other hand projected 564μs due to high pressure on SMEM, which reduced the number of active thread blocks.

A new kernel has a number of FLOPs equal to the sum of FLOPs for the original kernels plus FLOPs from the computation of thread block halo layers(s) that might have been added to maintain coherency. If M is the portion of original kernels that were fused to a new kernel and must compute a halo layer(s), N is the remaining portion of original kernels fused into the same new kernel, and S is the set of data arrays that required a halo layer(s). The objective function in Equation (1) is defined as the new kernel's projected runtime bound (in seconds) as follows

$$T_{pro} = \frac{(\sum^{N} Fl + \frac{\sum^{M} Flop(x)*Hal}{B*Thr}) * 10^{-9}}{P_{MemBound}}, x \in S \quad (10)$$

V. Performance Improvement by Kernel Fusion

Achieving an actual speedup by kernel fusion combines all the components discussed in the earlier sections: extracting the metadata and dependency graphs, the search heuristic, using the performance bound model to guide the search process, and finally applying the code transformation. Algorithm 1 shows the steps for improving performance by kernel fusion. While our use of this algorithm is tailored to our specific problem domains by using the bandwidth-focused projection model, the algorithm is scalable and flexible in the sense of having no problem-specific steps that would limit the applicability to a specific application. Note also that the components can be changed. For example, the performance projection method (step 4.ii) used as an objective function can be replaced with any performance-bound model (e.g. Roofline model). The search heuristic can also be replaced if required.

TABLE V: Attributes of 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

A. Components of Proposed Method

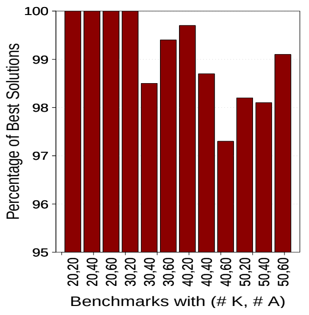
Different components of the proposed method were constructed using different tools. The data dependency is generated using the ROSE compiler. The graph is constructed by a pass that identifies the kernels and the data arrays used in those kernels. For the kind of applications targeted in this paper, using ROSE APIs to extract the graph is relatively simple (mainly due to the prominent use of Fortran). Extracting the graph for C/C++ codes using complex data representations, e.g. pointer alias, is beyond the scope of this paper. The order-of-execution graph is automatically generated from the data dependency graph using a program implemented in C++. Extracting the kernel metadata was partially manual in this study and is under effort to be fully automated through static analysis and code instrumentation. The search heuristic in Section III-C generates a list of proposed fusions. The actual fusion is done manually in the reported results. An automated fusion by a source-to-source code transformation as a part of a stencil domain-specific language is considered for future work.

VI. Results and Evaluation

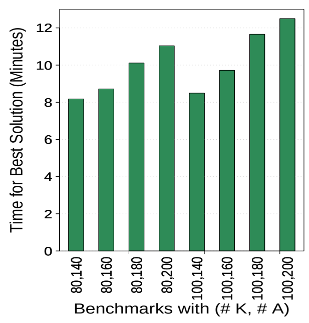
The main goals of this section are a) evaluate the speedup achieved by applying kernel fusion, and b) quantify the limitations leading to the missed portion of reducible GMEM traffic, e.g., SCALE-LES had 41% reducible traffic while 26% reduction in runtime was only achieved leaving 15% of wasted opportunity of improvement. Each of the following sections quantifies the effectiveness of an individual component, e.g., performance model, and highlights the contribution of that component to the missed portion of reducible memory traffic. The search heuristic section evaluates the search heuristic in terms of solution quality and runtime. The following section, performance-bound model, discusses the accuracy in projecting upper-bounds on performance. The performance and speedup section analyzes the performance improvement attributed to kernel fusion. The last section discusses how the actual reduction in memory traffic after fusion is translated into speedup.

