Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications

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

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

ABSTRACT

This paper proposes an end-to-end framework for automatically transforming stencil-based CUDA programs to exploit inter-kernel data locality. The CUDA-to-CUDA transformation collectively replaces the user-written kernels by auto-generated kernels optimized for data reuse. The transformation is based on two basic operations, kernel fusion and fission, and relies on a series of automated steps: gathering metadata, generating graphs expressing dependencies and precedency constraints, searching for optimal kernel fissions/fusions, and generation of optimized code. The framework is modeled to provide the flexibility required for accommodating different applications, allowing the programmer to monitor and amend the intermediate results of different phases of the transformation. We demonstrate the practicality and effectiveness of automatic transformations in exploiting exposed data localities using a variety of real-world applications with large codebases that contain dozens of kernels and data arrays. Experimental results show that the proposed end-to-end automated approach, with minimum intervention from the user, improved performance of six applications with speedups ranging between 1.12x to 1.76x.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Frameworks; D.3.4 [**Programming Languages**]: Processor—*Code generation, Optimization*

Keywords

GPU; stencil computations; source-to-source translation; CUDA

1. INTRODUCTION

To achieve performance improvement beyond the memory wall, memory-bound applications are in an increasing need for advanced optimizations such as kernel fusion. Kernel fusion is an advanced optimization at which codes of different memory-bound kernels are aggregated to expose inter-kernel data localities. Performance can then potentially improve if opportunities of data reuse in the transformed code are exploited effectively. Applying kernel fusion to real-world large-scale applications, however, is difficult. A main challenge is the need for a scalable method to search for the optimal choice of which kernels to fuse among the enumeration of feasible fusions. Real-world large applications can have over a hundred kernels using dozens of data arrays, as will be shown in the results section (Table 1). For example, SCALE-LES [29] weather model is estimated to have $\sim$2.6e45 feasible combinations of kernels to fuse together. Another main challenge is to design an architecture-aware method for collectively applying fusion as a transformation: exposing locality does not guarantee actual performance improvement unless the architecture-related features were taken into consideration when optimizing for data reuse. Finally, a transformation method for applying kernel fusion that is automated and applicable to a diversity of applications is a non-trivial challenge.

This paper addresses GPU kernel fusion for a class of memory-bound applications[1]. In particular, Partial Differential Equation (PDE) solvers comprise a significant fraction of scientific applications in a variety of areas. Those applications are often implemented using memory-bound finite-difference techniques: iteratively sweeping over a spatial grid to perform nearest neighbor computations called stencils. Stencil codes such as the Jacobi and Gauss-Seidel kernels are used in many scientific and engineering applications. Optimized stencil codes in GPU typically depend on the on-chip memory to enhance performance [19]. Kernel fusion further optimizes the use of on-chip memory in stencils and can be increasingly important in the future of performance optimization driven by the increased dependence on on-chip memory. Note that the effective on-chip memory capacity in Nvidia GPUs, for instance, has increased from one generation to another with the trend expected to continue.

The ideal realization of improving the performance of real-world GPU applications via kernel fusion is based on scalable and architecture-aware end-to-end automated transformation. Moreover, this automated transformation should be usable by stencil applications from different domains in science and engineering. The main requirements of the ideal realization are: a) expressing the complicated dependencies of kernels and data arrays in the large codebases of real-world applications, b) a scalable and robust method for identifying the prospective fusions that would give best locality, c) a method for generating code that would apply fusion such that the exposed localities lead to actual performance improvement, d) implementing the prospective fusions collectively for the entire application, and e) most importantly, using a transformation method that is automated and flexible in order to be generic (i.e. beneficial for applications from different areas).

[1]Kernel fusion is a general concept, however, GPU is the target architecture in this paper

HPDC'15, June 15–19, 2015, Portland, Oregon, USA

DOI: http://dx.doi.org/10.1145/2749246.2749255

The work in this paper targets this ideal realization: make the process of applying kernel fusion to improve performance a practical choice for real-world applications. The paper's main proposal is an end-to-end automated kernel transformation framework. The end-to-end solution includes the following steps. First, gathering of metadata via instrumentation and static analysis to understand the dependencies among kernels. Second, searching for the optimal fusion of kernels using a customized optimization algorithm. Third, auto-generating new optimized kernels to replace the original kernels. A positive consequence of the automated approach is an opportunity for improving the optimization algorithm. More specifically, this paper proposes the use of kernel fission to improve the search space exploration by increasing the number of feasible solutions through relaxation of the on-chip memory capacity constraint. This in turn increases the chances of finding kernel fusions that could expose higher locality.

Attending to applications from different domains having different levels of complexity is a challenge. Hence we provide the programmer with the option to guide the transformation. The programmer-guided optimization of the transformation works as follows. Initially, The programmer is provided with a report on the output of each phase including hints of possible inefficiencies. Next, the programmer can intervene by changing the output of any given stage before passing it to the next stage. It is worth mentioning that the programmer can guide the framework to execute until/from any given stage of the transformation. To clarify, in this paper, *automated transformation* is analogous to compilation and *programmer-guided transformation* is analogous to how the programmer can manually intervene at any of the compilation steps, e.g., using pragma directives in the source code to guide the compiler or changing the source code so that the compiler can automatically vectorize the code.

We build on previous work [28], which took the following steps. First, we formulated the kernel fusion problem as a combinatorial optimization problem. Second, we modeled the projected performance bound of prospective fusions. Third, it was demonstrated that manually applying kernel fusion is effective in enhancing performance for both a test suite and two real-world weather models. However, the steps taken in the previous work had limitations that hindered the ideal realization. First, massive effort is required in applying the transformation by manually rewriting the kernels, which includes non-trivial on-chip memory optimizations. Second, as an inherent nature of code transformation, modifications and expansion in the original codebase means redoing the whole process from the beginning. Third, the manual transformation inhibits the possibilities for improving the transformation when the manual efforts for those transformation improvements grow exponentially with the number of kernels and arrays in the application (kernel fission is an example). Therefore, manually applying kernel fusion is prohibitive vertically, for continuously growing applications, and horizontally, in limiting generality with respect to diversity in applications' features and areas. To summarize, the transformation method is not complete if not automated and not amenable for use by a variety of applications having different features. The main contributions in this paper are:

- A framework for automatically exploiting data locality exposed via a transformation based on kernel fission/fusion. The framework is modeled such that the programmer has the option to guide the transformation; the programmer can intervene at any of the transformation phases and change the intermediate results (Section 3).

  The transformation framework contains: a) a metadata gathering tool based on instrumentation and static analysis, b) a tool for creating and optimizing the data dependency and order-of-execution graphs, c) an optimization algorithm for identifying optimal fissions/fusions, and d) a code generator that creates the new kernels (Section 5).

- Capitalizing on the utility of automation, this paper proposes the use of kernel fission to relax the constraints on the search space explored for kernel fusion, which in turn increases the number of feasible kernel fusions and solutions with better localities. In the same context, tuning thread block size at the final transformation phase is also introduced (Section 4).

- We evaluate the proposed end-to-end solution by applying automated kernel transformations to enhance the performance of six real-world scientific applications in such diverse areas as weather modeling, seismic simulations, oceanic circulation modeling, electromagnetics, and fluid dynamics. The transformation resulted in speedups ranging between 1.12x to 1.76x compared to the original CUDA codebases.

  To assess the effectiveness of the automated approach, we compare the achieved speedup of two of the applications against the speedup of the manual approach used in the previous work. The automated framework is shown to achieve more than 85% of the performance of the previous manual approach. Applying a programmer-guided transformation by changing the transformation intermediate results drove the performance up to 92%. Investigation revealed the loss in performance was a result of sub-optimal code generation when fusing: a) kernels having deep nested loops, and b) kernels having highly varied loop sizes.

  In a separate experimentation, using kernel fission and tuning block size overly improved the effectiveness of kernel fusion and resulted in speedups higher than the manual approach which relied on kernel fusion only (Section 6).

