Optimizing Memory Bandwidth Efficiency with User-Preferred Kernel Merge

Nabeeh Jumah[1] (✉) and Julian Kunkel[2]

[1] Universität Hamburg – `Jumah@informatik.uni-hamburg.de`
[2] University of Reading – `j.m.kunkel@reading.ac.uk`

Abstract. Earth system modeling computations use stencils extensively while running many kernels. Optimal coding of the stencils is essential to efficiently use memory bandwidth of an underlying hardware. This is important as stencil computations are memory bound.
Even when the code within one kernel is written to optimally use the memory bandwidth, there are still opportunities for further optimization at the inter-kernel level. Stencils naturally exhibit data locality, and executing a sequence of stencils within separate kernels could waste caching capabilities. Interprocedural optimizations such as merging of kernels bears the potential to improve the use of the caches. However, due to semantic restrictions, it is difficult to achieve on general purpose languages.
Some tools were developed to automatically fuse loops instead of the manual optimization. However, scientists still implement fusion in different levels of loop nests manually to find optimal performance. To allow scientists to still apply loop fusions equal to manual loop fusion, we develop a technique to automatically analyze the code and allow scientists to select their preferred fusions by providing automatic dependency analysis and code transformation; this also bears the potential for automatic tools that make smart choices on behalf of the user. Our work is done using GGDML language extensions which enables performance portability over different architectures using a single source code.

Keywords: HPC; Earth system modeling; Software development

1 Introduction

Earth system modeling codes consist of many kernels, in which stencil operations are applied. Values of variables at spatially-neighboring points are read to evaluate some variable at some point in space. Neighborhoods give an opportunity to use the locality of data through caches. On the other hand, the arithmetic intensity of such computations is low, which makes them memory bound.

Efforts on optimizing operations within a kernel that applies a stencil operation is essential to optimize code performance, however, it is not sufficient. Taking into account the relationships between the consecutive kernels, it is sometimes possible to still improve performance. Reusing the data across stencil operations while still in caches makes this possible.

To exploit this inter-kernel possibilities, data dependencies should be applied to guarantee computation correctness. After making sure that a loop fusion does not impair the code in terms of computation correctness, the code should be transformed to apply the fusion. Instead of doing such effort by a programmer, tools have been developed to automatically apply such optimization.

Nested loops allow fusing loops in different combinations. Automatic loop fusions do not allow testing a specific set of loop fusions, which scientists would evaluate, may be to try other possible optimizations. We see such cases, e.g. where an outer loop encloses a set of consecutive second level nest each of which contains another inner loop, with many kernels fused in more complicated structure.

To allow scientists to still exploit loop fusion possibilities, while doing minimal effort, we develop a technique to apply preferred loop fusions that operates on the higher-level code abstraction of a domain-specific language. The **main contribution** of this work is a technique to automatically identify possible loop fusions with the necessary data dependency analysis, and apply the fusions which the user prefers. To maximize the benefit of this effort, we also allow automatic analysis for inter-module function inlining possibilities. This allows to fuse loops among different files within the source code.

2 Related Work

In this section, we review some research efforts which applied loop fusion in different ways and development contexts.

Data re-use and loop restructuring: An optimization alogorithm to reuse data was presented in [16], where loop nests were transformed by interchange, reversal, skewing, and tiling. Loop fusion and distribution to improve data locality was used in [9] besides to optimizing loop parallelism. A cost model that computes the spatial and the temporal reuse of cache lines was used in [10] to enable compiler optimizations using data locality. The authors used loop fusion as one of the transformations besides to loop permutation, distribution, and reversal.

Applicability of loop fusion: Fusion concept was used to serve optimization in differnt fields where performance is a main concern. An algorithm was presented in [5] in which loop fusion is used to reduce the use of temporary arrays. This effort was used to reduce the access to memory in data dominated applications like multimedia applications. Also, loop fusion was used for the purpose of energy consumption optimization. Fusion was proposed also to reduce the energy consumption [15] and improve the efficiency of power use on GPUs.