A. Systems and Specifications

Tsubame2.5 supercomputer was used in the Kepler fusion experimentation. As for the software, PGI 14.3 CUDA Fortran and CUDA 5.5 were used for the Kepler K20x and K40 GPUs and CUDA C 6.0 for the Maxwell GTX 750 Ti GPU. The test suite kernels are written in CUDA C and the real-world applications are written in CUDA Fortran. Since CUDA Fortran had no support for the Maxwell architecture at the time this paper was written, Maxwell results are reported for the test suite only. Performance results are averaged for 10 runs compiled at the highest possible compiler optimization. All results reported are for a single node execution. Stencil-based scientific applications widely favor weak scaling to be the major evaluation criterion, e.g., adding new nodes to a weather application means expanding the 3D grid atmospheric space



(a) Percentage of best solutions. Y-axis ranges from 95% to 100%



(b) Time to best solution

Fig. 5: Percentage and average runtime to best solution for different test suite benchmarks

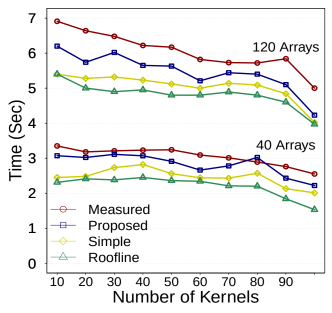
in the horizontal direction. Hence, the speedup achieved on a single node is expected to carry over to multi-node execution for applications having almost linear weak scaling.

The search algorithm was implemented in C++, parallelized by OpenMP and executed using an Intel Xeon X5670 2.93GHz machine (8 cores). The search heuristic was run 10 times for each application to confirm the same best solution was repeatedly found. The best solution was then used as the basis for manually fusing kernels.

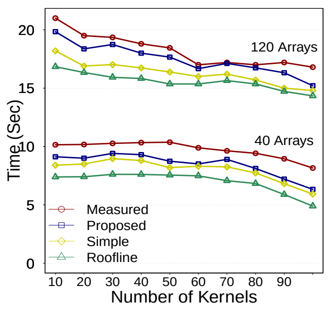
B. Benchmarking and Applications

1) Test Suite From Mini-App: CloverLeaf: CloverLeaf [26] 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; each kernel loops over the entire grid and updates one or more mesh variables based on a kernel-dependent computational stencil (The standard problem size is 962^2 cells). A test suite based on the kernels from CloverLeaf was constructed to help understand the factors affecting kernel fusion performance gains through a controlled study of benchmarks with different features. Table V summarizes 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 the 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*.

2) Real-world Applications: SCALE-LES and CAM-HOMME: SCALE-LES [27] is a next generation weather model designed to confirm the effects of grid spacing, domain size, aspect ratio, and precision on its accuracy. Investigating such issues requires a huge amount of computational resources beyond those of the current systems. Only the relevant features of SCALE-LES are discussed due to space limitation. For reference, [1] 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 perform stencil-based operations. The original kernels were rigorously optimized by ensuring all off-chip GMEM accesses were coalesced, using the SMEM for kernels that have data elements accessed by more than one thread in a thread block, proper alignment of warps with respect to different bounds of loops, and maintaining enough occupancy to ensure enough active warps in flight. The reported speedup of the version after kernel fusion is for the entire program, i.e., data transfer is accounted for.



(a) Kepler K40



(b) Maxwell GTX 750 Ti

Fig. 6: Measured and projected runtime (*Thread load* = 8)

HOMME is the dynamical core within the Community Atmospheric Model (CAM). The discretization uses a spectral element method based on a continuous Galerkin finite element method. Porting HOMME to GPU [6] was evaluated as a part of preparing CAM for post petascale machines. All modules of HOMME are not fully ported to GPU, e.g., packing and unpacking buffers at the boundary exchange remains on the CPU. The routines in the HOMME dynamical core dominating the runtime and tracer advection were ported to GPU by moving the parallelism down the call tree. The speedup reported in this paper is for the kernels already ported in the dynamical core. This is to avoid the skew in runtime expected from frequent PCIe transfer, which is to be eliminated when HOMME is entirely run on the device, as with SCALE-LES.

C. Search Heuristic

1) Test Suite: Figure 5a shows the percentage of best solutions found for benchmarks with variation in the *thread load* and *sharing set* cardinality according to Table V. To evaluate the effectiveness of the search heuristic in finding the optimal solutions, the search results for benchmarks of small sizes were verified using a deterministic method. Figure 5b shows the time used by the search heuristic for the largest benchmarks on the test suite. The stop criterion for the search algorithm is to show no change in solution for a predefined number of iterations. Changing the stop criterion to be a predefined number of generations to reduce the run time is possible, but comes at the expense of the quality of solutions.