2. BACKGROUND

The following is an overview of terms that will be frequently used. *kernel fusion* is the aggregation of the codes of several kernels into a single new kernel. Next, shared memory[2] is used to exploit exposed localities and hence improve performance through data reuse. In an application with dozens of kernels, those *original kernels* that use *data arrays* residing on the device memory are fused to generate *new kernels*. Data Dependency Graph (*DDG*) is a DAG composed of vertices that can be either kernels or data arrays to reveal the data inter-dependencies between kernels. Order-of-Execution Graph (*OEG*) is a DAG composed of vertices that represent the kernels and edges that correspond to the inter-kernel precedence that should not be violated.

This paragraph briefly overviews the scalable method for GPU kernel fusion introduced in previous work [28]. Initially, the search for the best prospective combination of kernels to be fused together was formulated as an optimization problem and a customized Grouped Genetic Algorithm (GGA) was used to solve the optimization problem. Next, a performance projection model for stencil kernels was derived to evaluate if any given fusion would potentially improve the performance or not. The performance model is codeless to be lightweight, i.e., relies on performance data and no form of code abstraction. Operating the optimization algorithm requires the following: a) *problem-related constraints* extracted from DDG and OEG, b) *architecture-related constraints* extracted from metadata

[2]Nvidia's term for on-chip scratchpad memory

related to the device and stencil operations in the kernels, and c) *objective function* derived from the performance projection model.

Finally, the output of the optimization algorithm, i.e., best promising fusions, was used as a basis for manually transforming the code such that the original kernels are replaced by new kernels.

3. AUTOMATED END-TO-END TRANSFORMATION

This section discusses the design and functions of different stages of the transformation. Figure 1 illustrates the end-to-end approach for multi-kernel transformation. Note that all of the steps of the transformation were done manually in the previous work with the exception of the genetic algorithm (enclosed in a red dotted rectangle in the figure in comparison to the entire end-to-end transformation that we automate in this paper).

First, metadata about the individual kernels, inter-kernel dependencies, and the target device is extracted using different methods. Next, the source code and metadata are used to construct and optimize the DDG and OEG. The metadata and graphs are then used to provide the input and constraints to the optimization algorithm. The optimization algorithm, a customized grouped genetic algorithm, identifies optimum kernel fusions that can enhance the performance. Next, new DDG and OEG are generated to reflect the dependencies and precedencies of the new kernels. Finally, a new set of kernels is created to replace the original kernels and the host side code is changed to invoke the new kernels. Note that kernel fission is embedded in the optimization algorithm. A detailed discussion of the motivation of kernel fission and how it works is discussed later in Section 4.1. Note also that tuning the thread block size, introduced later in Section 4.2, is embedded in the last step (i.e. when generating the code for the new kernels).

3.1 Design Considerations

3.1.1 Programmer's Intervention and Interaction

Real-world applications have different forms and levels of complexity regarding data localities (complexities are defined primarily by the algorithm and implementation). Pure automated approaches, i.e., compilation and transformation frameworks, can generally work well on most of the applications but do not give optimal performance as finding an optimal strategy for a given application is NP-hard [2]. From that perspective, we designed the framework to be fully automated, in the sense of automating every single stage of the transformation. Yet the programmer has the option to intervene at any of the steps to cater for specific features of his application without being obscured by the otherwise transparent framework.

3.1.2 Amenability for Use in Compilation Frameworks

The proposed framework is intended to be used as a standalone source-to-source transformer. However, it is of worth to use a set of constructs that can extend an existing compilation framework. Several programming standards support GPU, i.e., OpenACC, OpenMP, and OpenCL. Other high-level frameworks support specific types of computations, such as stencil, implicitly [15, 14]. For the proposed kernel fission/fusion to be an extended functionality in OpenACC for example, clauses can be added to the OpenACC's *parallel* construct to mark the kernels and data arrays that are target for fusion. Such a concept is aligned with how the framework is implemented as will be seen in Section 5. Granted that using a compilation framework will limit the programmer's intervention as explained in the previous section, yet automated transformation can still be beneficial to a large extent as will be shown in Section 6.2.2.

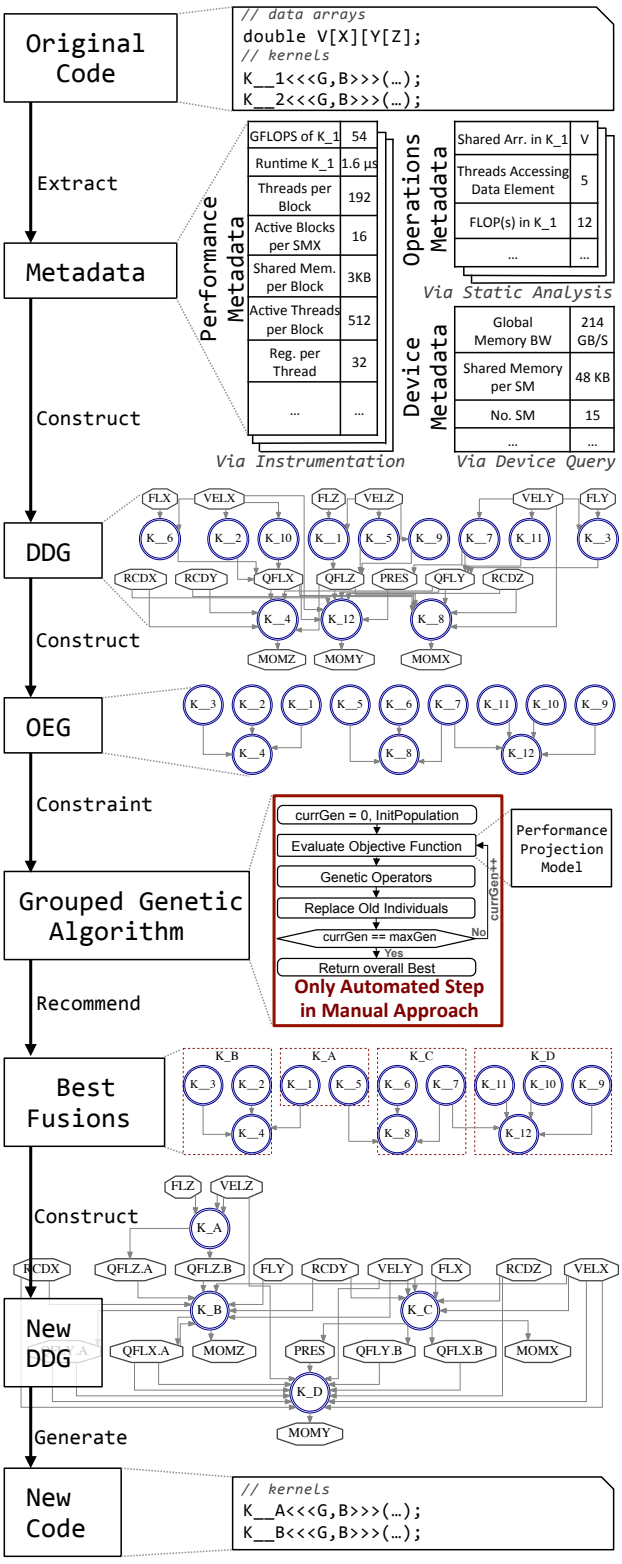

Figure 1: Workflow of end-to-end transformation including an example extracted from SCALE-LES [29] (workflow steps are on the left-hand side with an example for each step on the right-hand side). Metadata is extracted from source code using different methods. Source code and metadata are used to generate DDG and OEG. An optimization algorithm finds the optimal fission/fusions, which act as a guide for generating new kernels

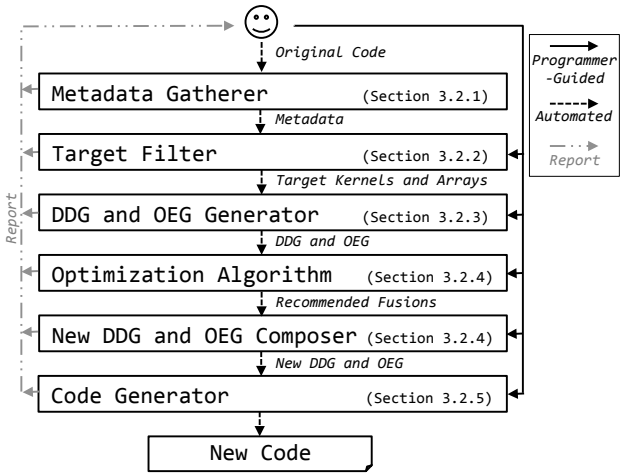

Figure 2: The Programmer can guide the transformation by amending the output of any transformation stage before using it as input to following stage. The programmer can favor an automated transformation, if intervention deemed unnecessary

3.2 Transformation Workflow

The programmer can choose to use the framework to apply the entire transformation without interfering with any of the stages (we call it *automated transformation*). Otherwise, in a *programmer-guided transformation*, the programmer can administer the workflow of stages and at each step he can choose to intervene or skip intervention. The programmer can execute up to a certain stage or starting from a certain stage by passing command-line arguments when executing the framework.

