A Generic Strategy for Multi-stage Stencils

Mauro Bianco and Benjamin Cumming

Swiss National Supercomputing Centre (CSCS)

Abstract. Stencil computations on regular grids are widely used in scientific simulations. Optimization techniques for such stencil computations typically exploit temporal locality across time steps. More complex stencil applications, like those in meteorology and seismic simulations, cannot easily take advantage of these techniques, since the number of physical fields and computation stages to consider at each time step flush all data present in the cache at the beginning of the next time step. In this paper we present a technique for improving performance of such computations, based only on spatial tiling, which is implemented as a generic algorithm.

More specifically, we investigate how to take advantage of producer-consumer relations of stencil loops, in a single time step, to improve memory hierarchy utilization. This approach makes it possible to balance computation and communication to improve resource usage. We implement our methods using generic programming constructs of C++, which we compare with hand-tuned implementations of the stencils. The results show that this technique can improve both single-threaded and multi-threaded performance to closely match that of hand-tuned implementations, with the convenience of a high-level specification.

1 Introduction

Stencil computations are an important algorithmic motif in scientific computing. When applied on regular grids, stencil computation is essentially a set of nested for loops in which the body of the innermost loop computes a function using grid values at fixed offsets from the coordinates specified by the loop variables. Stencil computations are often used in the solution of (partial) differential equations with explicit temporal integration. Such applications use a time loop which applies the same stencils on each iteration. Scientific simulations often employ 3D stencils because they map better to real world cases. From an algorithmic point of view, the 3D stencil computations typically used in scientific computing pose specific challenges when optimizing the use of memory hierarchy [10]. The literature focusing on this kind of algorithms is abundant, and very clever techniques have been developed to improve their performance.

Many of these optimization strategies take advantage of the fact that, for simple differential equations at least, temporal locality may be exploited across time steps (among others, [4,9,12,6,13]). However, for more complicated applications, such as meteorological simulations, the number of stencil functions applied in each time step is very high. In such applications, it is not possible, or very hard,

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 584–595, 2014.

```
for(i=bi−b10; i<ei+e10; ++i)
  for(j=bj−b12; j<ej+e11; ++j)
    for(k=bk−b12; k<ek+e12; ++k)
      t[i][j][k] = in[i+2][j][k]...

for(i=bi; i<ei; ++i)
  for(j=bj; j<ej; ++j)
    for(k=bk; k<ek; ++k)
      out[i][j][k] = t[i−1][j][k−1]...
```

(a)

```
struct stencil_operator {
  template <class S1, class S2>
  void
  operator()(S1 &v, S2 const& u) const
  {
      v() = 1.0/36.0 ∗(6∗u()
            − u(1,0,0) − u(−1,0,0)
            − u(0,1,0) − u(0,−1,0)
            − u(0,0,1) − u(0,0,−1));
  } };
```

(b)

Fig. 1. (a) Example of a typical structure of a stencil application. (b) Example of a GSCL stencil operator implementation for a 7-point Laplacian.

to retain data in cache between time steps, due to the complicated dependencies between the different stencil stages in each time step.

However, it is possible to exploit temporal locality in such applications. Here we investigate temporal locality in the producer-consumer relations between consecutive stencil loops. A trivial example of this is illustrated in Figure 1.a, where the output of the first stencil loops, stored in **t** is used in the second loop.

In this paper, we refer to a nested loop as a *stencil stage*, and a sequence of such stages as a *multi-stage stencil computation*. An example stencil from the numerical weather forecasting code COSMO (Consortium for Small-scale Modeling) [5] is the horizontal diffusion (HD) stencil, which applies a fourth-order dispersion operator to an input field, and writes the result to an output field. The horizontal diffusion stencil can be expressed using four stages: the first computes the Laplacian of the input field, then two independent stages that use the Laplacian values to compute orthogonal fluxes in the horizontal plane, and a final stage that combines the fluxes to compute the output.

In our work we distinguish between data that is the output of a multi-stage stencil, and temporary data that is produced and consumed in intermediate stages of a multi-stage stencil. For instance, the Laplacian and fluxes in the horizontal diffusion stencil are only consumed by the stages inside the stencil, so they can be discarded after the output has be computed. For such data fields that are only consumed in the stencil where they are computed, we take advantage of blocking techniques that allow us to balance computation intensity and memory bandwidth. We call the stencil stages that produce intermediate output *intermediate stages*, and in this paper we focus on multi-stage stencil computations with at least one intermediate stage.