Compiler optimizations to exploit the efficiency of the GPUs computational power for the data warehousing applications were proposed in [18]. The benefits of the loop fission and fusion on relational algebra operators are also evaluated. Again we see the code fissions and fusions in the same field in [17] where the split and fused loops are dynamically scheduled on CUDA streams and dispatched to the GPUs to improve the performance when running queries.

Automatic loop fusion tools: Manual loop fusion is time consuming. Automatic fusion was the alternative in many efforts. A source-to-source compiler was presented in [4] to automatically apply fusion. Other efforts focused on the identification of opportunities to apply loop fusion and to estimate its benefits like [11]. This effort presented a dataflow-based framework that analyzes a provided code to identify multi-kernel optimization opportunities and data management. The framework can then estimate the performance on GPUs without running it.

Finite difference method was also subject to the automatic analysis for fusion [13], where the space of possible kernel fusions is explored to find an optimal kernel fusion. Projections of the performance are done to get to the optimal kernel fusion. The authors again prposed a framework [14] to automatically transform stencil codes written in CUDA to exploit the data locality among multiple kernels. A compiler were also used in [3] to automatically fuse loops. CUDA kernel fusion was done on BLAS-1 and BLAS-2 routines.

Loop fusion through DSLs: Other efforts, e.g. [1] used DSLs which were designed to allow code generation of fused loops. Gridtools provides a DSL to specify stencil operations in a way that allows the user to define a computation in stages within the source code. The code generation process makes use of this information to exploit the data locality.

Directives are used in HybridFortran [12] to control the granularity of code. HybridFortran was developed to allow the user to port existing CPU code to GPUs by annotating the code with directives. The HybridFortran directives allow the tools to generate the code with the suitable granularity based on the target machine.

In contrast, **with our method**, the user does not need to manually fuse loops to apply the desired fusions. The tools handle the data dependency analysis, and the code transformation. Users choose from a list of automatically-detected fusion opportunities. So, in comparison to automatic, our technique enables scientists to have the flexibility to apply preferred fusions. But also, in comparison to manual fusion, scientists need to do less effort.

3 Methodology

We implemented the inlining and loop fusion procedures in the tool that translates GGDML [8] code to general purpose code. The tool runs automatic detection of inlining and fusion opportunities and shows a list to the user. When the user chooses an optimization from this list, the tool transforms the code automatically.

To evaluate the technique, we prepared a code that solves the shallow water equations [2] using the finite difference method[3]. The source code is written in the GGDML language extensions [8], which allows architecture-independent high-level code. The GGDML source-to-source translation technique [6] was used to generate and optimize the code for the different architectures and configurations.

[3] refer to `https://github.com/aimes-project/ShallowWaterEquations`

3.1 GGDML and the Code Translation

GGDML is a set of language extensions that provides performance portability for earth system modeling. Code is written with a high-level scientific abstraction of the problem as seen in Listing 1.1. A single source code can be translated into different targets by applying user-specified code schemata for different architectures. Typically, these schemata are developed by scientific programmers that understand code domain and the machine architecture. The key benefit is that these schemata and configuration files are used by many different kernels while the translation needs to be specified only once.

Listing 1.1: Example mixed GGDML and C code

```
float EDGE 2D f_U;
float EDGE 2D f_UT;
...
foreach e in grid
{
  f_U[e]=f_U[e]+f_UT[e]*dt;
}
```

This code updates the value of the X component of the velocity on the edges of the grid. It reflects the mathematical equation without optimization details.

GGDML code is translated for a specific machine based on a configuration description. Different optimization procedures, e.g memory layout transformations [7], are applied during the code translation process. Different configuration file sections guide the translation tool to apply the optimization procedures.

3.2 Inlining and Loop Fusion

The tools parse the different code files into AST structures. Inlining possibilities are checked by the tool by analysis of calls and function bodies. A call to a function, the body of which is defined even in a different code file, could be a candidiate for inlining. Close loops traversing same ranges are also analyzed for loop fusion possibilities. This analysis includes all data dependencies within loops, and possibilities to move code that resides between loops. If the loop fusion analysis is found to keep consistency of code, the fusion is listed as a candidate fusion. Inlining and fusion candidates are listed for the user to choose what to apply. According to user choice, the tool automatically uses analysis information to apply necessary transformations, including handling necessary variables, moving code around, transforming loops etc.