Figure 2 shows how the programmer can apply the transformation with the option of intervening at any step. The workflow is designed to allow the programmer to intervene at the pivotal points of the transformation: after data is collected about the program, after the kernels and data arrays are filtered to choose the targets of fusion, after the DDG and OEG are generated from the code, before/after the search, and after the new graphs are created.

3.2.1 Metadata

In its first stage, the framework will output the metadata in three text files. These are the files that the programmer can amend and pass to the next stage if he wishes to intervene. First, a file containing performance metadata that quantify performance metrics and device utilization for each original kernel, including: FLOPS, runtime, effective memory throughput, shared memory per thread block, registers per thread, the number of active threads, and active blocks per Streaming Multiprocessor (SM). Second, a file containing operations metadata that describe the features of the stencil operation(s) in each original kernel, such as: the stencil shape, access stride, loop size, arrays that are shared with other kernels, and FLOP(s) related to each data array. Finally, a file containing device metadata that include information about the target device required for use by the objective function, such as the number of SMs and available shared memory.

3.2.2 Identifying Targets

In an ideal scenario, all kernels in the application, by default, would be set as targets for optimization. However, passing all kernels as targets to the optimization algorithm would exponentially increase the size of the solution space, which can cause the optimization algorithm to converge significantly slower. Therefore, filtering out irrelevant kernels before passing the input to the optimization algorithm is necessary.

Algorithm 1 Heuristic for Generating DDG from CUDA Program

INPUT:
$K \leftarrow$ kernel invocations (extracted by scanning host code)
$D \leftarrow$ all data arrays residing in global memory
OUTPUT:
$DDG \leftarrow$ graph representing dependencies

```
1: BEGIN
2: for every kernel invocation K_i, i = 1, ..., N do
3:    kernelNode ← DDG.addNode(K_i, Tag = i)
4:    for each data array D_j touched in the K_i, j = 1, ..., M do
5:       if D_j has no node in DDG then
6:          dataNode ← DDG.addNode(D_j)
7:       else
8:          if firstInvocation(K_i) then
9:             dataNode ← DDG.getNode(D_j)
10:         else
11:            D'_j ← replicate(D_j)
12:            dataNode ← DDG.addNode(D'_j)
13:         end if
14:      end if
15:      DDG.addEdge(kernelNode, dataNode)
16:   end for
17: end for
```

In this stage of transformation, the code is analyzed to decide on the kernels to target for fission/fusion. The framework automatically excludes two types of kernels. First are the compute-bound kernels. Compute-bound kernels do not benefit from kernel fusion. Thus, they have no effect on the optimal fusions recommended by the optimization algorithm yet they increase the search space size. Compute-bound kernels are identified by mapping the operational intensity (FLOPS to bytes ratio) to the Roofline model [32]. Note that the information required for calculating the operational intensity exists in the operational metadata.

The second type of kernels that the framework excludes are memory-bound kernels that operate on a small subset of the data arrays target of locality, e.g., kernels applying boundary conditions on a few 2D planes of a 3D grid. Those kernels that apply boundary conditions do not have much benefit from fusion: the spatial locality is limited to a few 2D planes only. Moreover, the objective function evaluation time would significantly increase due to the complex analysis required. Finally and more importantly, leaving those kernels in the search space can lead to missed positives, i.e., they mask alternative fusions that could have exposed better locality.

The boundary kernels in stencil applications vary from doing the same operation as in non-boundary kernels up to kernels applying complex boundary conditions. Using the metadata, the boundary kernels are identified by the framework as kernels operating on a small number of iterations on a subset(s) of the data array(s).

3.2.3 DDG and OEG

As shown in Algorithm 1, the DDG is constructed by adding a node for every kernel invocation and every data array target of locality. Edges are added to express the intention of data arrays; an edge connects an input data array to the kernel using it and an edge connects a kernel to the output data array. The DDG is generated when doing the static analysis required for metadata gathering. The OEG is generated from the DDG by adding nodes to the OEG equivalent to all kernel nodes in DDG and then adding edges to express the precedency as dictated by the control flow or host side operations, i.e., copying data between device and host.

In addition to creating the graphs from the original code, the framework applies optimizations on the graphs that could require changing the original code. The following is an example for a case in which the graphs are optimized. Assume a kernel A touches arrays X and Y for reading and writing, respectively. Another kernel B touches X and Y but for writing and reading, respectively. This case, which actually occurs in one of the applications evaluated, creates a cycle in the DDG, causing incorrect constraints being passed to the optimization algorithm. The problem is resolved by the OEG generating heuristic that adds precedency for the kernel invoked first by the host. Another optimization is to add redundant instances for arrays having several kernels writing into them to relax dependencies (elaboration on this optimization in previous work [28]). After this stage in the transformation, a report is given to the programmer regarding changes that were done to the original code to optimize the two graphs. The programmer can intervene and change the DDG if he finds it improperly representing the actual dependencies in his program.

3.2.4 *Identifying the Best Kernel Fissions/Fusions*

At this step of the workflow, the input is the metadata, DDG, and OEG. In addition to that, a parameter input file for the optimization algorithm is required. The parameter file configures the population, genetic operators, generations, and constraints [27]. There is a default parameter file provided for the programmer. The default values were chosen based on empirical tests. One more important input to the optimization algorithm is the objective function. The previous work derived an objective function based on the projected performance bound. However, other objective functions, i.e., performance models, can be used as an objective function. The programmer can add his own objective function, written in standard C language, and recompile the framework. Next, the programmer can change an entry in the parameter file to use his objective function. Note that the objective function is a black box, i.e., it receives individual solutions as an input and returns the float value of a projected performance bound in GFLOPS.

Running the optimization algorithm results in a new DDG and OEG representing the new program with the recommended fusions and fissions. Note that the new OEG is different from the original OEG in that the fissions and fusions can be visualized in the DOT file (see red dotted lines in new OEG in Figure 1). The programmer, if not satisfied by the output of the optimization algorithm, can amend the OEG DOT file and have another run to automatically amend the new DDG. Once the programmer is satisfied, the new DDG and OEG can now be used as an input to the final transformation stage as shown in the following section.

3.2.5 *Applying Kernel Fission/Fusion*

The programmer can pass the command-line an argument to run only the code generator using as input the original code, the new DDG and OEG, and the metadata. The code generator generates new CUDA code that the programmer can compile using CUDA's *nvcc* compiler without any external dependencies. The programmer can also amend the new CUDA code with relative ease considering that the generated CUDA kernels are highly readable, with the exception of CUDA kernels that require a complex fusion as will be explained in Section 5.5. High readability is an outcome of the tool used for source code manipulation.

4. IMPROVED TRANSFORMATION AS ENABLED BY AUTOMATION

The automated approach for transformation gives an added opportunity for improving the transformation itself by introducing transformation components that would have been a prohibitive burden on the programmer's end in the manual approach. This section discusses two components in the transformation that are enabled by the automation.

