J Supercomput (2015) 71:3934–3957
DOI 10.1007/s11227-015-1483-z

# Optimizing CUDA code by kernel fusion: application on BLAS

**Jiří Filipovič**[1] · **Matúš Madzin**[1] · **Jan Fousek**[1] · **Luděk Matyska**[2]

Published online: 22 July 2015


**Abstract** Contemporary GPUs have significantly higher arithmetic throughput than a memory throughput. Hence, many GPU kernels are memory bound and cannot exploit arithmetic power of the GPU. Examples of memory-bound kernels are BLAS-1 (vector–vector) and BLAS-2 (matrix–vector) operations. However, when kernels share data, kernel fusion can improve memory locality by placing shared data, originally passed via off-chip global memory, into a faster, but distributed on-chip memory. In this paper, we show how kernels performing map, reduce or their nested combinations can be fused automatically by our source-to-source compiler. To demonstrate the usability of the compiler, we have implemented several BLAS-1 and BLAS-2 routines and show how the performance of their sequences can be improved by fusions. Compared with similar sequences using CUBLAS, our compiler is able to generate code that is up to 2.24× faster for the examples tested.

**Keywords** GPGPU · CUDA · BLAS · Kernel fusion · Code generation

✉ Jiří Filipovič
fila@mail.muni.cz

Matúš Madzin
gotti@mail.muni.cz

Jan Fousek
izaak@mail.muni.cz

Luděk Matyska
ludek@ics.muni.cz

[1] Faculty of Informatics, Masaryk University, Botanická 68a, 602 00 Brno, Czech Republic

[2] Institute of Computer Science, Masaryk University, Botanická 68a, 602 00 Brno, Czech Republic

## 1 Introduction and motivation

Today's accelerators, such as CUDA GPUs, are able to perform tens of arithmetic operations while a single word is transferred from or to global memory. Moreover, the dominance of arithmetic power over memory bandwidth grows with every new hardware generation.[1] The input and output of each GPU kernel (i.e., the subprogram executed on GPU) has to be stored in global memory. Thus, many kernels with low flop-to-word ratio are memory bound. However, when some data are shared across multiple kernels, performance can be improved by fusing these kernels into a larger one and placing shared data in a significantly faster on-chip memory.

It is difficult to fuse generic kernels automatically, but automation of fusion becomes possible when the type of operations performed by kernels is limited. In this paper, we study automatic fusions of kernels performing *map*, *reduce* or their nested combination. Although the idea of fusing GPU kernels performing map and reduce has been presented in several papers [3,14,19,21], first-order function is always serial in these cases. The important feature of our approach is that the first-order function executed by map or reduce can be parallel. Consequently, our approach allows user code to operate on matrix tiles or subvectors instead of matrix or vector scalar elements. It allows us to naturally implement inter-thread optimizations such as tiling (collective reading shared data into shared memory and reusing them by other threads).

In this paper, we present kernel fusion as an optimization method and we show how it can be automated by our source-to-source compiler when the type of fused kernels is restricted to map and reduce. The compiler works with a *library of elementary functions* and a *script* calling functions from the library. It fuses selected functions to improve their performance and preserve the semantics defined by the script. We note that fusing all kernels does not necessarily lead to performance improvement in all cases. Thus, the compiler searches and prunes the optimization space to find efficient fusions.

We address two main use cases by our approach.

- *Using fusion-equipped libraries* A general-purpose library can be implemented to be usable with our compiler, thus fused variants of library's kernels can be easily generated. Our BLAS routines, presented in this paper, are such an example.
- *Simplification of fusion optimization* In some cases, it is quite difficult to find efficient fusions. Thus, it is worth developing both the script and the library (even if it is not widely reusable) and using our compiler to find efficient fusions automatically (one such example appears in our previous paper [6]).

To demonstrate the performance benefit of kernel fusions generated by our compiler, *we have accelerated several sequences of BLAS* (Basic Linear Algebra Subprograms) routine calls. BLAS is a library of linear algebra routines that is frequently used in scientific computation and is believed to be well optimized. The BLAS-1 (vector–vector) and BLAS-2 (matrix–vector) routines are memory bound, thus their sequences are good candidates for improvement by fusions [1,12].

[1] The first CUDA processor, G80, has flop-to-word ratio about 24, GT200 has 27, GF110 has 33, GK110 has 63 and GM204 has 82.

The rest of the paper is structured as follows. The overview of related work is given in Sect. 2. The general discussion about effect of kernel fusion on the performance as well as its automation can be found in Sect. 3, whereas Sect. 4 describes the compiler allowing automatic fusions. The performance of a code generated by our compiler is evaluated in Sect. 5. Section 6 concludes the paper.

## 2 Related work

The GPU kernel fusion is enabled in some frameworks working with algorithmic skeletons. Algorithmic skeletons are predefined higher order functions performing given user-defined first-order functions [4,8]. The SkeTo framework automatically fuses skeletons to spare global memory transfers [19]. Fusions are also possible in Thrust [11], but the programmer has to explicitly set the kernels to be fused. The significant difference of our approach is that first-order functions can be parallel, which allows them to process larger data (e.g., small tensors [6] or matrix tiles). Second difference is that fusion optimization space is traversed to discard suboptimal fusions.

In array programming, one defines transformations of whole arrays using element-wise operations, reduction, etc. [13]. Although array and skeletal programming introduce different programming models, the transformations of arrays usually perform similar operations as skeletons and there are similar fusion opportunities. The Barracuda compiler is able to fuse arrays and perform operations on these arrays in a single kernel [14]. The fusions are performed whenever possible, without considering on-chip resources consumption. A similar fusion mechanism is implemented in Copperhead [3], which is a high-level data-parallel language embedded in Python. A programmer cannot write the native code of the transformation applied to array's elements (i.e., first-order functions), thus he or she cannot explicitly define parallel per-element code or implement any low-level optimization, contrary to our approach.

A tool by Gulati and Khatri [9] automatizes the partitioning of the input code into kernels and automatically generates the code of output kernels. The input code performs serial computation, the output code performs the computation multiple times in parallel (thus, it is an application of map function).

GROPHECY is a tool for GPU performance projection from CPU skeletons [15]. It searches optimization space and suggests the user to fuse kernels when it is expected to gain performance [16]. Another performance projection tool, focused on large stencil codes (sequences of many executed kernels), is presented in [22]. The main difference of our work is that our compiler is able to generate fused kernels.

A fusion method improving energy efficiency of CUDA kernels is proposed in [23]. This method does not aim at improving the execution times of kernels, as kernels are fused without improving data locality. Similar to our method, the previously implemented kernels are fused.

Belter et al. [1] introduce BTO BLAS compiler, which is able to fuse BLAS functions targeting modern CPUs. The DESOLA active library, presented by Russell et al. [18], performs fusions in time the BLAS functions are called, i.e., without previous compilation. The main difference between our research and those presented

in [1] and [18] is that we target GPU architecture, thus the technique of the fusion differs significantly. We fuse parallel kernels instead of loops, which requires different techniques to perform the fusion correctly. Moreover, the optimization space search and performance prediction also change due to a different nature of GPUs. Our approach addresses multiple types of computational problems, whereas BTO BLAS and DESOLA focus on only BLAS. We fuse the hand-tuned routines, whereas BTO BLAS uses high-level description of BLAS routines and DESOLA implements initial BLAS functions (which are further optimized automatically) in a language similar to C, without any architecture-specific optimizations.

The importance of kernel fusion of BLAS-1 routines has been demonstrated on biconjugate gradient method [5,20]. Opposite to our paper, these papers focus on manual fusion of kernels. Moreover, off-chip memory transfers are not spared in [20] and presented fusion method is efficient mainly when using relatively small vectors.