3.3 Code Structure and Merging

The standard code is the baseline for which we compare the performance improvements. In this modularized code, every kernel includes the necessary mathematical operations and expressions to update exactly one field. This code is easy to understand and maintain, and includes eight kernels updating: the two components of the flux: (the kernels are) $flux1$ and $flux2$, the tendencies of the two components of the velocity: *compute_u_tendency* and *compute_v_tendency*, the tendency of the surface level: *compute_h_tendency*, the two components of the velocity: *update_u* and *update_v*, and the surface level: *update_h*.

To create the merged code, the mathematical operations were remapped into three kernels such that the mathematical operations still keep the order to ensure correct computation. The merged code includes three kernels: *flux_and_tendencies*, *velocities*, and *compute_surface*.

Performance-aware users typically perform such code merging manually in the expense of readability. E.g., in a popular numerical weather prediction model, there is a single function with 2,000 LoC.

3.4 Performance Assessment

C codes with OpenMP/OpenACC were generated from the DSL representation to investigate behavior on multi-core processors, GPUs, and vector engines. The experiments are designed to understand the use of the memory bandwidth and exploiting caching/registers. To assess the performance, we derive behavioral models from the code and validate the models using monitoring tools. 'Likwid', NVIDIA's 'nvprof', and NEC's 'ftrace' tools were used on multi-core CPUs, GPUs, and vector engines respectively.

4 Evaluation

The test application solves the shallow water equations on a 2D regular grid with cyclic boundary conditions[4]. The application uses an explicit time stepping scheme in which all eight fields are updated once in each time step.

The multi-core processor experiments were run on dual socket Broadwell nodes with Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz processor. We used the Intel (ICC 17.0.5) C compiler. The GPU experiments are run on the Tesla P100 with 16 GB memory and PCIe interconnect to the host. We used the PGI (17.7.0) C compiler. The vector engine experiments were run on SX-Aurora TSUBASA vector engine using the NCC (1.3.0) C compiler.

Various more experiments have been made on other generations of GPUs and CPUs showing similar results – we selected the results conducted on the latest generation of hardware that we had access to.

4.1 Multi-Core Processors

First, we evaluate the code generated for Broadwell with different grid widths. The results before and after blocking (block size of 20000) are shown in Figure 1.

Merging the kernels results in the expected code optimization reducing the necessary memory traffic over all grid widths. Without blocking, the results of the

[4] refer to `https://github.com/aimes-project/ShallowWaterEquations`

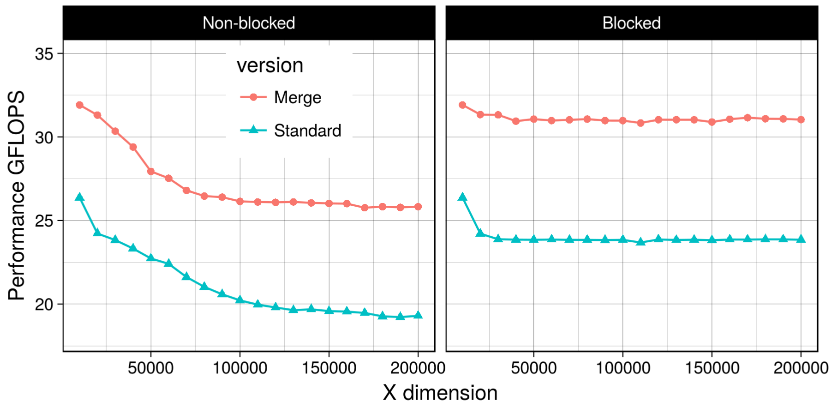
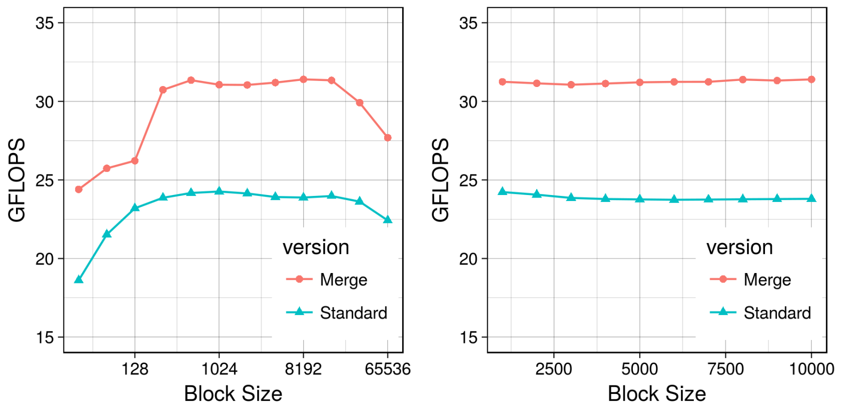