Algorithm 2 Heuristic for Fission of a Kernel

INPUT:
Original kernel K to be fissed
$D \leftarrow$ set of N arrays residing in shared memory and used in K
$G \leftarrow$ empty graph representing dependencies among arrays in D
OUTPUT:
$F \leftarrow$ set of kernels resulting from fissing K

```
1:  BEGIN
2:  for every array D_i, i = 1, ..., N do
3:     arrayNode = G.addNode(D_i, Tag = i)
4:  end for
5:  for every array D_i, i = 1, ..., N do
6:     for each data array D_j, i = 1, ..., N and i ≠ j do
7:        if dependencyExists(D_i, D_j) then
8:           G.addEdge(i, j)
9:        end if
10:    end for
11: end for
12: while G is Not Empty do
13:    Root ← random node Root from G
14:    SubG ← subgraph returned by breadth first_search(Root)
15:    F_i ← kernel fissed from K by extracting all instructions related to
       arrays in SubG_k
16:    for every node in SubG_i, i = 1, ..., |SubG| do
17:       delNode(SubG_i)
18:    end for
19: end while
```

4.1 Kernel Fission

Kernel fission in this paper refers to splitting a kernel into several kernels such that each data array and the operations acting on it for all iterations are kept together and not split among kernels. So for a set D of n arrays residing in shared memory and used in original kernel K, find m kernel subsets $K_1, \ldots, K_m \subseteq K$ such that a) the subsets are pairwise disjoint and complete (i.e., their union via code aggregation is exactly K), b) if $x_{ij} = 1$ when array D_i appears in kernel subset K_j, $x_{ij} = 0$ otherwise, $\sum_{j=1}^{m} x_{ij} = 1, \forall i \in \{1, \ldots, n\}$, and c) all operations related to array D_i only exist in kernel K_j at which D_i appears. D_i is in this case called a *separable data array*. Figure 3 shows a simple example of kernel fission.

Algorithm 2 shows the heuristic for splitting a kernel into many kernels by fission. Initially, a graph representing the dependencies among the data arrays using shared memory is constructed. An array A is not dependent on array B if all instructions touching elements in B have no effect on the value of any element in A and vice versa, i.e., altering values of one array has no side effect on the values of the other arrays. The dependency between two arrays is determined based on a scalable method for polyhedral analysis at the granularity of individual statements [16]. Further details of the analysis of the dependency are not included due to space limitation. If separable data arrays exist, the graph will be a disconnected graph, i.e., there exists pairs of nodes having no path between them. Next, a random node is picked and a breadth first search is conducted to find all nodes connected to it. The nodes are then removed from the graph and the process repeats with a new random node until there are no more nodes left in the graph. The enumerated disconnected subgraphs guide the auto generation of the new kernels fissed from the original kernel by confining the instructions reading/writing to the arrays in each disconnected subgraph. Note

that if no separable arrays exist, a single subgraph identical to the original graph will be returned by the breadth first search.

While kernel fission intuitively seems to reduce the chances for data locality, we argue that fission can be a useful tool in searching for optimal data locality. This is based on two observations: a) many applications have large sized CPU kernels and when ported to GPU the device kernels are consequently of large size (i.e., include many data arrays), and b) the main constraint affecting fusion is the capacity of shared memory. Therefore, in the evolving search for solutions in the genetic algorithm, candidate solutions, i.e., proposed kernel fusions, can often be stuck on the boundary of the feasible area of the search space. This means attempting to fuse any more kernels to a candidate fusion on the boundary of the search space will make the solution infeasible and be penalized by the penalty function. We relax the boundaries of the search space by applying fission to the kernels that hit the shared memory capacity constraint such that breaking up the kernel can provide shared memory space to explore new fusions. Kernel fission can also have significant impact of performance for applications that already have fused-like kernels. In such applications kernel fission is not only a strategy for relaxing the search space but can be the main force driving performance improvement in comparison to kernel fusion. The results section shows two applications of this type.

There are two prospective methods for including kernel fission in the transformation. The first prospective method is to apply an initial round of iterative fission before running the optimization algorithm such that all the kernels provided as input to the optimization algorithm can not be fissed any more, i.e., none of the kernels have any separable data arrays. This method is impractical, as it will cause an explosive expansion in the search space size, i.e., search space size is exponentially proportional to the number of kernels. The other prospective method for fission is to start with the original kernels without a fission pre-step, and apply fission on-demand for candidate solutions if required. This method is not possible because the performance projection method used as an objective function depends on empirically measured metadata, i.e., kernels fissed on-demand have no such metadata.

We introduce a compromising solution, *lazy fission*, at which fission is applied in a pre-step in which the metadata of the fissed kernels is gathered. Next, the optimization algorithm starts with the original kernels and not with the fissed ones. When the genetic algorithm is evolving, and upon investigating solutions on the search space boundary, fission is applied to relax the boundary. Therefore, the metadata gathered in the pre-step enables the use of the performance projection while avoiding the increase in search space size had the fissed kernels been used at the start of the search process. The decision to apply fission, when needed, is embedded in a dynamic penalty function. Hence the kernels that would benefit from fission would be less penalized, i.e., relaxed:

$$f_p(x) = f(x) + \sum_{i=1}^{m} C_i \delta_i - C_{SM}\delta_{SM}$$

$$\text{Where } \begin{cases} \delta_i \;\; = 1, \text{constrain } i \text{ is violated} \\ \delta_i \;\; = 0, \text{constrain } i \text{ is satisfied} \\ \delta_{SM} = 1, \text{shared memory capacity constrain is violated} \\ \delta_{SM} = 0, \text{shared memory capacity constrain is satisfied} \end{cases}$$

$f_p(x)$ is the penalized objective function, $f(x)$ is the unpenalized objective function, C_i is a constant imposed for the violation of constrain i, and C_{SM} is a positive variable corresponding to the anticipated relaxation in shared memory constraint if fission for any of the cached data arrays is possible. Otherwise C_{SM} is a negative number to penalize solutions on the boundary of the search space that cannot be fissed.

Before Fission

```
Kern_A<<<G, B>>>(R, S, T, Q, P, V, U, W, nz, c);

__global__Kern_A(R, S, T, Q, P, V, U, W, nz, c){
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]= S[i,j,k] * (V[i-1,j,k]/V[i,j,k]);
  W[i,j,k]= V[i-1,j,k] - V[i,j,k];
  P[i,j,k]= c*(Q[i-1,j,k]+Q[i,j-1,k]+Q[i,j,k]+Q[i+1,j,k]+Q[i,j+1,k]);
  U[i,j,k]= T[i,j,k]-Q[i,j,k]*(Q[i-1,j,k]-Q[i,j-1,k]);
 }
}
```

After Fission

```
Kern_X<<<G, B>>>(R,S,V,W,nz);
Kern_Y<<<G, B>>>(P,Q,U,T,nz,c);

__global__ Kern_X(R,S,V,W,nz){
int i= //absolute index in X-Dir
int j= //absolute index in y-Dir
for(int k=0; k<nz;k++) {
 R[i,j,k]= S[i,j,k]
        * (V[i-1,j,k]/V[i,j,k]);
 W[i,j,k]= V[i-1,j,k]-V[i,j,k]);
}  }
```

```
__global__ Kern_Y(P,Q,U,T,nz,c){
int i= //absolute index in X-Dir
int j= //absolute index in y-Dir
for(int k=0; k<nz;k++) {
 P[i,j,k]= c*(Q[i-1,j,k]+Q[i,j-1,k]
   +Q[i,j,k]+Q[i+1,j,k]+Q[i,j+1,k]);
 U[i,j,k]= T[i,j,k] - Q[i,j,k]
        * (Q[i-1,j,k]- Q[i,j-1,k]);
 }
}
```

Figure 3: Example of CUDA Kernel fission. Kernel A is fissed to kernels X and Y

4.2 Tuning Thread Block Size

A main opportunity for a simple yet effective optimization not addressed in the previous work is the tuning of thread block size to maintain high occupancy. Occupancy is the ratio of sustained active warps to maximum possible active warps. The factors affecting the occupancy are the thread block size, shared memory used by each thread block, and the registers used by each thread. Tuning thread block size can have a significant effect on performance [24]. Therefore, we operate towards the target of keeping the occupancy of the new kernel as high as possible by tuning the thread block size. We leverage the analysis done by the performance model to estimate the shared memory used per block and registers used per thread. We then enumerate all possible sizes of thread block and substitute in a series of equations using the same method as in the CUDA occupancy calculator tool. A block size giving the highest occupancy is chosen to be the new block size used in kernel invocation.

It is important to note that tuning is done in the final step of generating the new code and not included in the optimization algorithm as an additional degree of freedom. This is because occupancy is a measure of utilization and is not equivalent to performance. If occupancy becomes part of the optimization algorithm, the performance projection model would have an added inaccuracy not related to the performance.