2) Applications: Table VI lists the performance metrics and parameters of the search algorithm for a) SCALE-LES (142 original kernels and 65 *sharing sets*, estimated to have ∼2.6e45 feasible solutions) and, b) HOMME dynamical core (43 original kernels and 29 *sharing sets*). Profiling the main new kernels showed they used SMEM at the highest possible capacity without dropping occupancy significantly from those of the original kernels. Only three out of the main new kernels in SCALE-LES and two in HOMME did not use the SMEM at

TABLE VI: Performance & Parameters of Search Algorithm

Application	*No. of Generations*	*Population Size*	*Total # Evaluations*	*Average Runtime*
SCALE-LES	2000	100	5.4e6	9.51 Min.
HOMME	1000	100	2.7e6	6.11 Min.

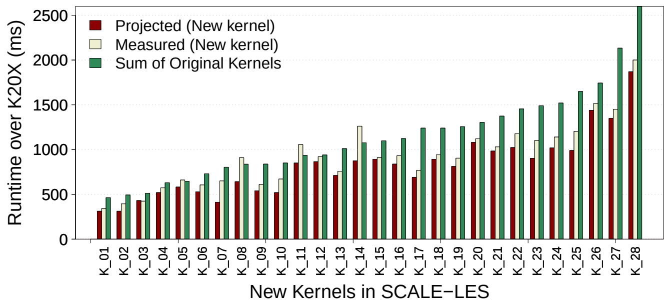


Fig. 7: Runtime on K20X for measured, projected and *original sum* of news kernels in SCALE-LES

maximum capacity. Note the relatively short runtime compared to the total number of evaluations of the objective function. Using a performance model that requires code representation would have increased the runtime significantly. For example, using the exhaustive search method in GROPHECY [28] 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 measured at 3ms.

D. Performance Upper-Bound Projection Model

1) Test Suite: Figure 6 shows the runtime of the benchmarks compared to the time bound estimated using the Roofline model, the *simple model* based on empirical observation, and the *proposed model* introduced in Section IV-A. Note that with the increased number of kernels, the performance model sustains inaccuracy within an acceptable range. Note also the significant difference in the Roofline model and simple model values included in the figure from the proposed upper-bound projection model. The use of the Roofline or simple model as an objective function would have included search solutions overly loaded with false positives, i.e., solutions that are feasible according to the search heuristic but show no speedup when new kernels are executed, as shown earlier in the motivating example. The new kernels executed on GTX 750 Ti show higher projection accuracy with the decrease in the number of arrays. Profiling those kernels showed that SMEM was not used near full capacity unlike the benchmarks with a larger number of arrays. This happens as the decreased number of arrays in the program increases the order-of-execution constraints and reduces the ability of the search algorithm in reaching larger new kernels despite the availability of SMEM capacity. Hence, with less pressure on resources, the proposed projection model approaches the accuracy of the simple model.

2) Applications: The best solution for SCALE-LES was fusing 117 out of the 142 kernels into 38 kernels; an average of ~ 3 original kernels transformed to one new kernel. For the 43 kernels of HOMME, the best-found solution was to fuse 22 kernels into 9 kernels. After implementing all fusions, 4 out of the 38 of SCALE-LES new kernels had a runtime higher than the *original sum* compared to one in HOMME. Figures 7 and 8 show the measured, projected, and *original sum* runtimes for the new kernels in increasing order of kernel execution time. Profiling the four unproductive new kernels in SCALE-LES showed that they have in common the feature of relatively high *thread load* for the *kernel pivot*, i.e., the original kernels had stencils with a relatively large neighborhood size leading to register pressure.