We will describe an algorithm that each thread in a shared memory parallel program, uses to trade computation intensity and communication bandwidth of multi-stage stencils, and show how this improves both the performance and scalability of the multi-stage stencil computation. We present our solution implemented in context of the Generic Stencil Computing Library (GSCL) [2], which is a generic C++ library for specifying stencil computations on different architectures, including multicore processors, graphic accelerators, up to large parallel

machines. The library expresses computations as *stencil operators* applied to some data fields according to *iteration spaces*, which specify data dependencies. An example of a stencil operator can be found in Figure 1.b. We show that our technique is capable of matching the performance of hand-tuned versions, while retaining ease of use through the generic interface of GSCL. Being generic, our solution also is suitable for being implemented in other contexts, such as a DSL.

2 Related Work

Stencil computations are typically bandwidth limited. As such, much of the research in this field has focused on optimizations that reduce global memory bandwidth by taking advantage of cache hierarchies. In [4,9,12] the optimization of simple stencil computations inside time loops was investigated. Similar stencils were studied in [6,13] in the context of cache obliviousness. The most complicated stencil computation investigated in the aforementioned papers is Jacobi iteration with two grids, while real applications like climate modeling or earthquake simulations [11,5] have between 5 to 15 grids. It is not clear to what extent the reasoning behind many optimizations for simpler stencils might be directly ported to real-world use cases. A similar problem is addressed in [10], in which, however, the Authors discuss solutions for other cases.

In this paper we use GSCL which is a C++ template library, as opposed to [3], which employes a DSL, and [8] which heavily uses macros. This gives us the ability to reach good expressivity in design and a robust implementation. In [7,3] Authors use auto-tuning techniques, which we are planning to investigate in future papers, but are not the aim of current work. More specific compiler approaches, such as [1], are also out of our scope, but try to solve similar problems.

3 Tiling and Buffering

In this section we describe how each thread in a multithread stencil execution can optimize the trade-off between computation intensity and memory bandwidth in order to improve execution time.

These computations are specified in GSCL by using *functional stencils*, or **f_stencil**s. Functional stencils specify that the value needed at the point of evaluation depends on a stencil operator computed at a given offset, specified through relative coordinates. The syntax is as follows: **f_stencil <F, i,j,k>(a,b,...)** where **i**, **j** and **k** are coordinates relative to the current point of evaluation where **F** is to be computed, and **(a,b,...)** is the list of arguments to be passed to **F**.

Figure 2 shows an implementation of the stencil operator **simpleHD** that computes a simplified version of the Horizontal Diffusion (HD) stencil from the COSMO weather forecasting code. Computing HD first requires the computation of a Laplacian (**lap**), then two fluxes (**fluxx** and **fluxy**). Finally, the **simpleHD** computes the output values. The figure also illustrates the call graph.

Functional stencils employ a functional specification of the stencil operation instead of the imperative approach illustrated in Figure 1.a. Loop bounds are