5. IMPLEMENTATION

ROSE compiler infrastructure [21] is used for source-to-source transformation. ROSE supports manipulating the AST (Abstract Syntax Tree), and automatically transforming CUDA code to AST, and vice-versa. Initially, the AST of the CUDA program is constructed. To gather metadata, new statements are injected in the AST to instrument the code. Next, a shell script runs the instrumented program in profiling mode and the metadata is later extracted from the output. To construct the DDG and OEG, we locate the statements in the AST that would be statically analyzed. Examples of the static analysis are identifying the kernels order of invocation and identifying the shared arrays. Next, the GA is executed independently to identify the best fissions/fusions. Based on the output of the GA, the AST is modified to remove the original kernels and create the new kernels. The new kernels include

statements from the original kernels plus new statements that are injected, e.g. loading/storing to shared memory. Finally, the AST is unparsed to generate the new source code.

The following sections briefly present the implementation of the metadata gatherer, target kernel identifier, DDG and OEG constructor, the optimization algorithm, and the code generator.

5.1 Metadata Gathering

First, performance metadata is gathered using code instrumentation. The code is instrumented, using ROSE, by injecting CUDA APIs dedicated for creating, destroying, recording, and measuring elapsed time of events such as kernel invocations. A shell script runs CUDA command-line profiler, *nvprof*, with the appropriate parameters to generate the metadata of the program. The required values are then automatically extracted from the profiler's output file. Note that only a single run of the instrumented code, on the target device, is required to gather the metadata.

Second, operations metadata is gathered using static analysis. The metadata is gathered by a static analysis of the program. To be more specific, the subset AST corresponding to each kernel is examined to identify the stencil operations, intra-kernel dependencies, and the data access pattern, i.e., shape of stencil, loop(s) size, and access stride. There are restrictions on the stencil operations in kernels. The limitations section will later discuss those restrictions.

Third, device metadata is gathered using a device query program. A small program queries the device for the required information; the program is named *deviceQuery* and is readily available with CUDA SDK samples. Note that device query can be done using the instrumentation being done for gathering operations metadata. However, since the device query is required only once per target device, we choose to leave it as a separate step.

5.2 Identifying Targets

To identify compute-bound kernels, operational intensity for all kernels in the program is calculated using the metadata. A precedency constraint can occur between two memory-bound kernels due to an intermediate invocation of a compute-bound kernel, hence the compute-bound kernels are added to the DDG and OEG. However, they are tagged as ineligible for fusion and are not used when randomly making an initial population in the optimization algorithm. Similar to the compute-bound kernels, boundary kernels are added to the DDG and OEG but not used when searching for fusions. Boundary kernels are identified as kernels having a small number of iterations applying stencil operations.

5.3 Constructing DDG and OEG

The following are functions related to the DDG and OEG: generation from parsed code, parsing them to guide generation of new code, and programmers amending them. The AST is used to generate and parse the graphs. An output file is generated in DOT format where the programmer can visualize the graphs using various tools such as GraphViz [1] (DDG and OEG in Figure 1 are generated with GraphViz). If the programmer changes the DDG, another run with an appropriate command-line argument is required just to generate a new OEG from the DDG the programmer amended.

5.4 Optimization Algorithm: GGA

Kernel fusion is defined as a combinatorial optimization problem. A customized grouped genetic algorithm is used to search the space of possible fissions/fusions while catering for the multivariate nature of the problem. The algorithm is implemented to be flexible; the programmer can tune the parameters of the optimization algorithm, write his own objective function, and add constraints. Details about the implementation of the optimization algorithm are not covered by this paper and can be found elsewhere [27].

5.5 Code Generation

As mentioned earlier, ROSE parses CUDA code into an AST at which transformations can be applied before unparsing the AST to generate the new code. Fission is applied iteratively as long as there is at least one separable data array in the target kernel(s). The following sections introduce the code generation process when fusing kernels, i.e., different cases for generating a new kernel.

5.5.1 No Fusion

This is the case when the framework decides that a kernel is not to be fused, i.e., fusing of the kernel is not expected to improve performance. The new kernel is a copy of the original kernel.

5.5.2 Simple Fusion

This is the case when the framework reaches a decision that two or more original kernels are to be fused together into one new kernel. Note that in simple fusion, unlike complex fusion, all of the original kernels to be fused together have no precedency constraints between them appearing in the OEG. This means that no operation in any kernel relies on data generated from operations in the other kernel(s). First, the codes of the original kernels are aggregated in a new kernel. Next, the code is changed such that the arrays that are target of locality are stored into shared memory before the relevant stencil operation(s). The statements operating on the arrays are adjusted accordingly to read from shared memory. Next, the code segments are aligned to the same loop boundaries by offsetting the indices of every element accessed. Finally, for code segments of fused kernels requiring iterations less than the largest of loop boundaries, conditional statements are added.

5.5.3 Complex Fusion

This is the case when the original kernels have at least one precedence constraint appearing among them in the OEG. This causes the fusion process to be complex because the operations appearing the new kernel must follow some order. First, the steps of the previous simple fusion steps are applied. On top of that, appropriate barrier(s) are used to prevent data races. In addition to the barrier(s), this type of fusion introduces a type of problem related to the architecture of GPUs: the scope of shared memory in GPU is a thread block, and shared memory is not coherent with off-chip cache and device memory. Therefore, the boundary threads of a thread block become unaware of the change of intermediate values held in the shared memory of other thread blocks. We resolve this problem using the common approach of temporal blocking in GPU stencil applications [8]. An extra layer(s) of values is initially loaded to the shared memory of each block. The number of layers depends on the radius of the stencil neighborhood. Hence the stencil operations before the barrier are applied to an extra layer(s) of stencil sites at each thread block. After the barrier, the stencil operations are only applied to the number of stencil sites originally intended per thread block.

5.5.4 Host Code

For the host code, the code generator replaces the invocations of the original kernels with those of the new kernels. The size of thread blocks for each invocation is determined according to the tuning method mentioned earlier. The order of invocation is extracted from the new OEG.

Table 1: Applications Attributes and the Effect of Automated Transformation

Application	No. Original Kernels	No. New Kernels	No. Fused Kernels	No. Kernels Output of Fusion	Avg. Fissions/-Generation	No. Arrays Targeted	No. Array Sharing sets
SCALE-LES	142	63	117	38	0.07	64	103
HOMME	43	30	22	9	0.163	27	51
Fluam	169	144	42	17	0.01	84	201
MITgcm	37	29	14	6	0.010	35	61
AWP-ODC-GPU	12	24	6	3	1.062	21	48
B-CALM	23	24	8	3	0.783	44	58

6. EVALUATION

This section does not demonstrate the effectiveness and limitations of the GPU kernel fusion introduced in the previous work (demonstrated in [28]). The aim of this section is to: a) demonstrate the effectiveness of the automated end-to-end approach in speeding up real-world applications and evaluating the effect of kernel fission and tuning thread block size (Section 6.2.1), and b) quantify the efficiency of the automated transformation, by comparison to manual kernel fusion, and highlighting the experience when using a programmer-guided transformation (Section 6.2.2).

6.1 Methodology

6.1.1 Applications

SCALE-LES [29]: a next generation weather model having over a hundred kernels and tens of data arrays. Most of the kernels, in the dynamical core, are memory-bound kernels applying iterative stencils. SCALE-LES is liberally estimated to have 41% of reducible off-chip traffic from which 26% was achieved [28] (1.35x speedup). The problem size used is 1280x32x32.

HOMME [4]: the dynamical core within the Community Atmospheric Model (CAM). Similar to SCALE-LES, most kernels are memory-bound yet not all modules of HOMME are ported to GPU so we report the speedup of the kernels and not the entire application. The problem size used is 4x26x101.

Fluam [25]: a fluctuating particle hydrodynamics application based on an hybrid Eulerian-Lagrangian approach. Fluam relies on finite volume stencils with explicit time stepping formulated as 3^{rd} order Runge-Kutta scheme. Fluam includes a large number of kernels, 207, of which 169 are stencil-based. The problem size used is 1000x8000x8000x384000.