In our previous papers, we have introduced basic principles of our compiler [7] and its non-trivial application [6]. Nevertheless, the compiler introduced in these papers was restricted to map kernels, which significantly limits its applicability. In this paper, we discuss fusion of nested map and reduce and show the structure of generated code and the process of its generation in deeper detail. Moreover, our compiler is now able to work with any data type and size and we evaluate optimization using three generations of GPU architecture.

## 3 Kernel fusion

In this section, we discuss kernel fusion in more detail, but still as a general concept, i.e., independently of the design and implementation decisions made for our compiler. First, the performance advantages and disadvantages of fusions are discussed. Second, the properties of map and reduce functions allowing their automatic fusion are described. Finally, the implementation of BLAS routines as fusible kernels is introduced.

### 3.1 Hardware model

While we focus on CUDA GPUs, the presented concept of automatic kernel fusions is generic and can be used with any hardware of the properties described in this section. Note that this section can also be used as a very basic dictionary for readers not familiar with CUDA.[2]

The properties of hardware considered for this work are as follows:

– *Many-core processor* The processor includes many ALUs, which process many light-weight *threads* in parallel. Unlike traditional CPU, threads usually solve small problem instances (e.g., one thread can execute only hundreds of instructions during its lifetime). Threads also cannot use large amounts of resources, otherwise, *occupancy* (utilized parallelism) is reduced which in general reduces performance.

[2] For more details about CUDA, we refer to [17].

– *Thread hierarchy* Threads are grouped into subsets called *threads blocks*, which have fixed and upper-bound size. Threads within the same block can communicate and synchronize (see below).
– *Distributed memory* The memory has explicit hierarchy (controlled by the programmer) and it is distributed. More precisely, each thread has its private fast memory—*registers*. The *shared memory* is a fast memory shared among all threads within the same thread block, but it cannot be shared across multiple blocks. Finally, *global memory* can be accessed by all threads from all thread blocks, but it is significantly slower than registers or shared memory. All input data and results of computation of the *kernel* (executed parallel function) have to be stored in global memory.
– *No global barrier within kernel* Within single kernel, only local barrier synchronization for threads in thread block can be executed. When a global barrier is needed, kernel has to end to ensure all thread blocks have been executed.

The hardware model used in OpenCL also fully matches the properties listed above.

### 3.2 Fusion as an optimization method

The main advantage of fusion is the improvement of memory locality. In CUDA, each kernel has to read its input from and store its output to off-chip global memory. When two kernels share some data, their fusion can hold shared data in on-chip memory—registers or shared memory.

Consider the example depicted in Fig. 1, left, where $z = f_3(f_1(x), f_2(y))$ is evaluated. When $f_1$, $f_2$ and $f_3$ are fused into a single kernel, the results computed by $f_1$ and $f_2$ can be held in on-chip memory and immediately used by $f_3$. Otherwise, the outputs of $f_1$ and $f_2$ have to be stored in global memory and loaded by $f_3$. If the performance of $f_1$, $f_2$ or $f_3$ is bounded by global memory bandwidth, the fusion increases performance by reducing global memory data transfers.

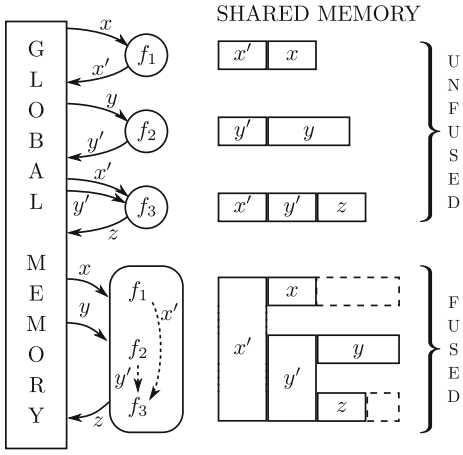


**Fig. 1** Computation of $z = f_3(f_1(x), f_2(y))$ as $x' = f_1(x)$, $y' = f_2(y)$, $z = f_3(x', y')$. *Left* data movement of unfused and fused versions. *Right* on-chip memory allocation in unfused and fused versions

An additional benefit of kernel fusion is the reduction of kernel launch overhead (a lower number of kernels is launched). Moreover, the fused kernels are more complex, thus the optimizing compiler has more room for instruction optimization, such as common subexpression elimination, loop fusion.

Besides the performance improvements mentioned above, fusion may also decrease performance. The occupancy of GPU (the number of warps that can concurrently run on GPU) must be sufficient to hide the memory latency [17]. When a kernel requires too much on-chip memory, occupancy is limited and the memory latency can decrease performance. When such a kernel is fused with another one, occupancy is limited for the whole fused kernel. Thus, the overall performance may be better when the kernel which limits occupancy is not fused.

Another factor that can limit occupancy is the storage of additional intermediate data in on-chip memory. Consider the example mentioned above. In the fused kernel, $f_1$ and $f_2$ have to be performed before $f_3$ in any ordering. Suppose that $f_1$ is performed before $f_2$. In this case, when $f_2$ is performed, the result of $f_1$ must be held in on-chip memory, thus at least for $f_2$ the consumption of on-chip memory is higher compared with the unfused version. This example is depicted in Fig. 1, right, where the execution of every unfused kernel would consume less on-chip memory compared with the fusion.

Finally, the optimal number of threads processing data elements may vary for different kernels. In this case, kernels can be implemented to use the same number of threads (i.e., at least one of them is suboptimal), or kernels with different parallelism can be fused, but parallelism needs to be reduced during the computation (i.e., some threads idle in a part of the computation). The parallelism reduction decreases fusion performance; thus, fusion of kernels with different parallelism may, or may not improve performance.

As we have shown, kernel fusion may increase as well as decrease performance. The number of possible fusions and their combinations is large (see Table 7 or [6,22]) and a manual search for the best-performing one is time consuming and error prone. Thus, the automatic generation of efficient fused code is necessary.

### 3.3 Kernel fusibility

To fuse two kernels, one has to correctly glue kernel codes into a single kernel preserving the original functionality. The automatic fusion of generic kernels is difficult for two main reasons.

- *On-chip memory is distributed* Some data, which was originally exchanged via global memory, reside in on-chip memory in fused kernel. This data can be transferred via on-chip memory when the following holds for all kernels to be fused: (i) kernels use the same mapping of threads to exchanged data placed in registers and (ii) kernels use the same mapping of thread blocks to exchanged data placed in shared memory.
- *Global barrier is not available inside kernel* Kernel execution creates a global barrier, which cannot be generally implemented within a kernel. Two or more kernels

can be fused only if this global barrier is not necessary, i.e., it can be replaced by a local barrier or avoided entirely.

In our paper, we have restricted the types of kernels to map and reduce and their nested combinations (mapped map, or mapped reduce—a map function cannot be used as a reduction operator). These kernels have a wide range of applications as map and reduce have sufficient expressive power for many computing tasks. Also, map and reduce allow automatic fusion, as it is shown below.

Using parallel first-order functions makes their fusions more complicated. Thus, we discuss how to fuse them in more detail.