Fig. 1: Variable grid width with/o blocking on Broadwell

measurements show that the performance decreased with wider grids since the capacity of the caches is exhausted. Appropriate blocking eliminates performance loss. Given that the data are stored as single precision floating point, and that the maximum number of fields to access within a kernel is eight, the 20000 block width means the cache holds 0.61 MB per grid row. The processor has 2.5 MB L3 cache per core. Therefore, the 20000 blocking factor guarantees that more than two grid rows, and hence all the elements of the stencil (both in X and Y dimensions) are still in the L3 caches.

To better understand kernel merging and blocking relationship, we varied the block sizes. We fixed the grid width to 100k cells in the X dimension. We tested blocking with two categories of block sizes: powers of two ranging from 32 to 65536, and multiples of 1,000 from 1,000 to 10,000. Results are shown in Figure 2. Kernel merge provided performance improvement over all the tested blocking factors except very small/large factors.



(a) Power of 2 block sizes (b) Multiple of 1000 block sizes
Fig. 2: Different block sizes on Broadwell

Theoretical analysis: To understand the data movement between the cores and the main memory we instrumented the code with 'Likwid'. The measured metrics and values for the different kernels are shown in Table 1.

The kernels are bound by the memory bandwidth. Theoretical max. memory bandwidth of the Broadwell processor is 76.8 GB/s[5]. The kernels are optimized to read each variable only once from memory. For example the kernel $flux1$ accesses the memory to read two fields –reused more than once– and update one field. Multiplying the number of bytes accessed per grid cell by the grid dimensions and the time steps, this kernel needs to access 1491 GB during an application run. To compare with the measured values, if we multiply the kernel's runtime (26.86 s) by the measured memory bandwidth (61.22 GB/s), we find that the kernel accessed 1605 GB which is close to the theoretical calculations.

The achieved memory throughput of the code is close to the optimum. As long as we access the minimum amount of data in the memory with a high percentage of max memory bandwidth, the only way to optimize the code further is to decrease number of memory accesses for the application level.

In the standard code version, we need 33 accesses to the main memory for each grid cell in each time step. The arithmetic intensity of the code is 0.45 FLOP/Byte. Given the peak processor performance (2.3 GHz $\cdot$ 18 cores $\cdot$ 16 Single FP/core $\cdot$ 2) and the memory bandwidth (76.8 GB/s), the threshold arithmetic intensity to achieve the peak performance is 17.25 FLOP/Byte. The arithmetic intensity of the code is far from this threshold intensity, which explains why the achieved performance is far from the peak performance of the processor. Optimizations must increase the arithmetic intensity to increase the performance of the application.

What we gain in the merged code is reusing the values of some fields while they are still in the caches or the processor registers instead of reading them from the memory. This reduces the number of accesses to the main memory from 33 accesses to 24 accesses for each grid cell in each time step. This way, we can increase the intensity of the code to 0.63 FLOP/Byte. This is an increase by about 37% which explains the performance gain we can observe in the diagrams.

4.2 GPUs

To understand data movement between the GPU threads and the device memory, we prepared experiments for the P100 GPU. We record the performance measurements for the application with different grid widths (see Figure 3). Without blocking, the performance decreases over the tested grid widths with and without merge. However, merged code performance degrades faster after the grid width of 110k. Performance drops beyond the standard code around the grid width of 140k. This is a result of the cache limitation on the GPU as a merged kernel accesses more variables per grid cell. A kernel that accesses 8 fields on a

[5] The streaming benchmark 'stream_sp_mem_avx' from the 'Likwid' tools measured 67 GBytes/s on the processor.