MITgcm [12]: is an oceanic general circulation model relying on a finite volume numerical method with partial cells. The setting used for experimentation [20] is for a non-hydrostatic flow mode, which makes the simulation's hotspot a set of kernels with simple stencil operations: a 3D conjugate gradient solver for surface pressure. The problem size used is $804\text{x}2064\text{x}(24.9\text{x}10^6)\text{x}25\text{m}$.

AWP-ODC-GPU [9]: an earthquake wave propagation simulator. 3D velocity-stress wave equations are solved by explicit staggered-grid 4^{th} order finite difference. AWP-ODC-GPU includes vector and stress computation kernels using 3D arrays each having the same size as the 3D simulation domain. It is worth mentioning that AWP-ODC-GPU is highly optimized for GPU architecture features; overlapping computation with communication highly influenced the design of the kernels and ghost cell regions. The problem size used is 3500x2500x1500.

B-CALM [30]: a 3D-FDTD simulator which models the permittivity of dispersive material, i.e., propagation of electromagnetic waves in metallic nanostructures. B-CALM breaks down the update equations for both the electric and magnetic fields into separate kernels to optimize for minimizing thread divergence in simulating materials with high number of poles. The approach of breaking the kernels in such way adds extra memory traffic to hold intermediate results in global memory in-between kernel invocations. Hence a setting of high resolution, i.e., large problem size, is used in this evaluation to amplify the effect of the extra memory traffic. The problem size is 208x208x400 cells with $0.1nm$ resolution.

6.1.2 Platform and Tools

The results were collected using CUDA 6.0 for Nvidia's Kepler K20X and K40 GPUs. Two of the applications, SCALE-LES and CAM-HOMME, were written originally in CUDA Fortran. For experimentation purposes, the kernels were manually translated to CUDA C while the host related computation remains in Fortran 90. The non-hydrostatic flow mode setting used for MITgcm in experimentation was also translated to CUDA C. Performance results are averaged for 10 runs compiled at the highest possible compiler optimization. Output of the automated method was verified against the output of the original code base for every single run. All the results reported are for double precision in 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 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. Initialization and output writing parts are excluded from this calculation. The I/O time is negligible when time iterations of tens to hundreds of thousands of time steps are involved.

The framework is implemented in C++, parallelized by OpenMP, and executed using an Intel Xeon E5-2670 2.60GHz machine (8 cores). The optimization algorithm runs for 500 generations, the population size is 100 individuals. Evaluation of the objective function constituted more than 90% of the total runtime of the optimization algorithm for all applications (total runtime for an automated transformation averages around 11 minutes).

6.2 Results

We start this section by summarizing the important results. The transformations applied on the six applications resulted in speedups ranging between 1.12x to 1.76x compared to the original CUDA codebases. Kernel fission appeared to have the primary influence on performance improvement for applications that have original kernels that are already fused to some extent. Tuning thread block size proved to be effective for most applications. The automated transformation of framework is shown to enhance performance within high proximity to performance enhancement due to manual fusions. Programmer-guided transformation proved to be of value in identifying sub-optimal transformations that were the reason of difference in performance improvement between the manual and automated methods.

Section 6.2.1 quantifies the effect of automated transformation on performance and investigates the impact of applying kernel fission and thread block tuning. Section 6.2.2 analyzes the efficiency of the automated transformation in comparison to the manual transformation and discusses the effectiveness of programmer-guided transformation.

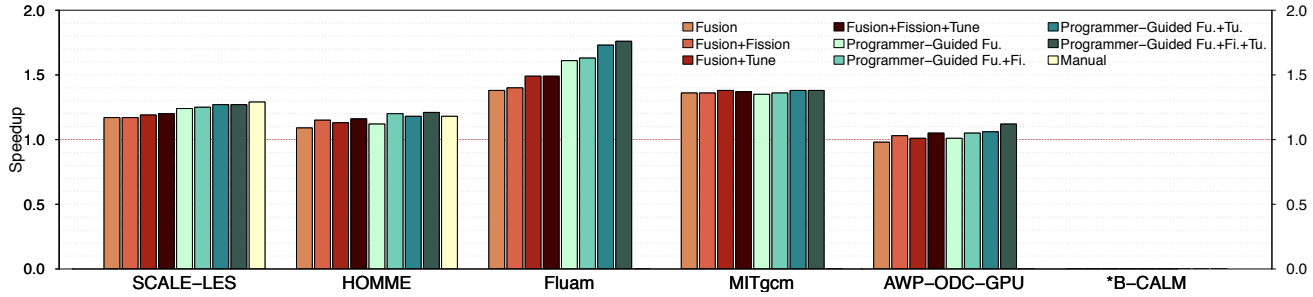

Figure 4: K20X speedup compared to baseline CUDA version with no kernel fission/fusion (*B-CALM run not possible due to limited device memory capacity)

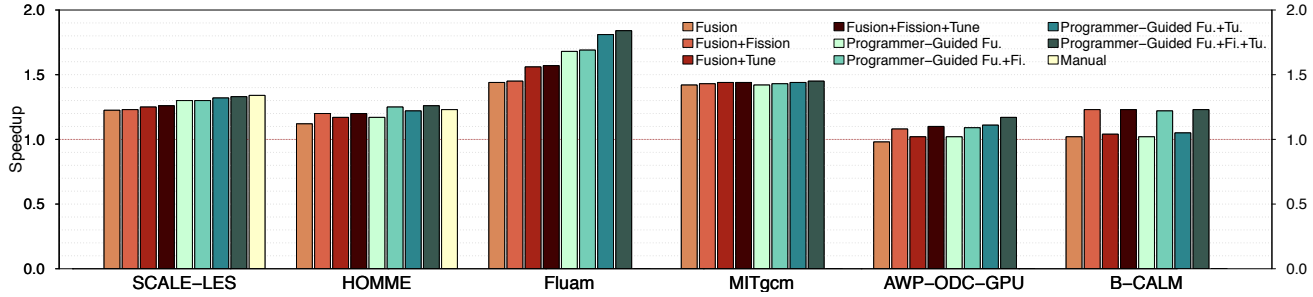

Figure 5: K40 speedup compared to baseline CUDA version with no kernel fission/fusion

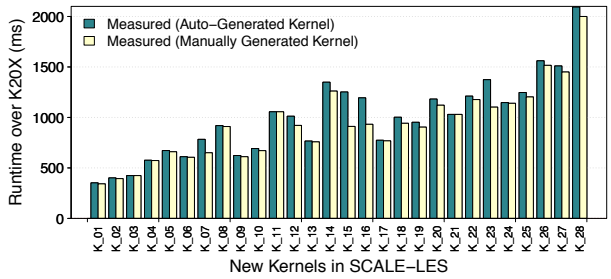

Figure 6: Runtime on K20X for new kernels of SCALE-LES. Manually generated vs. auto-generated kernels

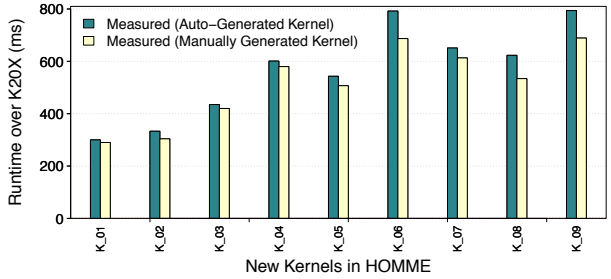

Figure 7: Runtime on K20X for new kernels of HOMME. Manually generated vs. auto-generated kernels

6.2.1 *Performance and Speedup*

Table 1 summarizes the collective effect of transformation on kernels. The two factors affecting the size of the optimization problem are the number of original kernels and the number of array sharing sets (array sharing sets is an enumeration of possible reuse combinations). Note that the number of original kernels can potentially increase during the optimization process if fissions are feasible. As a consequence, for two applications, AWP-ODC-GPU and B-CALM, the number of new kernels is more than the number of original kernels. This means that the actual number of kernels available for fusion, after applying fission, was higher than the number of original kernels (27 rather than 12 for AWP-ODC-GPU and 29 rather than 23 for B-CALM). Analyzing the reason for this effect on those two applications revealed that the original kernels of the two applications were large in code size compared to the other applications. This implies the kernels of both applications are already in an almost-fused state. It also indicates a higher possibility of finding separable data arrays that can be fissed. The other applications, however, have tight kernels, which decreases the opportunity for fission. The difference is demonstrated in Table 1 where the average number of fissions per generation of the optimization algorithm is orders of magnitude higher in the two applications. It is worth mentioning that the higher tendency towards fission rather than fusion in these two applications was made possible by the performance model and optimization algorithm. Had a naive fission been applied, best fissions, which are the fissions that would relax the penalty for desired fusions, would have not been guaranteed.