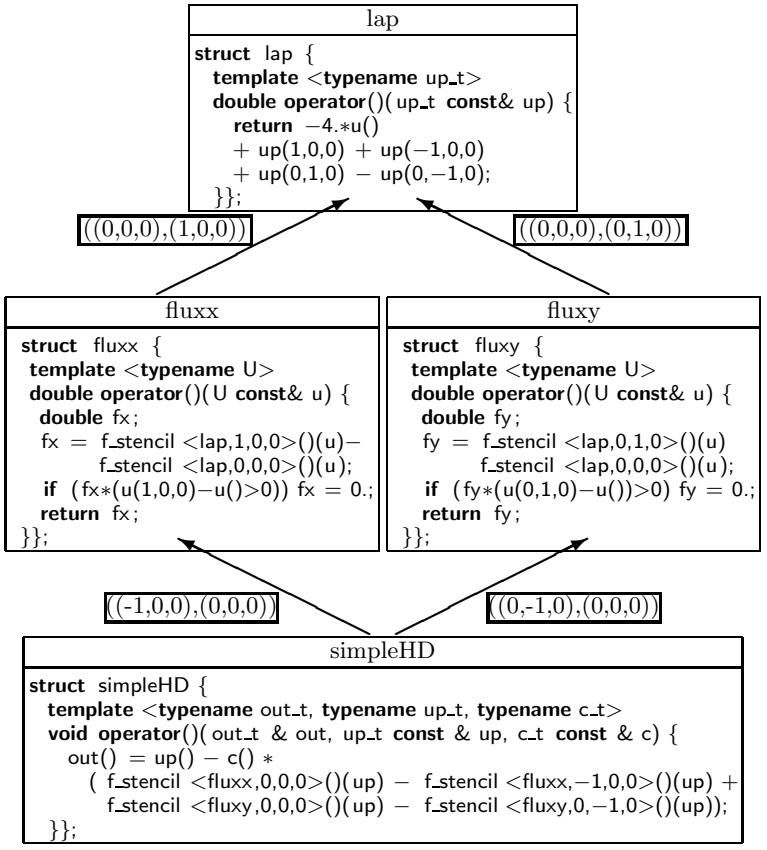

Fig. 2. The illustration of the GSCL implementation of the HD multi-stage stencil

determined automatically from the output dimensions, removing the burden of manually adjusting the loop limits **b10**, **b11**, **e10**, etc.. When executed as functions, functional stencils reduce memory transactions in memory-bound stencil computations by computing intermediate fields on the fly, at the expense of increasing the number of arithmetic operations. Hence, as the depth and arity of the call tree increases, the use of functional stencils can make the computation compute-bound with possible poor performance. However, the functional flavor of the algorithm specification allows the implementation to choose to buffer intermediate results in order to trade memory pressure with computation redundancy, thus transforming calls to functional stencils into *reads* from pre-computed buffers.

3.1 The Algorithm

The algorithm for executing a multi-stage stencil has two phases. The first is an analysis phase that computes the sizes of the intermediate buffers, the second is the computation of the stencil itself.

Algorithm: Analysis
Input: Sets E_u, set V of operators
Output: Graph $G = (V, E, \phi)$
$E \leftarrow \emptyset$;
for $u, v \in V$ **do**
  $\phi_{uv} \leftarrow \text{MBR}\{p : (u, v, p) \in E_u\}$;
  Insert (u, v) in E;
end
for $v \in V$ **do**
  $\Phi_v \leftarrow$ Empty rectangle;
end
$s \leftarrow$ Source of G;
Update_$\Phi(s, G)$;

Algorithm: Update_Φ
Input: Current node u, graph G
Output: Updated Φ_v values
for v *such that* $(u, v) \in E$ **do**
  $\Phi_v \leftarrow \text{MBR}(\Phi_v, \Phi_u \oplus \phi_{uv})$;
  Update_$\Phi(v, G)$;
end

Algorithm 1. Algorithms for determining the buffer sizes of the nodes of the call graph

We start with a set V of stencil operators that call other operators as functional stencils. For each stencil operator $u \in V$ we have the set $E_u = \{(v, p)\}$, where v is a stencil operator called as a functional stencil by u at position p relative to current evaluation point of u, where p is a tuple of (integer) relative coordinates.

We can obtain a weighted direct graph $G = (V, E, \phi)$ such that $E = \{(u, v) : (v, p) \in E_u$ for some $p\}$, and the weight ϕ_{uv} represent the minimum bounding rectangle (MBR) of the points in $\{p : (v, p) \in E_u\}$, the set of all the offsets at which u calls v. A rectangle is a pair of tuples with minimum and maximum coordinates among the points in the set. We indicate a rectangle as (p^b, p^t) to indicate the coordinates of the "bottom" (typically with non positive coordinates) and the "top" corners (typically with non negative coordinates) of the rectangle. The structure of the graph G for the HD stencil is shown in Figure 2 along with the weights. We are interested in computations where graph G is acyclic (a DAG) (an operator cannot call a predecessor in the graph) and with a single source node that we indicate as s.