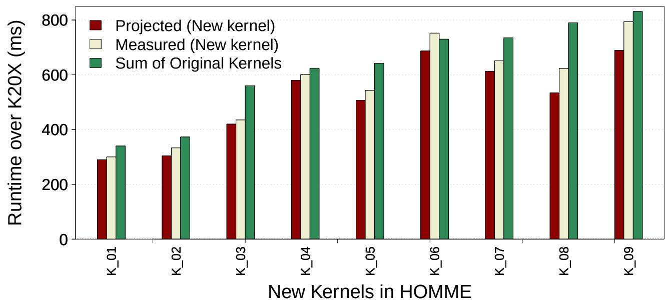


Fig. 8: Runtime on K20X for measured, projected, and *original sum* of new kernels in HOMME

E. Performance and Speedup Analysis

1) Test Suite: Analyzing the speedup is used to evaluate the proposed method's effectiveness for different patterns of data dependency. Figure 9 shows the speedups achieved from fusing the benchmarks built from CloverLeaf. Maxwell exhibited higher speedup compared to Kepler mainly due to the increase in SMEM capacity resulting in larger new kernels. Note that the lower number of arrays enforces a more strict order-of-execution, especially for low kernel count; hence, lower speedups. However, the effect of stricter order-execution has less impact on reducing the speedup for Maxwell. By analyzing this type of fusions, it appeared that the higher SMEM capacity allowed the search algorithm to favor *complex fusions* having intra-kernel dependencies requiring extra SMEM for halo layer(s) over *simple fusions*.

2) Applications: Table VII lists SCALE-LES and HOMME speedups for the original GPU version vs. the version after kernel fusion. The problems size for SCALE-LES is 1280x32x32 and HOMME is 4x26x101. The GPU version with fusion for SCALE-LES and HOMME shows over 1.35x and 1.2x speedup for a single node, respectively. SCALE-LES and HOMME were reported to have an almost linear weak scaling [1, 6] over a spectrum of different systems; hence, a decrease in runtime for a single node would yield almost the same decrease in runtime when using multiple nodes (assuming overlapped computation and communication).

Our projection model allows us to study potential performance improvements with hypothetical architecture configurations. One interesting scenario is the effect of SMEM capacity on performance improvement. By running the model with 128KB and 256KB for SCALE-LES, data reuse by kernel fusion projects 1.56x and 1.65x improvements, respectively. Although the increased capacity would also imply architectural trade-off, the speculative study with our modeling enables us to gain interesting observations on the impact of future architectures.

F. Data Reuse and Performance Improvement

To quantify how the reduction in GMEM operations compares to a corresponding increase in performance, we define the metric *Fusion Efficiency*. It compares the reduction in runtime of new kernel F to the reduction in the number of GMEM memory operations resulting from fusion. For LD_i

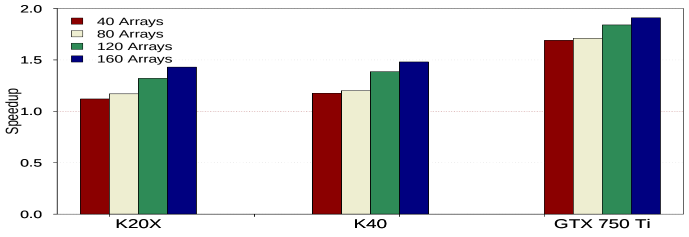


Fig. 9: Test suite speedups (Thread load = 8)

and ST_i the number of loads and stores in kernels K_i such that $i = (1, \ldots, n)$ that are fused to F, the maximum theoretical gain in performance is

$$\frac{LD_F + ST_F}{\sum_{i=1}^{n} LD_i + ST_i} \quad (11)$$

Note that the above value is just a theoretical limit that builds on the assumption of the Roofline model in which all computation is totally hidden by being overlapped with memory operations. With T being the runtime, the *Fusion Efficiency* can be defined as

$$FE = \frac{LD_F + ST_F}{\sum_{i=1}^{n} LD_i + ST_i} \Bigg/ \frac{T(F)}{\sum_{i=1}^{n} T(K_i)} \quad (12)$$

Fusion Efficiency of 1.0 means that reduction in runtime is equal to the reduction in the memory traffic to the GMEM. However, *Fusion Efficiency* would typically be less than ideal. The new kernels of the test suite, SCALE-LES and HOMME have an FE ranging between 87%∼96%. The FE was observed to increase slightly for Maxwell, likely due to the reduction in instruction latencies and improved instruction scheduling. There is space for improving the FE by implicitly addressing the limitations on efficiency through the proposed performance projection model. The inefficiency fraction is attributed to a) latency caused by the new SMEM operations for reusing the shared data arrays, b) different operations from different original kernels acting at different loop bounds when aligned would generate diverging warps, c) possible decrease in occupancy due to register pressure leading to lower ability in hiding latency, d) overhead of barriers between codes of original kernels having an order-of-execution, especially if many barriers are needed, and e) there is a small chance for different SMXs obtaining a hit in L2 cache since the blocks executed are computing the stencils from random positions, yet chances for obtaining a hit in L2 would occur even less in new kernels.