Figures 4 and 5 show the speedups achieved when using the framework in both automated and programmer-guided modes. Note that speedup for the ”Manual” kernel fusion is included for only SCALE-LES and HOMME, while the other four applications were transformed only automatically with no reference implementation for manual fusion existing. Several points are important to note from the figures. First, kernel fusion showed no speedups for AWP-ODC-GPU and B-CALM. Surprisingly, kernel fission+fusion results in significant improvement in performance for those applications. Second, thread block tuning shows worthwhile improvement in performance. Finally, programmer-guided transformation shows significant improvement over the automated transformation for SCALE-LES, HOMME, and Fluam. The following section discusses the role of programmer-guided transformation.

Table 2 shows the effect of thread block tuning on the occupancy of the new kernels (1.0 is top occupancy). The occupancy reported in the table is the occupancy measured after kernel fusion and not the estimated occupancy that guided the tuning.

Table 2: Tuning Thread Block Size for New Kernels

Application	No. Kernels Output of Fusion	No. Tuned Kernels	Avg. Occupancy Before Tuning	Avg. Occupancy After Tuning
SCALE-LES	38	14	0.65	0.80
HOMME	9	4	0.55	0.85
Fluam	17	11	0.81	0.90
MITgcm	6	3	0.95	0.96
AWP-ODC-GPU	3	2	0.75	0.77
B-CALM	3	0	0.72	0.72

6.2.2 *Efficiency of Automated Transformation*

This section discusses two issues. The first issue is the source of inefficiencies in automated transformation, i.e., the reason for the difference in performance between the automated vs. manual transformation. This was done by analyzing the reports of each transformation stage to understand the behavior of the framework components, which in turn helped in identifying the reason in performance gap between the manual and automated kernel fusion.

The second issue discussed is how the programmer-guided transformation was used in this context to close the performance gap; after identifying the inefficiencies of the automated transformation, manual intervention was minimally introduced to improve the performance. As will be shown, in the case of SCALE-LES and HOMME, the manual changes were to address problems in the auto-generation of code. In the case of Fluam, the manual changes were to address a problem in filtering target kernels.

There are four possible reasons why the automated transformation can result in lower performance than the manual transformation are: a) inaccuracy in gathering metadata, b) inaccuracy in generation of the DDG and OEG, c) failing to generate new kernels that reuse data optimally, and d) failing to filter the target kernels automatically. We used the programmer-guided transformation to identify which of the four reasons was the source of performance difference for each application:

SCALE-LES: When compared against the speedup of the previously reported manual method [28], baseline automated transformation for SCALE-LES achieves 85% of manual method compared to 92% for the programmer-guided transformation (Figures 4 and 5). Thread block tuning showed improvement in performance for both automated and programmer-guided transformation, while fission did not show a significant difference.

We identified the performance gap to be caused by the auto-generated kernels, i.e., auto-generated kernels have lower aggregate performance compared to the manually generated kernels. Accordingly, we analyzed the auto-generated kernels that show the highest difference in runtime from the manually generated kernels. Figure 6 shows the runtime for the new kernels of SCALE-LES executed on K20X. Note that few kernels are contributing the highest in the total performance difference: kernels K_07, K_15, K_16, and K_23. The problem in the generated code of the mentioned kernels was the inefficiency in fusing kernels that had deep nested loops inside them. Normally, kernels would have a single loop that iterates on the vertical 3D space. However, some kernels can have nested loops where stencil operations would be applied to arrays with more than three dimensions. The programmer that manually fused the kernels aggregated the two codes such that they share the most inner loop if codes of both kernels do not share the external loops. The auto-generated code on the other hand did not aggregate the codes in same manner. Therefore iterations for the code of the first kernel were entirely finished before the iterations of the second kernel started and shared data was never reused. Automatically resolving this problem by running a polyhedral model is a main point in the future work.

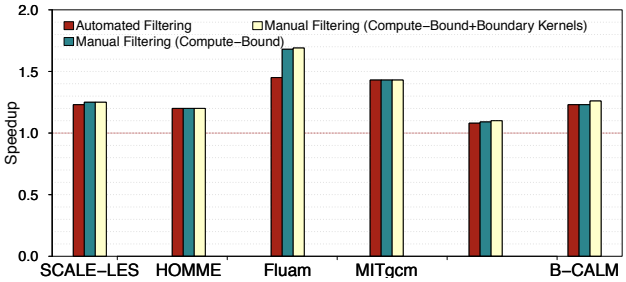

Figure 8: K40 speedup for automated vs. manual kernel filtering compared to baseline CUDA version with no kernel fission/fusion

HOMME: Like SCALE-LES, HOMME's performance gap was caused by the auto-generated kernels. However, the exact cause of the difference in performance in HOMME was found to be of different nature. The runtime comparison in Figure 7 shows a more even distribution of difference in performance with two kernels slightly higher than the others. Analysis showed that the reason of performance degradation was auto-generating code that incurs intra-warp divergent branches, which does not appear in manually fused kernels. In the proposed method for fusion, the code segments originating from different kernels are dimension-aligned by offsetting the indices of every element accessed. Conditional statements are added to account for the difference in dimension lengths. When loops bounds are different on both ends of the warp dimension, i.e., *tx*, the manual code accumulated the diverging threads in the last thread block on the warp dimension. The automated version, however, adds branches for the first and last thread blocks, which increases the intra-warp divergence. Optimizing the generation of branching statements to close the performance gap is part of the future work. Finally, it is worth mentioning that due to the significant effect of fission on performance for HOMME, the programmer-guided transformation with fission exceeds the manual approach in speedup.

Fluam: In Fluam, we observed that the optimization algorithm converged poorly in comparison to the other applications. This implies the excessive size of the solution space, i.e., original kernels that unnecessarily appear as targets. To understand this behavior, we evaluated the effectiveness of filtering the kernels by comparing the automated kernel filtration of the framework with a version that was filtered manually. Similar to the automated method, the manual filtration filters out the compute-bound kernels and boundary memory-bound kernels. Figure 8 shows a comparison in speedups generated by both the automated and manual filtering. All applications, excluding Fluam, give the same speedup when comparing the automated with manual versions. This implies the effectiveness of the framework in automatically filtering out kernels that are not worthy of being targets. The transformation of Fluam was further investigated and the anomaly is likely to be a number of kernels that have latency problems (i.e., poor computation and memory overlapping). Those kernels falsely appeared to the framework as memory-bound kernels and populated the search space. Which in turn caused less convergence in the optimization algorithm running to a pre-defined number of steps. Finally, it is worth mentioning that when no filtering out at all was applied, automated or manual, the optimization algorithm converged 2.5x slower on average.