After having obtained the DAG G, we need to compute the extent (a rectangle) at which each of the nodes $v \in V$ is needed during the computation and we call it Φ_v. To do this, we first set $\Phi_s = (\mathbf{0}, \mathbf{0})$ as an empty rectangle, centered at the origin, representing the point of evaluation. Next we traverse the graph in pre-order. When node u is visited we update the values Φ_v, of nodes v adjacent to u, to the MBR including the rectangles Φ_v and $\Phi_u \oplus \phi_{uv}$, where the sum for two rectangles (p^b, p^t) and (q^b, q^t) is defined as $(p^b, p^t) \oplus (q^b, q^t) = (p^b + q^b, p^t + q^t)$. When Φ_v is updated for the first time, it is set to $\Phi_u \oplus \phi_{uv}$, which are defined since G is traversed in pre-order. Algorithm 1 shows the pseudo-code for this procedure.

Proof: We now offer an informal proof that the node v is never invoked outside of the bounding box Φ_v computed using Algorithm 1. If we assume that all the ϕ_{uv} are correct, then if v was needed at a coordinate outside Φ_v, a predecessor u

of v would have to be accessed outside of Φ_u. Likewise, if u is not the source we can apply the same reasoning backward. When we reach the source it means that it is accessed outside the point of evaluation, which is against the hypotheses. We can also see that if there are no conditional branches that can falsify this statement in particular cases, the *edges* of the rectangles are always accessed, so the bounding is tight.

There may be values in the rectangles that are not needed. However, if the call tree is wide enough, the additional storage overheads are compensated for by the use of simple affine expressions to access data. The final objective is to tile the computation with blocks of size $B_I \times B_J \times B_K$. Before doing so, given $\Phi_v = (p^b, p^t)$, we associate a buffer with each stencil operator in G, where the buffer b_u has dimension $(B_I - p_i^b + p_i^t) \times (B_J - p_j^b + p_j^t) \times (B_K - p_k^b + p_k^t)$ with origin set to $-p_b$ so that accesses to the halo region are valid.

To balance the computation/communication ratio, nodes in the DAG can be marked be either buffered or to computed on the fly. Execution of the stencil then proceeds as a post order visit on the DAG G, so first the adjacency list of a node is evaluated and then the node itself. If a node is marked to compute values in a buffer, it is executed, otherwise nothing is done, and the node will be invoked as a functional stencil in subsequent stages.

3.2 Implementation

The algorithm described in the previous subsection is modified for implementation in GSCL because some of the required information is computed during compilation. Since C++ does not allow introspection, the structure of the unweighted version of the DAG G has to be provided to GSCL in the form of a *call graph* object. The object type encodes the topology of the graph as a list of levels corresponding to the topological sort of the DAG to guarantee the producer consumer relations of the computation. The interface requires the first level to be a **procedure**, i.e. it behaves as a regular GSCL stencil operator that writes the results into some of the output arguments. The other levels can either be **function**s, or a list of functions that are **independent**, all of which will be called as functional stencils. Additionally, since GSCL cannot know which arguments will be passed to the functional stencils, an argument mapping is also needed. For the simple HD stencil in Figure 2, the call-graph type is

```
typedef call_graph_type <procedure<simpleHD>,
                         independent<function<fluxx,arg_map<1> >,
                                     function<fluxy,arg_map<1> > >,
                         function<lap,arg_map<1> > > cg_type;
```

We would like to emphasize that, although our implementation uses C++, the technique is more general. For instance a specialized compiler could collect the information about the call graph from the code without user intervention.

To execute the multi-stage stencil, an object of type **call_graph_type**, which is the implementation of call graph object mentioned before, can then be passed to a **do_all_ms**, i.e., a special iteration space that process call graph types, where the suffix **ms** stands for multi-stage. After some transformations to adapt

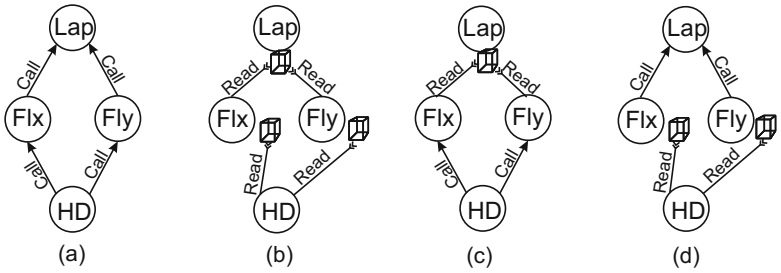

Fig. 3. Graphical representation of the different implementations that GSCL provides for the HD stencil. Circles are procedure/function nodes, while rectangles are node buffers.

the call graph, **do_all_ms** computes the rectangles ϕ and Φ for determining the sizes of the blocks by simulating the computation passing to the operators **test_stencil**s to collect the proper information.