Kernel	Time (s)	GFLOPS	Memory Bandwidth (GB/s)
flux1	26.9	11.2	59.8
flux2	26.6	11.3	62.8
compute_U_tendency	41.3	41.2	62.3
update_U	19.5	10.3	62.8
compute_V_tendency	46.4	36.7	61.8
update_V	19.3	10.3	63.3
compute_H_tendency	26.6	11.3	62.9
update_H	19.8	10.1	62.4
Standard_code	226.3	23.8	62.2
flux_and_tendencies	96.9	41.3	59.5
velocities	39.6	10.1	61.3
compute_surface	40.6	12.3	60.7
Merged_code	177.0	31.0	60.2

Table 1: Likwid profiles on Broadwell for all kernels and both code versions

grid that is 140k wide, where each field needs 4 bytes per cell, needs 4.27 MB, which exceeds the 4 MB L2 cache of the P100 GPU.

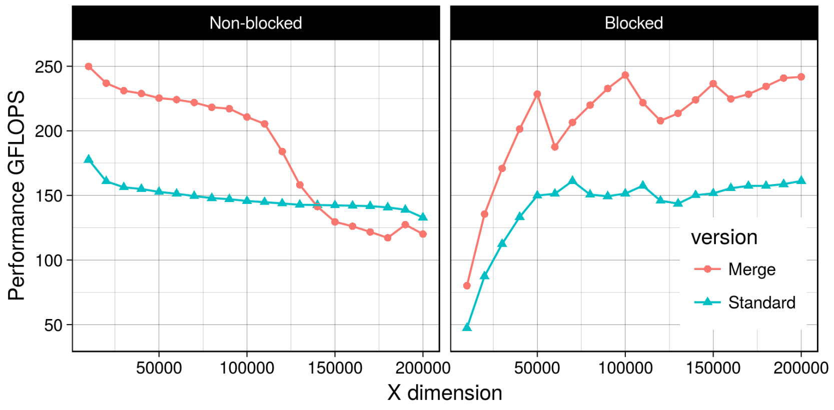
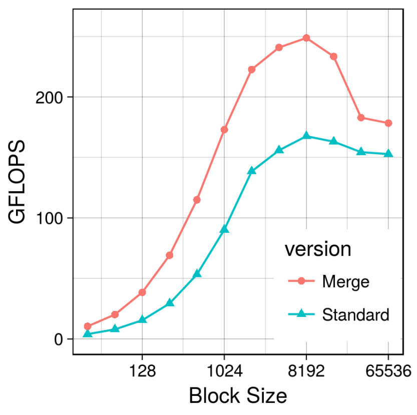


Fig. 3: Different grid widths on P100 GPU

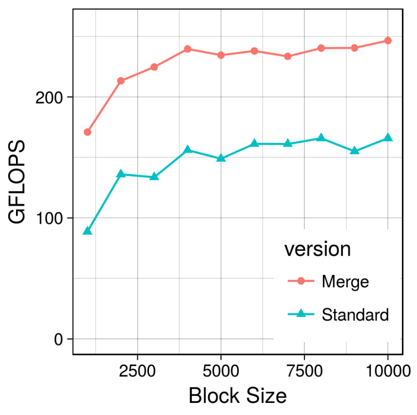
The blocking version (20k block size) does not exhibit the sharp drop over wider grids, and the merged code is better over the tested grid widths. This is a result of fitting the kernel data within the caches (remember that the 20k row in a block needs 0.61 MB for a kernel that accesses 8 fields).

To investigate further the impact of the kernel merging along with blocking, we test different block sizes again (see Figure 4). In general, kernel merging improves performance with all the tested block sizes. Optimal block sizes are around 10k. Smaller (and larger) block sizes harm the performance for both code versions.

Theoretical analysis: To gain a deeper understanding the 'nvprof' tool is used to collect different metrics. Table 2 shows the kernels measured memory throughput and accessed data volumes. Execution times and GFLOPS are also shown.