VII. Related Work

Improving GPU performance by fusing kernels is introduced in different contexts. However, extending the existing approaches would be challenging due to a) the use of non-scalable methods to search the space of possible fusions, e.g., exhaustive search, rendering the kernel fusion prohibitive in large problems (i.e., large number of kernels and data arrays), and b) the use of computationally expensive performance projection methods that require some form of code representation which further limits scaling to large problems. The following is a review of notable GPU kernel fusion related work.

Kernel fusion/fission was proposed [8] to reduce the impact of the limited PCIe bandwidth for codes using relational algebra operators. The study emphasized the effect of the number of operations on the fusion process with less regard to fusing multiple kernels. GROPHECY [28] is a GPU performance projection tool that uses CPU code skeletons. The tool is further used to fuse kernels [9]. GROPHECY uses a high level abstraction to model the performance of CPU codes by a brute-force search for code layouts. It was shown to be effective for the reported fusion of three or less kernels.

TABLE VII: SCALE-LES and HOMME Speedups After Kernel Fusion

	K40	K20X
SCALE-LES	1.35x	1.32x
HOMME	1.20x	1.18x

Fousek et al. [29] proposed a decomposition-fusion scheme that decomposes a computational problem to be solved by several simple functions implemented as standalone kernels then fuse some of these functions later into more complex kernels to improve memory locality. The scheme was reported to be effective, yet not appropriate for modeling applications having kernels of different resources requirements and data access patterns. The work introduced in [30] uses a Map-Reduce approach to fuse kernels composed of linear algebra operations. Representing BLAS functions as Map and Reduce operations simplified the search process at the expense of bounding the kernels to a fixed set of problem-specific elementary functions.

Fusing kernels was also proposed [31] to optimize for power. This study was not motivated by exposing memory locality; the motivation of kernel fusion was to achieve higher utilization and a more balanced demand for hardware resources, which provides more potential for power optimization. The problem was represented as a dynamic programming problem of fusing a small number of kernels. Reported fusion of two kernels was shown to be rewarding for power optimization.

VIII. Conclusion

Scientific HPC applications are increasingly ported to GPUs to benefit from the high throughput of graphic cards. Many of these applications, e.g., weather models, include dozens of kernels of stencil operations bound by bandwidth. We proposed a scalable method for kernel fusion to improve the performance of this class of applications. Kernel fusion works through exposing data localities and using the potential for improvement through data reuse. However, large problems, complex data dependencies, kernels order-of-execution, and physical constraints on the on-chip memory used for data reuse make the problem nontrivial.

We defined the kernel fusion problem as a combinatorial optimization problem and proposed a method to solve large problems guided using our performance-bound projection model. The introduced HGGA well suits the kernel fusion optimization problem, which exhibits a high level of data dependencies between kernels. Our performance-bound projection model, which requires no form of code representation, was used to overcome the overhead of conventional performance prediction models. Such overhead renders the search process prohibitive for applications with a large number of kernels.

The proposed method was evaluated using a test suite and two next-generation weather models: SCALE-LES and CAM-HOMME. The proposed method was proven scalable to large benchmarks in the test suite, and applicable to SCALE-LES and HOMME yielding 1.35x and 1.2x speedups. Future work includes automating the process by using annotations to guide an end-to-end code transformation as an extension to an existing domain-specific language designed for stencil codes.

Acknowledgments

This project was partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

References

[1] M. Wahib and N. Maruyama, "Highly Optimized Full GPU-Acceleration of Non-hydrostatic Weather Model SCALE-LES," ser. CLUSTER'13, 2013, pp. 77–85.

[2] J. Michalakes and M. Vachharajani, "GPU Acceleration of Numerical Weather Prediction," *Parallel Processing Letters*, vol. 18, pp. 531–548, 2008.

[3] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-Fold speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code," ser. SC'10, 2010, pp. 1–11.