To mark nodes to compute on the fly and other to buffer, two compile time constant parameters are given as **do_all_ms<low, high>(...)**, where **low** and **high** indicates where to turn buffering on and off, respectively. This mechanism allows the programmer to specify *different thresholds for different computations in the same program*, and incurs no runtime overhead by virtue of being performed at compile time. Figure 3 shows different implementations for the horizontal diffusion (HD) stencil. If the thresholds for turning on and off the buffering define an empty interval, the implementation turns all of the functional stencils into function calls to compute values on the fly (Figure 3.a). If the thresholds include all levels then all functional stencil calls read results from previously computed blocks (Figure 3.b). We can specify that levels 1 and 2 are to be buffered, and have the fluxes computed on the fly as in Figure 3.c, or that levels 0 and 1 are to be buffered, and get the behavior shown in Figure 3.d.

At this point the execution of the multi-stage by stencil computation is performed through a post-order visit of the call-graph, which is inlined and has virtually no runtime overhead. The OpenMP implementation of GSCL, first partitions the global iteration space, then each thread applies the multi-stage stencil on its partition.

3.3 Analysis

The versions of **simpleHD** obtainable by GSCL, and depicted in Figure 3, plus the base version that does not use loop tiling, have different computation to main memory access ratios. By analyzing the code in Figure 2, it is not difficult to see that the a base version, that does not use **f_stencil**s, perform 18 operations per output value. For the tiling, in this example the operation count does not depend on B_k (no *halo* in the third dimension). By picking $B_i = B_j = 8$, we obtain block sizes that provide good cache usage. In this case the version that computes everything on the fly, corresponding to Figure 3.a, needs 61 operations

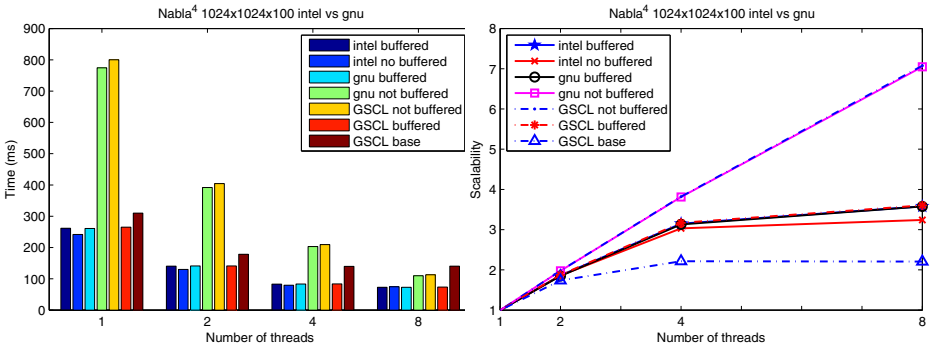

Fig. 4. Comparison of execution times and scalability of hand-coded implementations of ∇^4 (Intel C++ and GNU C++ compilers) and GSCL (only GNU)

per output value. When we buffer all the intermediate results, Figure 3.b, we obtain 21.8 operations per output value, which is higher that the base version since the computation on the tiles is redundant. The cases of Figure 3.c and Figure 3.d have respectively 28.8 and 36.5 operations per output value.

Assuming that the block sizes are small enough to keep all the intermediate storage in cache, the characteristics of memory accesses of the GSCL implementations, other than the base, are similar thanks to the loop tiling. Counting the number of operations is then enough to give an indication of the ratio between computation and main memory requests, whose rate is limited by the physical bandwidth. We would expect the versions that buffer only the Laplacian to perform better than the version the buffer the fluxes, since the computation intensity is quite high for the latter version and because two buffers have to be kept in cache instead of one, thus increasing the cache pressure. In general the performance of the actual computation depends on the *arity* and depth of the call three and it not easy to predict a priori which implementation is the best.