### *3.3.1 Map kernels*

Let $L_i = [e_1^i, e_2^i, \ldots, e_m^i]$ be a list of $m$ elements $e_1^i, \ldots, e_m^i$. The map is a higher order function that applies a given n-ary[3] function $f$ element-wise to all elements of the lists $L_1, \ldots, L_n$, producing the list of results:

$$map(f, L_1, \ldots, L_n) = [f(e_1^1, \ldots, e_1^n), \ldots, f(e_m^1, \ldots, e_m^n)]$$

Suppose two data-dependent calls of map function $map(g, map(f, L))$, $L = [e_1, \ldots, e_n]$. The mapped functions $f$, $g$ can be fused, i.e., kernel performing $map(g \circ f, L)$ can be created, if and only if $\forall i \in [1, n]$, $f(e_i)$ runs in the same thread block as $g(f(e_i))$. It guarantees that the result of $f$ can be transferred to $g$ via on-chip shared memory, because the shared memory is visible for all threads within the same block. It also guarantees that no global barrier is needed, as no data are exchanged among different blocks. We note that single instance of each mapped function has to fit into thread block, i.e., has to use reasonable number of resources (threads, on-chip memory).

### *3.3.2 Reduce kernels*

Let $\oplus$ be a binary associative operator. The reduce is a higher order function applying given operator $\oplus$ recursively to all elements of the list $L$ building a resulting element.

$$reduce(\oplus, L) = e_1 \oplus e_2 \oplus e_3 \oplus \cdots \oplus e_n$$

The result of reduction is constructed using all elements $e_1 \ldots e_n$, therefore, a global barrier is needed to obtain the result of reduction. Consequently, the result of the reduction cannot be used within the same kernel where the reduction is computed. Nevertheless, we can fuse a reduction kernel with other kernel producing or sharing reduction's input. Because of $\oplus$ associativity, a partial reduction can be computed locally per thread block without global barrier. The final result of reduction is obtained after global barrier by reducing results of all partial reductions.

[3] Some programming languages use map only for unary functions and introduce zipwith for n-ary functions.

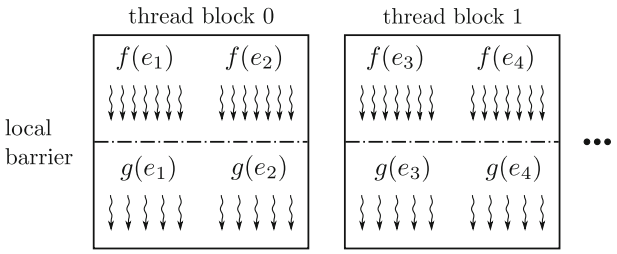


**Fig. 2** Fused kernel performing $map(g \circ f, L)$

The final result of the reduction can be computed in several ways (i) by extra kernel, (ii) by the last running block of kernel performing partial reduction (global barrier is replaced by a test of termination of all other blocks) or (iii) automatically after kernel is finished when atomic instructions are available.

### *3.3.3 Local barriers and registers*

Let $f$ and $g$ be kernels being fused. The thread-to-data mapping of $f$ and $g$ is the same if and only if each word transferred from $f$ to $g$ is stored in $f$ and loaded in $g$ by the same thread. As our mapped functions or reduce operators can be parallel, the thread-to-data mapping can differ in kernels being fused. In that case, data have to be transferred via shared memory and local barrier needs to be performed between kernel codes.

The local barrier is not needed between $f$ and $g$ when the thread-to-data mapping is the same. When all functions accessing data element $e$ access it with same thread-to-data mapping, and the access pattern is not data-dependent,[4] the element $e$ can be stored in registers.

Figure 2 illustrates an example of kernel fusion, showing fused kernel performing $map(g \circ f, L)$. In this example, two instances $g(f(e_{2i-1}))$ and $g(f(e_{2i}))$ run in a single thread block and each function is performed by a different number of threads. Let all threads of $f$ write a result in this example, thus data exchanged between $f$ and $g$ cannot be placed in registers. As no instance is divided among thread blocks, data can be passed via shared memory. Finally, a local barrier has to be used.

We do not fuse functions with different nesting depth, as it yields redundant execution of functions with lower nesting depth. We note that all rules discussed in this chapter are the same for both nested and unnested map and reduce.

## 3.4 BLAS functions expressed as map and reduce calls

In this paper, we demonstrate kernel fusion on BLAS kernels. Any kernel which is to be processed by our compiler has to be annotated by a higher level function, so the compiler knows how to fuse it. We can express BLAS-1 operations by map and reduce functions, e.g., DOT kernel ($z \leftarrow x^T y$) multiplies each element from vector

[4] Data element can be placed in registers only if their indexing can be determined at compile time [17].

$x$ by corresponding element of vector $y$ (map) and sums the results of multiplication over all elements (reduce), i.e., $z = reduce(+, map(\cdot, x, y))$. BLAS-2 kernels can be seen as mapping a BLAS-1 operation to matrix row or column, e.g., GEMV kernel ($y = Ax$) can be expressed as $y = map(reduce(+, map(\cdot, A_i, x)), A)$, where $A_i$ is i-th row of matrix $A$.

As our compiler allows to use parallel first-order functions, we process each vector as a list of subvectors and each matrix as a (nested) list of tiles instead of scalars. The first-order function is implemented as an operation on subvectors or matrix tiles of the fixed size, e.g., first-order function used for matrix–vector multiplication implements multiplication of $32 \times 32$ matrix tile by subvector of size 32, and this function is mapped across the whole matrix and results of its instances are reduced. Consequently, fine-grained optimizations can be implemented. Note that it is still possible to process vectors and matrices of any size (i.e., not divisible by 32 in our case), as the first-order function can omit or zeroize data outside actual size of matrix or vector.

## 4 The compilation process

Based on observations given in Sect. 3, we have developed a source-to-source compiler, which is able to fuse CUDA kernels. In this section, we focus on the process of creating fusions and fusion code generation and briefly describe the process of fusion space exploration, which is discussed in our previous paper [7] in more detail.

### 4.1 Compilation stages

Our compiler works with a special form of kernel implementations—we call the special form *elementary function*. The main purpose of the compiler is to transform a sequence of elementary function calls into the sequence of kernel calls, where single kernel can include one or more elementary functions, maximizing performance of output code.

Recall that the input of our compiler consists of a high-level *script* and a *library* of elementary functions. Each elementary function can be present in several alternative implementations in the library with different performance characteristics. The script calls functions from the library, thus it defines data dependencies.

The compilation process is divided into three main stages:

- parsing the script and the library (reading elementary functions and their metadata);
- generation and searching of the optimization space;
- code generation.

The script and metadata parsing is straightforward and is not discussed here. The optimization space exploration and code generation are discussed in the following sections in more detail.

### 4.2 Generation and searching the optimization space

The input script is parsed creating data dependency graph, where vertices represent elementary function calls and edges represent data dependency between functions.

Having data dependency graph built and library data parsed, the code without fusions can be generated (i.e., each elementary function is translated to a separate kernel). However, there is usually a large number of possible fusions. Thus, the optimization space is generated and searched for the code with the best expected performance. There are three main steps in the generation of the optimization space.

- *Generation of fusions* We define fusion as a fusible subgraph of data dependency graph (selection of elementary functions, which can be safely fused without influencing input program semantics). At this step, a space of all reasonable fusions is generated.
- *Generation of fusion implementations* Each fusion can be implemented in many ways, differing in (i) calling order of functions (which can affect the amount of allocated on-chip memory), (ii) chosen implementations of elementary functions, (iii) thread block size (determining the number of instances running parallel in a thread block) and (iv) number of serial iterations (determining the number of instances performed in serial in each thread block). At this step, implementations of all fusions are generated and their performances are predicted (performance prediction method is described in [7]).
- *Generation of combination of fusion implementations* The combination of fusion implementations is such a selection of fusion implementations and unfused kernels, that covers all calls of elementary functions defined in input script and maximizes predicted performance. The combination of fusion implementations and unfused kernels can be transformed to output CUDA code covering the whole computation given by the script. The generation of combinations can be repeated many times (omitting previously selected combinations) to allow empirical search for output code with the best performance.