7. LIMITATIONS

Despite demonstrating promising results for real-world applications, the proposed framework is certainly not complete. Limitations and restrictions that the programmer must be aware of are summarized in the following points:

Table 3: Related Work Summary

Domain	Technique	Relationship/*What is lacking*
GPU Kernel Fission and Fusion	GROPHECY [17]	Analytical model based on code skeletons. *Writing code skeletons is a significant effort and requires the kernel code to already exist.*
	Fuse kernels applying basic operations [11, 10]	Using building blocks, e.g. map-reduce operations, simplifies the search process at the expense of bounding the kernels to a fixed set of problem-specific elementary functions.
	Fuse kernels to optimize for power [31]	The problem is represented as a dynamic programming problem of fusing a small number of kernels. Reported fusion of two kernels was shown effective for power optimization. *Effective for demand balancing of hardware resources and not locality.*
CUDA Code Generation	PPCG [26], PLUTO [3]	Tiling strategy and a code generation scheme for the parallelization and locality optimization of imperfectly nested loops. *Targets fine-grain locality, i.e., nested loops and polyhedral iterations. Not for coarse-grain locality spanning entire programs.*
	Stencil DSL generating CUDA [14, 15]	Generate CUDA code implicitly, via directives or leveraging C++ templates. Optimizations, such as computation-communication overlapping, are applied. *No optimization available for exploiting inter-kernel locality.*
Coarse-grained Data Locality	Pipeline parallelism of stream programs [23]	Changing the boundaries of the pipeline stages by fission/fusion to optimize for load balance on multicores. *Assumes load balance to be the objective of optimization with locality implicitly optimized. Works for cache systems having coherencey*
	Task clustering in data-intensive scientific workflows [22, 6]	Task clustering is aggregating multiple workflow tasks into a single job to reduce the job execution overhead. Locality in workflows is focused on assigning tasks to where the data resides and not merging tasks using the same data. *Runtime and dependency imbalances [7], if not considered in task clustering, could degrade the performance.*

Pointer aliasing: Pointer aliasing is extremely complicated in static analysis. Hence we limit the programmer to restrict the pointers for data arrays that are targets for locality. The following improvement is under consideration for the future work. For the majority of stencil applications, the data operated-on across the kernels is stable throughout the lifetime of the program. By stable we mean that the memory dependences between different kernels is predictable. A pre-run of the program for a few iteration to detect the data usage pattern can provide a more practical method to address the pointer aliasing problem (this approach was originally proposed in pipelined parallelism for stream programs [23]).

Supported stencils: Our framework specifically targets stencils for dense multidimensional Cartesian grids, e.g., 3D atmospheric grid. For example, stencils for unstructured grids are not possible. One approach is to extend the framework to introduce support for different stencil operations. Another radical approach is to have the proposed framework sit on top of a framework that can support different stencils via intrinsic data models, e.g., Chapel programming language [5]. The proposed framework would then extend the runtime and compiler to leverage the data models.

Data access: As mentioned earlier, the static analysis can detect the access stride. However, the code generator will assume that the stencil is following the common horizontal mapping to CUDA grid and iterating in the vertical direction [19] and will not be effective in translating stencils with irregular access. Furthermore, stencils with large neighborhood size can limit the performance in some cases, i.e., when thread block halo layers are exceedingly large in size.

Sensitivity to input: The analysis done by the framework is dependent on the properties of the original kernels. More specifically, the performance model projects the performance of the kernels based on empirically measured properties of stencil operations. Using a different problem size with the transformed code could negatively impact the effectiveness of kernel fusion. However, we argue that this is unlikely to happen considering that, in practice, stencil applications favor design for weak scaling. Hence as the problem size grows with the number of nodes, the input size per node remains the same. Another point of concern is the change in program behavior if specific values are used as input. The supported stencils in this work, however, are unlikely to have such problem. Nonetheless, a proper analysis of the sensitivity of the performance of transformed code to change in input is important.

8. RELATED WORK

Table 3 contrasts with related work on GPU kernel fusion, CUDA code generation, and coarse-grained data locality in literature. We elaborate on a few other related works below.

Auto-tuning CUDA by traversing the space of possible implementations can be used to generate optimal code variants [13]. The work is reported to be based on a compiler framework, which makes it extendable, in concept, to more advanced optimizations such as kernel fusion. Meng at al. [18] proposed an automated optimization for selection of the optimal ghost zone size depending on the characteristics of both the architecture and the application. The optimization addresses single kernels, unlike kernel fusion, yet it demonstrated the effectiveness of complex automated memory optimizations for GPU kernels.

9. CONCLUSION

This paper introduces an automated end-to-end method for exposing and exploiting hidden localities in a class of GPU applications with stencil-based CUDA kernels. The proposed work builds on previous work, which successfully formulated the GPU kernel fusion problem and demonstrated effectiveness, yet lacked a practical automated approach. The main contribution introduced in this paper is an end-to-end framework for applying kernel fission/fusion transformations. The framework includes a set of automated components starting from analysis of the original code and ending with generation of new code. The programmer can compile the new code using the native CUDA compiler without runtime dependencies. In addition, the paper introduced optimizations for the transformation mainly enabled by the automation. One improvement is a scheme for lazy fission that enables on-demand use of fission in the search process. Another optimization is tuning of thread block size for kernels generated from fusion to achieve high occupancy.

The effectiveness of the proposed framework is demonstrated by achieving speedups ranging between 1.12x and 1.76x for six real-world applications and with speedups comparable to a previous manual kernel fusion of two of the applications. Using the programmer-guided transformation for investigation, potential ar-

eas of improvement in code generation were identified: a polyhedral model for guiding fusion in kernels having deep nested loops, and a more efficient loop alignment scheme.

Acknowledgments

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

10. REFERENCES

[1] `http://www.graphviz.org`.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding Effective Compilation Sequences. LCTES '04, pages 231–239, 2004.

[3] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. ETAPS CC '08, 2008.

[4] 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.*, 27:335–347, 2013.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312.

[6] W. Chen and E. Deelman. Integration of Workflow Partitioning and Resource Provisioning. CCGRID'12, pages 764–768, 2012.

[7] W. Chen, R. F. D. Silva, E. Deelman, and R. Sakellariou. Balanced Task Clustering in Scientific Workflows. e-SCIENCE '13, pages 188–195, 2013.

[8] 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. HipHaC '08, pages 47–54, 2008.

[9] Y. Cui, E. Poyraz, K. B. Olsen, J. Zhou, K. Withers, S. Callaghan, J. Larkin, C. Guest, D. Choi, A. Chourasia, Z. Shi, S. M. Day, P. J. Maechling, and T. H. Jordan. Physics-based Seismic Hazard Analysis on Petascale Heterogeneous Supercomputers. SC '13, pages 1–12, 2013.

[10] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA Code By Kernel Fusion: Application on BLAS. *CoRR*, 1305, 2013.

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

[12] 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.*, 6:18–28, 2004.

[13] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A Script-based Autotuning Compiler System to Generate High-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9:1–25, 2013.

[14] X. Lapillonne and O. Fuhrer. Using Compiler Directives to Port Large Scientific Applications to GPUs: An Example from Atmospheric Science. *Parallel Processing Letters*, 24(1), 2014.

[15] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. SC '11, pages 1–12, 2011.

[16] S. Mehta, P.-H. Lin, and P.-C. Yew. Revisiting Loop Fusion in the Polyhedral Framework. PPoPP '14, pages 233–246.

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

[18] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. ICS'09, pages 256–265, 2009.

[19] P. Micikevicius. 3D Finite Difference Computation on GPUs Using CUDA. In *Proceedings of 2ND Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, 2009.

[20] O. Padon. Using graphics processing unit computing to improve the performance of an oceanic general circulation model the mitgcm and implementation to dead sea circulation. Master's thesis, Ben-Gurion University, 2014.

[21] M. Schordan and D. Quinlan. A Source-To-Source Architecture for User-Defined Optimizations. In *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. 2003.

[22] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta. Workflow Task Clustering for Best Effort Systems with Pegasus. In *Proceedings of the 15th ACM Mardi Gras Conference*, MG '08, pages 1–8, 2008.

[23] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. MICRO'07, pages 356–369, 2007.

[24] Y. Torres, A. Gonzalez-Escribano, and D. Llanos. Understanding the Impact of CUDA Tuning Techniques for Fermi. HPCS'11, pages 631–639, 2011.

[25] F. B. Usabiaga, I. Pagonabarraga, and R. Delgado-Buscalioni. Inertial Coupling for Point Particle Fluctuating Hydrodynamics. *J. Comput. Phys.*, 235:701–722.

[26] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), 2013.

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

[28] M. Wahib and N. Maruyama. Scalable Kernel Fusion for Memory-Bound GPU Applications. SC'14, pages 191–202.

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

[30] P. Wahl, D.-S. Ly-Gagnon, C. Debaes, J. Van Erps, N. Vermeulen, D. Miller, and H. Thienpont. B-CALM: An Open-Source GPU-based 3D-FDTD with Multi-Pole Pispersion for Plasmonics. *Progress In Electromagnetics Research*, 138:467–478, 2013.

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

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