For the base version we draw a somewhat different conclusion. As the problem size increases we should see a gap between this and the tiled versions as the intermediate fields for the base version become to large to reside in cache.

4 Results

In this Section we show and discuss some performance results obtained with our algorithm. We test on the cores of a single socket in this paper to avoid NUMA effects, and with the understanding that GSCL typically has one MPI process per NUMA domain. Testing was performed on an eight-core Intel Sandybridge processor (Xeon E5-2670) running at 2.60GHz, without hyper-threading. Each core of the chip has 32KB of L1 cache, with all 8 cores sharing 256KB of L2 on-chip cache and 20MB of L3 off-chip cache. Version 4.7.1 of the GNU C++ compiler and version 13.0.1 of the Intel C++ compiler were used. The benchmark

code is designed to run a test multiple times and to flush the caches at the beginning of each iteration, so as to test the hypothesis that no data is held in cache at the beginning the iterations. To obtain stable measurements we show the minimum execution times of several iterations. However, we note that the execution times exhibit little noise on average. We also instrumented the code with PAPI counters to measure vectorization and cache behaviors.

4.1 Fourth-Order Dispersion

We first show the results for a fourth-order dispersion operator ∇^4, also referred to as *nabla*^{4}, which can be implemented by twice applying a Laplacian operator to an input field. The base version uses two separate **do_alls** and an explicit temporary storage area.

To validate performance, we implemented hand-tuned versions of compute-on-the-fly and buffered implementations. They were developed in a distinct source files, since the modularity is reduced in these versions. It should be noted that the hand-tuned versions are not generic at all and their code is much longer than GSCL code. It is made of several loop nests (for tiling and iterating within blocks), plus pointer arithmetic, and specific **#pragmas** for the compiler[1].

In Figure 4.a we show the execution times of ∇^4 for different number of threads (one per core on the chip) and a fairly large input size (in the context of COSMO). The versions not labelled with "GSCL" are hand-tuned versions. The GNU compiler did not perform well when no buffering is employed in GSCL, that is, when we compute all values needed by ∇^4 on the fly. In this case the GNU compiler is unable to exploit vectorization, and because this version is the most computationally intense, the penalty for not using vector instruction is the highest. On the other hand this results in almost ideal scalability. As we can see, the base implementation scales poorly due to bandwidth memory limitations do to lack of loop-tiling (Figure 4.b).

For this input size, GSCL performs comparably to hand tuned versions. On smaller inputs GSCL is slightly slower but still competitive (results not showed for space constraints). The compute-on-the-fly hand-tuned version is quite fast compared to the corresponding GNU compiled version. This is because unlike the GNU compiler, the Intel compiler can vectorize this computation very well, which is important for this computationally intensive case.

4.2 SimpleHD

In this section we discuss the performance of the **simpleHD** example we analyzed throughout the paper. **simpleHD** implementation corresponds to Figure 2 which allows us to test with turning on and off buffering. First, we show how performance varies as we tune the levels for turning buffering on and off. Figure 5 shows a comparison of the base GSCL version against the four different

[1] For a fair comparison of compiler generated code, explicit prefetching and intrinsics were not used in the hand-tuned codes.