During the generation of the optimization space, some implementations are automatically pruned, e.g., fusions that do not spare memory transfers or fusion implementations that use larger amount of on-chip memory per instance than another implementation of the same fusion. After the pruned optimization space is generated, the performance of each fusion implementation is predicted. The fusion implementations are sorted according to their predicted performance. Thus, when only one combination of fusion implementations is generated by the compiler, fusion implementations with the best predicted performance are generated. When the compiler is configured to generate more combinations, fusion implementations with worse predicted time are also generated; thus, it is possible to overcome inaccuracy of performance prediction by generating more combinations and empirical searching for the best-performing one.

### 4.3 Creating kernels from elementary functions

Recall that our compiler is not able to fuse generic kernels implemented in C for CUDA, but works with elementary functions. In fact, elementary function used by our compiler contains nearly complete code of unfused kernel—however, it must fulfill several requirements described below.

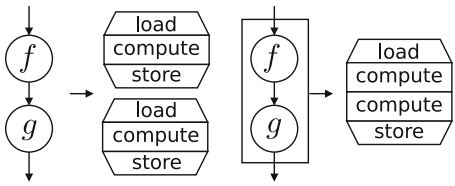


**Fig. 3** Illustration of a simple fusion

The elementary function is implemented to perform a higher order function applying a first-order function on many elements. A single instance of elementary function performs the first-order function using some input elements generating an output element, i.e., when elementary function performs $map(f, L)$, single instance performs $f(e_i)$.

To be usable with our compiler, the elementary function is associated with *metadata*, which defines its properties, such as parallelism requirements or implemented higher order function. Each elementary function has to be implemented in several *routines* (functions called from CUDA code):

- *load* (separate for each input type), loads input data stored in global memory into on-chip memory;
- *compute* performs computation on data in on-chip memory;
- *store* stores data from on-chip memory into global memory.

The decomposition of elementary function into routines is the core principle which significantly simplifies the code generation. The kernel is created as a sequence of load, compute and store routine calls. When functions are fused, the stores and loads for elements that remain in on-chip memory are not called and the remaining calls are glued into single kernel (see Fig. 3 for illustration of a simple fusion).

The compiler generates routine code, kernels calling these routines and a code encapsulating kernels allowing to empirically search for the most efficient one.

#### *4.3.1 Routine code generation*

First, the compiler generates routines: it copies their code from the library, assigns values to macros and modifies local memory addressing, when registers are used to store input or output elements [6]. Macros in routines have prescribed names and are used to determine the thread block size and the number of serial iterations. The reason for using macros is to improve optimizations performed by the CUDA compiler—expression evaluation and loop unrolling.

#### *4.3.2 Main kernel structure*

When the kernel code is created from elementary functions, the compiler knows the type of the higher order function that is implemented by the elementary function (defined in metadata). It allows the compiler to (i) generate the computation of thread and block indices and configure the block and grid size, (ii) force a global barrier

**Algorithm 1** Schema of kernel

```
1: allocate variables in shared memory
2: create arrays in registers
3: compute thread and block indices
4: load invariant data
5: clear outputs of accumulated reductions
6: for i = 0; i < iters; i++ do
7:    call non-invariant load, compute and store routines
8:    recompute block indices
9: end for
10: call stores of accumulated reductions
```

before the result of the reduction is used[5] and (iii) correctly place loads of invariant variables and stores of accumulable variables (i.e., variables that can be accumulated outside of the cycle performing sequential iterations).

The unnested function runs in a 1D grid. When more than one serial iteration is performed, the grid is adequately shrunk and block indices are recomputed in each iteration, simulating the execution of the full-sized grid. For the nested functions (recall that only nesting level 2 is supported in the current implementation), a 2D grid is used, and iterations shrink the grid in one dimension, working similarly to unnested functions. In the following paragraphs, we show structure of the generated code.

Algorithm 1 sketches the basic structure of the generated (fused or unfused) kernel. All data exchanged between routines via shared memory are allocated at the beginning of the kernel (line 1). Elements in shared memory can overlap when it is possible to spare shared memory usage [7]. This is technically realized by allocating one large array and creating pointers into this array, representing data elements. For data elements stored in registers, local arrays are defined (line 2). The size of each (per-thread) local array is set to the size of one element regardless of the fraction of element used by one thread [6] (register space is not wasted, because array indexing is determined during compilation and unused array indices are not associated to registers).

The thread and block indices are set at line 3 according to real thread and block indices for the kernel. When a routine within the kernel needs different parallelism, thread indices are recomputed before the routine is called.

For nested map or reduce, some input data elements can be invariant across iterations (e.g., for matrix–vector multiplication, a subvector can multiply several matrix tiles), thus invariant loads are called before the loop (line 4). Both nested and unnested variants of reduce can accumulate their results across iterations, thus their results are cleared before the loop (line 5) and stored after the loop finishes (line 10). The rest of the routines is called within the loop (line 7) according to selected calling order and the block indices are recomputed at the end of each iteration (line 8).

Fusion of nested and unnested functions is not efficient, as it results in repetition of unnested operations and hence does not spare global memory transfers. Therefore, our compiler does not fuse functions with different nesting level. Consequently, compute

[5] This is trivially fulfilled in code generation stage, as outputs of all reductions are used outside of the fusion implementation performing the reduction, thus the global barrier is performed by finishing the kernel.

**Algorithm 2** Schema of routine call

```
1: call local barrier
2: clear output of the reduction
3: if thread participates then
4:    recompute thread indices
5:    call routine
6: end if
```

routines are always performed within the loop, as no result of a compute routine performed within the fusion is invariant across loop iterations.

### *4.3.3 Generation of routine call*

In Algorithm 1, routines are called at lines 4, 7 and 10. The more detailed schema of a generated routine call is described in Algorithm 2. First, the local barrier call can be generated. The local barrier before routine $r$ is generated, if one of the following conditions holds.

- Routine $r$ accesses at least one input element $e$, that has been modified by routine $s$, and (i) thread-to-data mapping of access to $e$ is different for $r$ and $s$ and (ii) no local barrier is called between $r$ and $s$ calls so far.
- Routine $r$ writes the element $e$ into shared memory, and $e$ overlaps with another element $e'$, that is accessed after the last synchronization called before $r$.

The first condition ensures that all words of the element $e$ are written into shared memory before they are read by $r$. The second condition provides synchronization of all warps before element $e'$ is rewritten by $e$ to ensure that all routines accessing $e'$ are finished before its rewriting.

When the routine performing reduction is to be called and its output is not accumulated among iterations, the code clearing its output is generated at line 2.

If the routine is performed by a lower number of threads than it is available within the kernel, lines 3 and 6 reducing the parallelism are generated. The code reducing parallelism is created to keep maximum of warps fully utilized, i.e., when parallelism is reduced from $m$ to $n$ threads, threads $<0, \ldots, n-1>$ continue in computation whereas threads $<n, m-1>$ stall.

The thread indices recomputation (line 4) is generated when parallelism is reduced, or when the thread arrangement of the routine differs from the thread arrangement of the fusion (e.g., a routine needs a block of $9 \times n \times 1$ threads for $n$ instances, whereas the fused kernel uses block of $3 \times 3 \times n$ threads, i.e., the same number of threads, but different indexing). The compiler generates the indexing computation that maps adjacent indices to adjacent threads to create at most one under-populated warp. Moreover, as it knows the number of threads in each dimension required by routines in compile time, it optimizes the number of arithmetic operations needed to recompute indices. After the parallelism is restricted and indices are recomputed, the routine can be called (line 5) with new indices.