(a) Power of 2 block sizes



(b) Multiple of 1000 block sizes

Fig. 4: Different block sizes on P100 GPU

Kernel	Memory Throughput (GB/s)	Data Volume (GB)	Kernel Time (s)	GFLOPS
flux1	447	1,175	2.63	114
flux2	478	1,570	3.29	91
compute_u_tendency	358	3,338	9.33	225
update_u	376	1,126	2.99	67
compute_v_tendency	374	4,195	11.22	196
update_v	376	1,126	3.00	67
compute_h_tendency	333	1,588	4.77	105
update_h	387	1,126	2.91	69
Standard_code	380	15,244	40.13	149
flux_and_tendencies	396	5,970	15.08	325
velocities	360	2,268	6.31	63
compute_surface	403	2,303	5.71	123
Merged_code	389	10,542	27.11	221

Table 2: Kernels measurements in both code versions on P100 GPU

The measured data volumes that kernels access show data reuse at warp level. For example, the $flux1$ kernel accesses the device memory to read two fields – reused within the kernel – and updates one field. The memory access is coalesced, thus, the theoretical estimation of the data volume that the threads should access during the runtime of the kernel should be 12 bytes multiplied by the grid size and by the count of the time steps, which gives 1117 GB. In comparison, the computed value based on the 'nvprof' measurements is 1175 GB as shown in the table which is close to our expectation.

All kernels are memory bound. The measured memory throughput of the P100 on the test nodes was measured with a CUDA STREAM benchmark yielding about 498 GB/s. The memory throughput that was measured for the kernels shows high percentages (67%-96%) of the streaming memory throughput. Reducing device memory access leads to focus on the application-level optimization.

The data access is coalesced in all the kernels, before and after merging. With data reuse (coalescing means data is in cache), the standard kernels access the device memory 38 times $\cdot$ grid cells $\cdot$ time steps in total. However, the merged kernels reduce the accesses to 26. The numbers of the accesses look different from those of the Broadwell because the scheduling of the work on GPU threads is different, and hence the caching of the data is different. The access reduction explains the performance improvement between the two code versions (221 GFLOPS : 149 GFLOPS) as the arithmetic intensity is shifted from 0.39 to 0.58 through merging.

4.3 Vector Engines

On Aurora vector engine, we vary the grid width from 10k to 100k and measure the performance (see Figure 5). Merging improved performance over all the grid widths. Performance is not dropping without blocking (at least at the chosen grid widths).

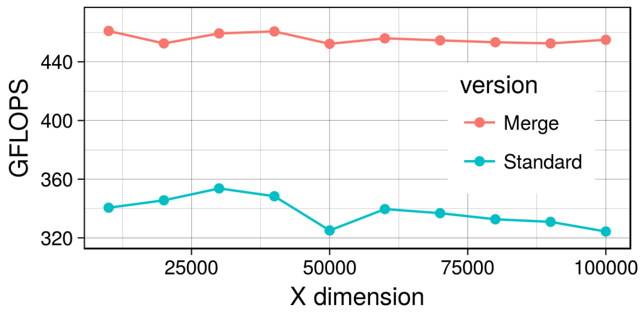


Fig. 5: Different grid widths on NEC Aurora vector engine

To understand the performance NEC's 'ftrace' tool is used (see Table 3). The theoretical memory bandwidth of the vector engine is 1.2 TB/s. Based on the 'ftrace' measurements, the computed values of the memory throughput show that all the kernels run with a high percentage of the memory bandwidth (80%) before and after the kernel merging.

The performance ratio before and after the kernel merging is 453 GFLOPS : 322 GFLOPS. This result is roughly the ratio of the arithmetic intensities which we discussed in the multi-core processor results (0.63 : 0.45).

5 Summary

With manual fusion, a 2k LOC function represents a challenge for scientists to find and test optimal fusions, while automatic fusion does not allow this flexibility. In this work, we presented a technique to replace manual and automatic loop fusion with a new genuine alternative. Code is automatically analyzed for fusion opportunities, and function inlining is also detected across source files. A list of possibilities is given to the user. Based on the user preferences, code transformation is applied based on the inlining/fusion that the user chooses.