[4] M. Bianco, T. Diamanti, O. Fuhrer, T. Gysi, X. Lapillonne, C. Osuna, and T. Schulthess, "A GPU Capable Version of the COSMO Weather Model," ser. ISC'13, 2013, pp. 34–35.

[5] I. Demeshko, N. Maruyama, H. Tomita, and S. Matsuoka, "Multi-GPU Implementation of the NICAM Atmospheric Model," *Lecture Notes in Computer Science*, vol. 7640, pp. 175–184, 2013.

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

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, pp. 65–76, 2009.

[8] H. Wu, G. F. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. T. Chakradhar, "Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission," in *IPDPS Workshops*, 2012, pp. 2433–2442.

[9] J. Meng, V. A. Morozov, V. Vishwanath, and K. Kumaran, "Dataflow-driven GPU Performance Projection for Multi-kernel Transformations," ser. SC'12, 2012, pp. 1–11.

[10] C. Ding and K. Kennedy, "Improving Effective Bandwidth Through Compiler Enhancement of Global Cache Reuse," *J. Parallel Distrib. Comput.*, vol. 64, pp. 108–134, 2004.

[11] S. Mehta, P.-H. Lin, and P.-C. Yew, "Revisiting Loop Fusion in the Polyhedral Framework," ser. PPoPP'14, 2014, pp. 233–246.

[12] M. Schordan and D. Quinlan, "A Source-To-Source Architecture for User-Defined Optimizations," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science, 2003, vol. 2789, pp. 214–223.

[13] W. C. Skamarock and J. B. Klemp, "A Time-split Nonhydrostatic Atmospheric Model for Weather Research and Forecasting Applications," *J. Comput. Phys.*, vol. 227, pp. 3465–3485, 2008.

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

[15] M. Christen, O. Schenk, P. Messmer, E. Neufeld, and H. Burkhart, "Accelerating Stencil-Based Computations by Increased Temporal Locality on Modern Multi- and Many-Core Architectures," ser. HipHaC'08, 2008, pp. 47–54.

[16] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization," ser. SC'11, 2011, pp. 12:1–12:11.

[17] R. E. Korf, "A New Algorithm for Optimal Bin Packing," in *Eighteenth national conference on Artificial intelligence*, 2002, pp. 731–736.

[18] "NEOS Server for Optimization," http://www.neos-server.org/neos/, 2014.

[19] E. Falkenauer, "A Hybrid Grouping Genetic Algorithm for Bin Packing," *J. of Heuristics*, vol. 2, pp. 5–30, 1996.

[20] K. Praditwong, "Solving Software Module Clustering Problem by Evolutionary Algorithms," in *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, 2011, pp. 154–159.

[21] M. Wahib and N. Maruyama, "A Hybrid Grouped Genetic Algorithm for Optimizing GPU Kernel Fusion," http://mt.aics.riken.jp/publications/GGA.pdf, 2014.

[22] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," ser. ISCA'09, 2009, pp. 152–163.

[23] S. Baghsorkhi, I. Gelado, M. Delahaye, and W. Hwu, "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors," ser. PPoPP'12, 2012, pp. 23–34.

[24] J. Lai and A. Seznec, "Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs," ser. CGO'13, 2013, pp. 1–10.

[25] C. Gou and G. Gaydadjiev, "Addressing GPU On-Chip Shared Memory Bank Conflicts Using Elastic Pipeline," *International Journal of Parallel Programming*, vol. 41, pp. 400–429, 2013.

[26] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA," ser. SCC'12, 2012, pp. 465–471.

[27] Y. Sato, S. Nishizawa, H. Yashiro, Y. Miyamoto, and H. Tomita, "Potential of Retrieving Shallow-Cloud Life Cycle from Future Generation Satellite Observations through Cloud Evolution Diagrams: A Suggestion from a Large Eddy Simulation," *SOLA*, vol. 10, pp. 10–14, 2014.

[28] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU Performance Projection from CPU Code Skeletons," ser. SC'11, 2011, pp. 1–11.

[29] J. Fousek, J. Filipovic, and M. Madzin, "Automatic Fusions of CUDA-GPU Kernels for Parallel Map," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 98–99, 2011.

[30] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska, "Optimizing CUDA Code By Kernel Fusion—Application on BLAS," *CoRR*, vol. 1305.13, 2013.

[31] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *GreenCom'10*, 2010, pp. 344–350.