### 4.4 An example of code generation

To demonstrate the compiler's features described above, we have chosen the computation of BiCGK sequence as an example. The sequence performs

$$q = Ap$$
$$s = A^T r$$

It demonstrates kernel fusion ($q = Ap$ and $s = A^T r$ are implemented as separate elementary functions) as well as working with a nested operation.

Recall that the vector and matrix elements can be represented by a single number, or a larger structure. We are using a 32-number subvector as a vector element and a $32 \times 32$ tile as a matrix element. These element sizes allow us to write efficient elementary functions, such as $q = Ap$ or $s = A^T r$, where single instance multiplies $32 \times 32$ matrix tile by subvector of size 32, giving a good opportunity for manual optimizations. When a vector or matrix is of size not divisible by 32, data outside the vector or matrix boundaries are zeroized in load routines and are not stored in store routines.

```
1  TILE32x32 A;
2  subvector32 p, q, r, s;
3
4  input A, p, r;
5
6  q = sgemv(A, p);
7  s = sgemtv(A, r);
8
9  return q, s;
```

**Listing 1** Script performing BiCGK sequence

We have implemented elementary functions `sgemv` ($q = Ap$) and `sgemtv` ($s = A^T r$). The script performing BiCGK sequence is listed in Listing 1.

The pseudo-code of the generated fused kernel of BiCGK sequence is listed in Algorithm 3. The algorithm has several inputs: $A$ is an $m \times n$ matrix, $p, s$ are vectors of size $m$, and $q, r$ are vectors of size $n$. Load, compute and store routines, which are called in the generated code, are present in the library of elementary functions. In the optimization space searching phase, the compiler has decided to perform several serial iterations in each instance, thus, the `for` loop going over several matrix tiles is generated in the kernel.

The code generation works as follows. The compiler generates a shared memory allocation for all on-chip variables. Each variable in shared memory can be padded (the padding is configured by the developer of elementary function). In this example, the $x$-dimension of $A$ is enlarged by 1, so $A$ is allocated as array of size $33 \times 32$. The padding removes shared memory bank conflicts when a warp accesses column of $A$.[6] After memory allocation, the compiler generates the computation of the thread indices and block indices $x$, $y$, where $y$ is stridden according to the number of serial iterations $i$. When indices are computed, the routines can be called. The local part

[6] For more details about shared memory bank conflicts, we reffer to [17].

**Algorithm 3** Fused $q = Ap, s = A^T r$

```
1: allocate A_l, p_l, q_l, r_l, s_l in shared memory
2: compute thread indices
3: compute tile indices x ← block.x, y ← i · block.y
4: p_l ← load(p, x)
5: s_l ← 0
6: for y' = y; y' < min(n, y + i); do
7:    r_l ← load(r, y')
8:    A_l ← load(A, x, y')
9:    s_l ← compute_gemtv(A_l, r_l, x, y')
10:   q_l ← 0
11:   q_l ← compute_gemv(A_l, p_l, x, y')
12:   q ← store(q_l, y')
13:   y' ← y' + 1
14: end for
15: s ← store(s_l, x)
```

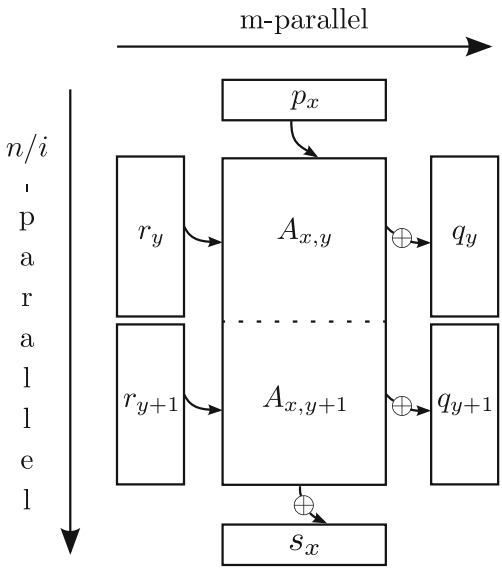


**Fig. 4** A data usage of a single instance of the fused BiCGK

of vector $p$ loaded into $p_l$ is invariant across iterations, and the output of the partial reduction $s_l$ can be accumulated across iterations. Thus, the compiler puts loading of $p_l$ and zeroing of $s_l$ before the loop. Within the `for` loop, the local part of $r$ and $A$ are loaded to $r_l$ and $A_l$, and $A_l^T r_l$ is computed and added to $s_l$ (line 9). To compute $A_l p_l$ (line 11), $p_l$ is zeroed (line 10) and stored (line 12) after the computation in each iteration. When all the iterations are finished, accumulated result in $s_l$ is stored. For simplicity, local synchronizations are not shown in the pseudo-code.

Figure 4 illustrates the data movement in the computation. The single instance of BiCGK processes two tiles of $A$ in depicted example ($i = 2$), thus two subvectors of vectors $r, q$, and one subvector from both $p$ and $s$ are moved between global and on-chip memory. The instances are created $m \times \frac{n}{2}$ times over the matrix.

The important property of the algorithm described above is that $Ap$ can be fused together with $A^T r$, although both matrix rows and columns are used in inner vector

dot product—the only difference is in the placement of routines call with respect to the loop (for invariants or accumulable output).

## 5 Evaluation

In this section, the optimization of sequences of BLAS routines is evaluated. First, various sequences of linear algebra kernels used in our experiments are defined and the possibilities for optimizing them are analyzed. Second, the performance of implementations generated by our compiler is evaluated and compared with CUBLAS and unfused implementations. Third, our speedup is compared with BTO BLAS using CPU. Finally, the accuracy of the performance prediction method and compiler timing is analyzed.

### 5.1 Experiment setup

To test the code efficiency of our compiler, we have used the same sequences as in [1], which are specified in Table 1. These sequences are a representative selection of generally interesting operations, where many of them have important applications (BiCGK is used in biconjugate gradient method, ATAX in normal equations computation), are added to the updated BLAS specification [2] (AXPYDOT, GEMVT, GEMVER, GESUMMV, WAXPBY), are in original BLAS specification (GEMV and SCAL) or are generally usable (MADD, VADD). Some of these sequences can be significantly improved by fusions whereas others cannot. The adoption of sequences

**Table 1** Sequences used in our performance study, adopted from [1]

| Sequence | Operation | Tag |
|---|---|---|
| AXPYDOT | $z \leftarrow w - \alpha v$ | FS |
| | $r \leftarrow z^T u$ | |
| ATAX | $y \leftarrow A^T Ax$ | |
| BiCGK | $q \leftarrow Ap$ | F |
| | $s \leftarrow A^T r$ | |
| GEMV | $z \leftarrow \alpha Ax + \beta y$ | B |
| GEMVT | $x \leftarrow \beta A^T y + z$ | (S) |
| | $w \leftarrow \alpha Ax$ | |
| SCAL | $x \leftarrow \alpha x$ | B |
| GEMVER | $B \leftarrow A + u_1 v_1^T + u_2 v_2^T$ | FS |
| | $x \leftarrow \beta B^T y + z$ | |
| | $w \leftarrow \alpha Bx$ | |
| GESUMMV | $y \leftarrow \alpha Ax + \beta Bx$ | (F) |
| MADD | $C \leftarrow A + B$ | S |
| VADD | $x \leftarrow w + y + z$ | FS |
| WAXPBY | $w \leftarrow \alpha x + \beta y$ | F |

Tags: *F* improvable by the fusion, *S* improvable by kernel specialization, *B* equivalent of CUBLAS kernel

**Table 2** The most relevant parameters of GPUs used in our tests

| GPU | Memory bandwidth | SP performance | DP performance |
|---|---|---|---|
| M2090 | 177 GB/s | 1332 GFlops | 666 GFlops |
| K20 | 208 GB/s | 3520 GFlops | 1170 GFlops |
| GTX980 | 224 GB/s | 4612 GFlops | 144 GFlops |

from [1] allows us to compare effect of fusion on two different processors—multi-core CPU and many-core GPU.