GGDML was used to develop high-level code that can be translated into different architectures. This shallow water equations solver was then translated and executed on multi-core processors, GPUs, and vector engines.

Kernel	Time (s)	GFLOPS	Memory Throughput (GB/s)
flux1	1.30	230	858
flux2	1.51	199	989
compute_U_tendency	5.29	359	986
update_U	1.21	166	927
compute_V_tendency	5.22	384	1,001
update_V	1.21	165	924
compute_H_tendency	1.52	330	984
update_H	1.20	167	934
Standard_code	18.63	322	961
flux_and_tendencies	8.40	500	911
velocities	2.43	165	922
compute_surface	2.31	303	940
Merged_code	13.25	453	911

Table 3: Kernel measurements of both code versions on the NEC Aurora

The results show the success of the technique to improve the efficiency of the use of the memory bandwidth on the different architectures. Scientists can apply the fusions (even across source files) and test any set of loop fusions as they prefer. As a future work, we plan to explore exploiting temporal locality between timesteps using the semantics of GGDML, and to explore using machine learning to recommend fusion sequences.

6 Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 Software for Exascale Computing SPPEXA (GZ: LU 1353/11-1). We also thank the 'Regionales Rechenzentrum Erlangen' (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), the Swiss National Supercomputing Center (CSCS), and NEC Deutschland, who provided access to their machines to run the experiments. We also thank Prof. John Thuburn – University of Exeter, for his help to develop the code of the shallow water equations.

References

1. CSCS GridTools. `https://pasc17.pasc-conference.org/fileadmin/user_upload/pasc17/program/post144s2.pdf`. Accessed: 2017-12-22.
2. Vincenzo Casulli. Semi-implicit finite difference methods for the two-dimensional shallow water equations. *Journal of Computational Physics*, 86(1):56–74, 1990.
3. Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing cuda code by kernel fusion: application on blas. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.
4. Jan Fousek, Jiři Filipovič, and Matuš Madzin. Automatic fusions of cuda-gpu kernels for parallel map. *ACM SIGARCH Computer Architecture News*, 39(4):98–99, 2011.

5. Antoine Fraboulet, Karen Kodary, and Anne Mignotte. Loop fusion for memory space optimization. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 95–100. ACM, 2001.
6. Nabeeh Jum'ah and Julian Kunkel. Performance portability of earth system models with user-controlled ggdml code translation. In Rio Yokota, Michèle Weiland, John Shalf, and Sadaf Alam, editors, *High Performance Computing*, pages 693–710, Cham, 2018. Springer International Publishing.
7. Nabeeh Jumah and Julian Kunkel. Automatic vectorization of stencil codes with the ggdml language extensions. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'19, pages 2:1–2:7, New York, NY, USA, 2019. ACM.
8. Nabeeh Jumah, Julian M Kunkel, Günther Zängl, Hisashi Yashiro, Thomas Dubos, and Thomas Meurdesoif. Ggdml: icosahedral models language extensions. *Journal of Computer Science Technology Updates*, 4(1):1–10, 2017.
9. Ken Kennedy and Kathryn S McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer, 1993.
10. Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.
11. Jiayuan Meng, Vitali A Morozov, Venkatram Vishwanath, and Kalyan Kumaran. Dataflow-driven gpu performance projection for multi-kernel transformations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 82. IEEE Computer Society Press, 2012.
12. Michel Müller and Takayuki Aoki. Hybrid fortran: High productivity gpu porting framework applied to japanese weather prediction model. *arXiv preprint arXiv:1710.08616*, 2017.
13. Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE Press, 2014.
14. Mohamed Wahib and Naoya Maruyama. Automated gpu kernel transformations in large-scale production stencil applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 259–270. ACM, 2015.
15. Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344–350. IEEE, 2010.
16. Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.
17. Haicheng Wu, Srihari Cadambi, and Srimat T Chakradhar. Optimizing data warehousing applications for gpus using dynamic stream scheduling and dispatch of fused and split kernels, March 24 2015. US Patent 8,990,827.
18. Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2433–2442. IEEE, 2012.