We have assigned tags to each sequence in Table 1. These tags indicate optimizations that our compiler is able to perform. Tag F indicates that fusion can be used to improve performance. Tag S indicates that more specialized kernels saving some work compared with CUBLAS can be generated. Finally, tag B indicates sequences that have their equivalents in CUBLAS—in this case, any optimization that can be used by our compiler can also be implemented in CUBLAS. When a tag is enclosed in brackets, its significance is low, i.e., it is related to BLAS-1 operations in sequences where much more time-consuming BLAS-2 operations are executed.

For some sequences, the tag assignment does not have to be straightforward, thus we discuss it in more detail.

- ATAX and GEMVT cannot be improved by fusion. In both cases, matrix $A$ is used twice, but a global barrier is needed between uses of $A$, and thus must be used in separate kernels.
- GESUMMV can spare the reading of vector $x$ when it is performed in a single kernel. However, because of reading the matrices $A$ and $B$, the amount of data transfer is almost the same in the fused and unfused versions.
- All sequences with the S tag require memory copy or cleaning in the CUBLAS implementation because of the in-place implementation of some CUBLAS kernels, whereas kernels generated by our compiler can work out-of-place when needed.

Note that several sequences used in this evaluation cannot be improved by fusion or kernel specialization. Our motivation to evaluate these sequences is to compare their performance with hand-tuned CUBLAS. More precisely, we can identify potential slowdown in cases, where our compiler cannot make any optimization unavailable to CUBLAS developers.

The performance of sequences listed in Table 1 has been measured using CUDA 6.5 on three different GPU architectures: Fermi (Tesla M2090), Kepler (Tesla K20) and Maxwell (GeForce GTX980). Basic parameters of these GPUs are listed in Table 2. ECC has been activated at Tesla GPUs. We have measured performance using both double precision (DP) and single precision (SP) arithmetic. Although consumer Maxwell GPU with reduced DP performance has been used, it is still reasonable to measure DP performance as all kernels in this study are heavily memory bound.

### 5.2 Performance results

In this section, we measure the performance of our fused code using sequences in Table 1 and compare their performance with sequences implemented with CUBLAS.

We also show fusion speedup separately, i.e., compare performance of our generated fused code and our generated code without fusions. Finally, we compare our speedup on GPU with speedup reached by BTO BLAS using CPU.

### *5.2.1 Measured performance and CUBLAS comparison*

Performance of implementations generated by our compiler is listed and compared with CUBLAS in Tables 3 (single precision) and 4 (double precision). In the perfor-

**Table 3** Performance (in GFlops) of generated sequences and the speedup over CUBLAS

| | Fermi (M2090) | | Kepler (K20) | | Maxwell (GTX980) | |
|---|---|---|---|---|---|---|
| | Perf. | Speedup | Perf. | Speedup | Perf. | Speedup |
| AXPYDOT | 28.38 | 1.85× | 30.46 | 1.84× | 40.44 | 1.67× |
| ATAX | 48.18 | 1.11× | 52.36 | 1.11× | 86.76 | 1.02× |
| BiCGK | 86.97 | 2.02× | 79.43 | 1.68× | 169.1 | 1.99× |
| GEMV | 48.00 | 0.96× | 53.32 | 1.15× | 85.21 | 1.03× |
| GEMVT | 48.04 | 1.11× | 51.85 | 1.10× | 85.87 | 1.01× |
| SCAL | 14.15 | 0.95× | 15.07 | 0.94× | 20.15 | 0.96× |
| GEMVER | 38.27 | 1.72× | 38.83 | 1.56× | 78.28 | 2.24× |
| GESUMMV | 50.24 | 1.33× | 53.49 | 1.11× | 86.6 | 0.99× |
| MADD | 8.53 | 1.37× | 8.82 | 1.26× | 13.6 | 1.59× |
| VADD | 15.44 | 2.04× | 17.68 | 2.14× | 21.07 | 1.48× |
| WAXPBY | 31.88 | 2.05× | 33.75 | 2.00× | 41.22 | 1.93× |

Single precision arithmetics is used in all sequences

**Table 4** Performance (in GFlops) comparison of generated and CUBLAS implementations of studied sequences using double precision arithmetics

| | Fermi (M2090) | | Kepler (K20) | | Maxwell (GTX980) | |
|---|---|---|---|---|---|---|
| | Perf. | Speedup | Perf. | Speedup | Perf. | Speedup |
| AXPYDOT | 19.01 | 1.71× | 16.65 | 1.72× | 20.14 | 1.78× |
| ATAX | 27.84 | 0.86× | 24.78 | 0.77× | 36.15 | 0.84× |
| BiCGK | 46.79 | 1.45× | 37.81 | 1.18× | 64.39 | 1.50× |
| GEMV | 28.08 | 0.81× | 24.96 | 0.80× | 40.10 | 0.96× |
| GEMVT | 27.81 | 0.86× | 24.74 | 0.77× | 36.04 | 0.84× |
| SCAL | 9.56 | 0.97× | 9.27 | 1.00× | 10.18 | 0.99× |
| GEMVER | 18.45 | 1.61× | 22.31 | 1.53× | 29.55 | 1.63× |
| GESUMMV | 22.71 | 0.98× | 25.01 | 0.76× | 40.28 | 0.91× |
| MADD | 3.51 | 1.08× | 3.48 | 0.92× | 6.97 | 1.64× |
| VADD | 7.96 | 1.98× | 9.14 | 1.98× | 10.66 | 2.04× |
| WAXPBY | 16.35 | 2.04× | 18.45 | 2.01× | 20.84 | 1.99× |

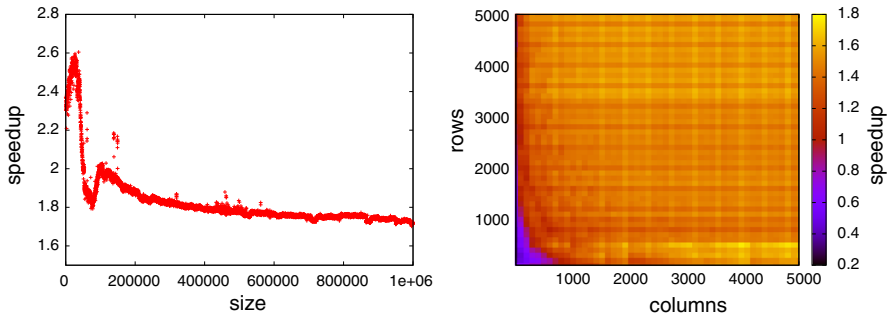


**Fig. 5** Speedup of DP AXPYDOT (*left*) and DP GEMVER (*right*) for various input sizes using Kepler

mance evaluation of sequences using BLAS-2, matrices of size $5000 \times 5000$ (and thus vectors of size 5000) have been used. For sequences working only with BLAS-1, the size of vectors has been set to 1,000,000. The compiler has been set to generate 1000 implementations at most and the fastest one has been found using empirical search.

The speedup in single precision varies from 0.94 (SCAL at Kepler) to 2.24 (GEMVER at Maxwell). We consider it to be a good result as the reached slowdown is very small only, whereas speedup is significant. If double precision is considered, the speedup varies from 0.76 (GESUMMV on Kepler) to 2.04 (VADD at Maxwell, or WAXPBY at Fermi). The source of noticeable slowdown of double precision calculations is in matrix–vector multiplication. More precisely, when a matrix tile is multiplied by a subvector, the reduction has to be performed. We use square tiles of $n \times n$ elements, and they are processed by $n \times m$ threads, where $m > 1$ for good GPU occupancy. Therefore, the reduction needs to be performed across threads. Considering SP, atomic operations in shared memory can be used. However, no native atomic operation for DP is supported by current GPUs, thus our implementation serializes warps during reduction decreasing performance.[7]

Significant speedup is obtained in the case of sequences where the fusion can improve memory locality (tag F) as well as when kernel specialization is possible (tag S). Those sequences show the strength of our compiler, as they are improved by compiler's optimizations which cannot be implemented in CUBLAS (without modification of its API).

The measurements mentioned above have been done using fixed size of data. To show the speedup for wider range of data size, we have selected two representative sequences—AXPYDOT for vector operations and GEMVER for matrix operations (both using DP) and we measure the speedup for different input sizes on Kepler (see Fig. 5). In the case of small structures, the speedup can be higher (AXPYDOT), or a slowdown can be obtained (GEMVER). However, for larger data structures, the speedup is relatively stable.

[7] It is naturally possible to use rectangular tiles, which decrease the reduction overhead. However, such tiles forbid efficient fusion of operations working with matrix and its transposition.

**Table 5** Fusion speedup

| | Fermi (M2090) | | Kepler (K20) | | Maxwell (GTX980) | |
|---|---|---|---|---|---|---|
| | SP | DP | SP | DP | SP | DP |
| AXPYDOT | 1.25× | 1.28× | 1.39× | 1.22× | 1.28× | 1.25× |
| BiCGK | 1.81× | 1.81× | 1.52× | 1.53× | 1.95× | 1.78× |
| GEMVER | 2.15× | 2.35× | 2.00× | 2.76× | 2.67× | 2.08× |
| GESUMMV | 1.05× | 1.01× | 1.01× | 1.00× | 1.02× | 1.01× |
| VADD | 1.47× | 1.47× | 1.75× | 1.53× | 1.53× | 1.53× |
| WAXPBY | 2.51× | 2.33× | 2.62× | 2.36× | 2.36× | 2.36× |

**Table 6** Range of speedups measured with BTO BLAS and our compiler

| Sequence | BTO speedup | Our speedup |
|---|---|---|
| AXPYDOT | 1.22 × –1.58 × | 1.67 × –1.85× |
| ATAX | 1.05 × –1.37 × | 0.77 × –1.11× |
| BiCGK | 1.12 × –1.50 × | 1.18 × –2.02× |
| GEMV | 0.61 × –0.83 × | 0.80 × –1.15× |
| GEMVT | 0.94 × –1.29 × | 0.77 × –1.11× |
| GEMVER | 2.04 × –2.37 × | 1.53 × –2.24× |
| GESUMMV | 0.75 × –0.93 × | 0.76 × –1.33× |
| MADD | 1.35 × –1.47 × | 0.92 × –1.59× |
| VADD | 1.13 × –1.83 × | 1.48 × –2.14× |
| WAXPBY | 1.18 × –1.88 × | 1.93 × –2.05× |

### *5.2.2 Fusion speedup*

We have compared the performance of fused and unfused code generated by our compiler. It allows us to evaluate the effect of the fusion optimization only, separately from other effects that can arise when we compare our kernels with CUBLAS (higher or lower CUBLAS implementation efficiency and kernels specialization). In Table 5, the speedup of the fused kernels compared with unfused ones is shown. Only the sequences with tag F (e.g., where the fusion is possible) are listed.

In this case, fused kernels always perform the same or better than unfused ones. The speedup ranges from 1 (in GESUMMV, where the effect of the fusion is negligible) to 2.76 (in GEMVER, where there is a lot of room for fusion optimization). The fusion speedup is similar in SP and DP, contrary to the comparison with CUBLAS, where a less efficient DP matrix–vector multiplication decreases the speedup of our code.

### *5.2.3 BTO BLAS comparison*

To the best of our knowledge, there is no other system allowing automatic fusion of BLAS functions for GPUs that could be compared with our results. Nevertheless, we can compare the relative speedup of our generated codes with relative speedup of CPU code generated by BTO BLAS [1], see Table 6. Note that the speedup comparison is

not precise, as different data sizes are used and our measurements are both for DP and SP—the purpose of this comparison is only to outline performance differences between GPU and CPU fusions.

Our speedup is higher than the speedup of BTO BLAS where fusion can be used (sequences AXPYDOT, BiCGK, VADD, WAXPBY), except GEMVER, where our speedup is lower. BTO BLAS has a wider opportunity to enhance code performance by fusions.[8] The wider fusion opportunity of BTO BLAS caused better speedup in sequences ATAX and GEMVT. The higher speedup of our fused code is probably achieved because we fuse a hand-tuned code, whereas BTO BLAS generates code from high-level description.

### 5.3 GPU architectures

We have used three generations of GPU architecture in this evaluation. The library of elementary functions is the same for all GPUs, thus all GPU-specific optimizations are up to the compiler. Nevertheless, based on our experience with Kepler architecture, we have added alternative implementations of elementary functions for matrix addition ($C \leftarrow A + B$) and rank-1 update ($w \leftarrow uv^T$) into the library. These alternative implementations use more threads per matrix tiles, which allow Kepler to hide global memory latency better.

We have analyzed the best-performing kernels generated for different GPU architectures or for different floating-point precisions. Even for simple sequences, best kernels differ significantly in thread block size and number of serial iterations, depending on the target GPU and the precision used. However, there are also differences in fusions. In SP GEMVER, kernels using higher parallelism (rank-1 update and matrix addition) were fused with lower-parallel matrix–vector multiplication for Kepler and Maxwell (thus, parallelism has been reduced during matrix–vector multiplication). On the other hand, the fastest SP GEMVER at Fermi uses the same parallelism level. The fastest DP GEMVER does not fuse matrix–vector multiplication for any GPU architecture.

### 5.4 Performance prediction accuracy

We have analyzed the accuracy of the performance prediction. As numerous possible implementations can be generated, good performance prediction makes it possible to reduce the empirical search to several promising candidates or eliminate empirical searching entirely. The results are shown for double precision computations.

[8] When the function $f$ performs reduction on each row of the matrix and the reduction's result is an input of function $g$ processing the same row, CPU is able to hold the row in the cache and reuse it after reduction finish (thus, outer loops in $f$ and $g$ going over rows are fused, whereas inner loops are unfused). Considering GPU, the row needs to be partitioned among more thread blocks when it is read into on-chip memory by $f$, thus thread blocks need to be synchronized before the result of the reduction is available. Our compiler performs the synchronization by a new kernel invocation, thus all on-chip data are lost before the result of the reduction is available for $g$, so no row data can be reused. The only way to reuse row data on GPU is to use persistent threads [10], but it is not clear if it could have a positive performance impact.

**Table 7** For each studied sequence, the second column shows the count of all implementations that have been generated

In the next columns, performance of the first implementation generated is compared with the best implementation of all possible implementations. In other worlds, this table shows how the performance differs when no empirical search is performed

| Sequence name | Impl. count | Fermi no search (%) | Kepler no search (%) | Maxwell no search (%) |
|---|---|---|---|---|
| AXPYDOT | 25 | 91 | 99 | 99 |
| ATAX | 1 | 100 | 100 | 100 |
| BiCGK | 5 | 99 | 100 | 100 |
| SGEMV | 121 | 100 | 100 | 100 |
| SGEMVT | 41 | 100 | 100 | 100 |
| SSCAL | 1 | 100 | 100 | 100 |
| GEMVER | 1000 | 60 | 68 | 57 |
| GESUMMV | 585 | 99 | 95 | 100 |
| MADD | 1 | 100 | 100 | 100 |
| VADD | 41 | 97 | 100 | 99 |
| WAXPBY | 121 | 98 | 100 | 99 |

Table 7 shows the number of possible implementations of each sequence (upper-bound by 1000) and the relative performance of the first implementation generated (i.e., the implementation with the highest predicted performance). Excepting GEMVER, the first implementation is close to the fastest implementation or is the fastest one. In the case of GEMVER, the fastest implementation has been found within first 11 implementations for all GPUs. Thus, the empirical search is necessary for good performance of GEMVER, but only few implementations need to be searched.

### 5.5 Compilation time

In this section, we have analyzed the compilation and empirical search time. All measurements have been done using Intel Core i7-3820 (3.6 GHz). When empirical search is not required, running time of our compiler is negligible—under 0.2 s in all cases. When empirical search is used, our compiler needs to generate more implementations, and these implementations have to be compiled by nvcc and benchmarked. The generation of 1000 implementations of GEMVER takes 75 s. The empirical search (nvcc compilation and benchmarking) takes 12 s at most for one implementation (for GEMVER). Thus, empirical search of all possible GEMVER implementations takes several hours. However, it has been shown that only a few implementations need to be tested to find the one with the best performance (see previous section). Thus, several minutes of empirical search is enough to find best-performing implementation for each sequence tested in this paper.

## 6 Conclusions and future work

In this paper, we have significantly extended our approach to automatic kernel fusion by introducing the fusion of (nested) map and reduce kernels. Moreover, we have modified our compiler to work with data structures of arbitrary size (i.e., not divisible by the element size). We have shown that kernel fusion can improve the performance of

highly tuned memory-bound kernels. Although the fusion of generic kernels is difficult to automate, we have argued that the automation is possible and demonstrated the automation for (possibly nested) map and reduce kernels using our source-to-source compiler. The application of our compiler has been demonstrated by fusing sequences of BLAS calls, where significant speedup comparing to CUBLAS has been observed.

We plan to focus on generalization of the presented method, making it possible to fuse more types of kernels, to work with irregular data structures or to target multiple GPUs.

- *Support for more types of fusible kernels* A more general model of temporal locality in GPU memory hierarchy could be formulated, which would allow us to handle more types of kernels, such as stencils, scatters or gathers. Supporting more general kernels significantly extends the applicability of our compiler, e.g., in image processing, ODE solvers, or Finite Difference Method.
- *Support for irregular data types* The operations working with irregular data types, such as triangular or diagonal matrices, or sparse arrays, need more general higher order functions, than are currently supported. Irregular or sparse structures enrich application area of our compiler, e.g., for large simulation of physical phenomena.
- *Support for multi-GPU computations* To allow scaling of GPU applications, the workload needs to be distributed among multiple GPUs. Although the distribution of map and reduce is quite straightforward, more complicated functions, such as nested map and reduce or stencils, yield significantly more difficult data exchange pattern.

Besides improving the compiler, we will develop libraries of elementary functions. We plan to implement more functions from the BLAS standard that are fusible by the compiler and a library of linear algebra operations on small elements usable for element subroutines in FEM.

**Acknowledgments** This work was supported by Ministry of Education, Youth and Sport of the Czech Republic under the Project "CERIT Scientific Cloud" (No. ED3.2.00/08.0144). The first author was supported by the Ministry of Education, Youth, and Sport Project CZ.1.07/2.3.00/30.0037—Employment of Best Young Scientists for International Cooperation Empowerment.

## References

1. Belter G, Jessup ER, Karlin I, Siek JG (2009) Automating the generation of composed linear algebra kernels. In: Proceedings of the conference on high performance computing, networking, storage and analysis (SC09), ACM, 2009, pp 1–12
2. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2002) An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Softw 28:135–151
3. Catanzaro B, Garland M, Keutzer K (2011) Copperhead: compiling an embedded data parallel language In: The 16th ACM symposium on principles and practice of parallel programming (PPoPP)
4. Cole M (1989) Algorithmic skeletons: structural management of parallel computation. Research monographs in parallel and distributed computing. MIT Press, Cambridge
5. Dehnavi MM, Fernandez DM, Giannacopoulos D (2011) Enhancing the performance of conjugate gradient solvers on graphic processing units. IEEE Trans Magn 47:1162–1165
6. Filipovič J, Fousek J, Lakomý B, Madzin M (2012) Automatically optimized GPU acceleration of element subroutines in finite element method. In: Symposium on application accelerators in high-performance computing (SAAHPC)

7. Fousek J, Filipovič J, Madzin M (2011) Automatic fusions of CUDA-GPU kernels for parallel map. In: Second international workshop on highly-efficient accelerators and reconfigurable technologies (HEART), pp 42–47
8. González-Vélez H, Leyton M (2010) A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw Pract Exp 40:1135–1160
9. Gulati K, Khatri SP (2009) An automated approach for simd kernel generation for GPU based software acceleration. In: Symposium on application accelerators in high performance computing (SAAHPC)
10. Gupta K, Stuart JA, Owens JD (2012) A study of persistent threads style GPU programming for GPGPU workloads. In: Innovative parallel computing
11. Hoberock J, Bell N (2009) Thrust: a parallel template library
12. Howell GW, Demmel JW, Fulton CT, Hammarling S, Marmol K (2008) Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Trans Math Softw (TOMS) 34:1–14
13. Iverson KE (1962) A programming language. In: Spring joint computer conference (AIEE-IRE)
14. Larsen B (2011) Simple optimizations for an applicative array language for graphics processors. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP), 2011
15. Meng J, Morozov VA, Kumaran K, Vishwanath V, Uram TD (2011) Grophecy: GPU performance projection from CPU code skeletons. In: International conference for high performance computing, networking, storage and analysis (SC11)
16. Meng J, Morozov VA, Vishwanath V, Kumaran K (2012) Dataflow-driven gpu performance projection for multi-kernel transformations. In: International conference for high performance computing, networking, storage and analysis (SC12)
17. NVIDIA, CUDA C Programming Guide, version 6.5., (2014)
18. Russell FP, Mellor MR, Kelly PH, Beckmann O (2011) DESOLA: an active linear algebra library using delayed evaluation and runtime code generation. Sci Comput Program 76:227–242
19. Sato S, Iwasaki H (2009) A skeletal parallel framework with fusion optimizer for GPGPU programming. In: Programming languages and systems, vol 5904 of Lecture Notes in Computer Science. Springer Berlin
20. Tabik S, Ortega G, Garzón EM (2014) Performance evaluation of kernel fusion blas routines on the GPU: iterative solvers as case study. J Supercomput 70:577–587
21. Tarditi D, Puri S, Oglesby J (2006) Accelerator: using data parallelism to program GPUs for general-purpose uses, SIGARCH Computer Architecture News, 34
22. Wahib M, Marutama N (2014) Scalable kernel fusion for memory-bound GPU applications. In: International conference for high performance computing, networking, storage and analysis (SC14)
23. Wang G, Lin Y, Yi W (2010) Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: IEEE/ACM international conference on green computing and communications and international conference on cyber, physical and social computing (GREENCOM–CPSCOM)