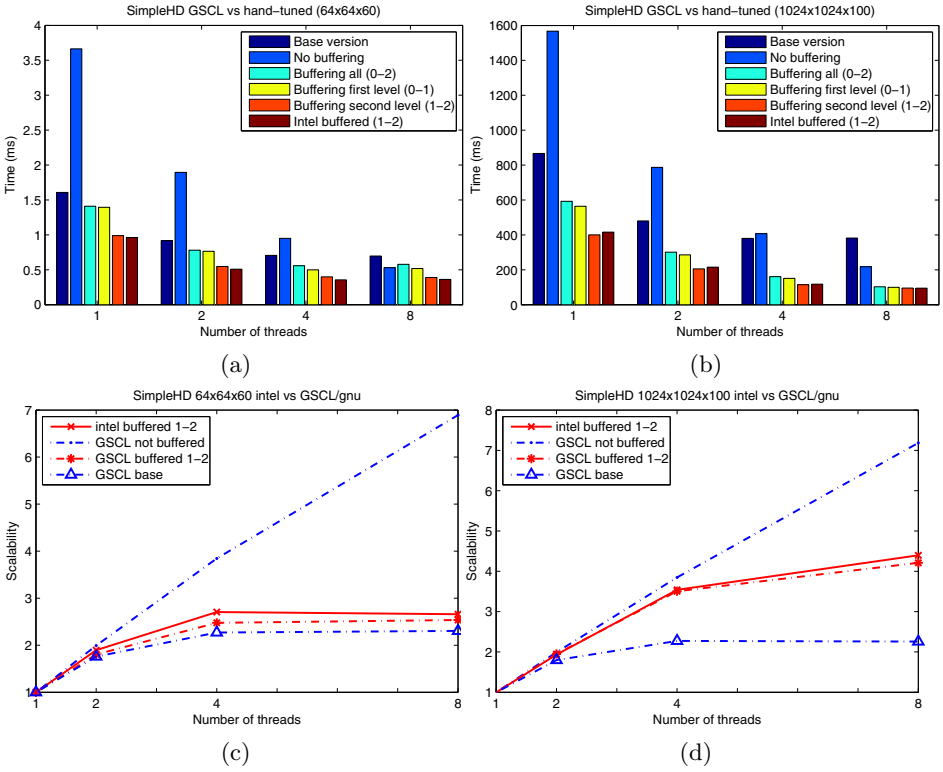

Fig. 5. Comparison of hand-tuned implementation and different thresholds to trigger buffering on and off in the simpleHD example. The versions are indicated by the thresholds used to turn on buffering. (a) and (b) shows times, (c) and (d) scaling.

combinations shown in Figure 3. When we turned off buffering at all levels the performance was low on a single thread. This is mostly due to the inability of the GNU compiler to employ vectorization in this compute bound case, as can be seem in Table 1 entry for "GSCL-none", which is the version that does not employ buffering. As the number of threads and problem size increases (Figure 5.a and .b) the performance increases, getting better than the base version, due to better memory use. This observation is also evident in Figure 5.c and .d, which shows scaling with thread count for the implementations.

As expected from the analysis in Section 3.3, the performance of buffering all the nodes (threshold from 0 to 2, dubbed 0-2) was similar to buffering the computation of the fluxes while computing the Laplacian on the fly (threshold from 0 to 1, so dubbed 0-1). Buffering the Laplacian and computing the fluxes on the fly (dubbed 1-2) gave the best performance for both the small and large meshes. As the number of threads increased, the advantage of on the fly computation increased since it is less bandwidth-eager. The execution time for the

Table 1. Floating point operations issued and cache misses for different number of for each implementation of the simpleHD stencil, 8 threads. The total is the total operations amount of scalar floating point operations performed in each case. The last column is the ratio between accessed to main memory with those that fail in L1 cache.

Version	Scalar	SSE-128	AVX-256	Tot. ops	L1 misses	L3 misses	L1/L3 ratio
GSCL-Base	91	0	505	2111	212.174	44.97	0.212
GSCL-none	5455	0	0	5455	106.252	2.012	0.0189
GSCL-0-2	0	0	654	2616	250.056	7.203	0.0288
GSCL-0-1	0	0	1134	4536	153.379	4.235	0.0276
GSCL-1-2	0	0	785	3154	190.338	3.789	0.0199
Intel-1-2	0	426	575	3152	180.408	5.771	0.0319

base version on one thread is $8.63 \times 10^{-1}s$, while the "1-2" version is $4 \times 10^{-1}s$, which is 2.15 times faster. On 8 threads the ratio is $3.8 \times 10^{-1}/9.5 \times 10^{-2} \simeq 4$.

In Figure 5 we also compare the performance of GSCL implementation with Intel compiled hand-tuned version (Intel buffered 1-2). The performance of these two equivalent algorithms are very similar, even though the source codes are completely different. This can also be seen clearly in Table 1 which shows that the operation counts for "Intel-1-2" and "GSCL-1-2" are in the range of the noise of the performance counters, despite the two implementations having a different mixture of 128 bit and 256 bit vector instructions. This indicates that the GSCL implementation does not loose performance by employing sophisticated generic programming techniques, thus ensuring that cost of abstraction in GSCL is negligible. In the same table it can be noted that, while the base implementation has very poor cache performance, since 21% of the accesses that fails to L1 reach main memory. The other implementations shows ratios of 2-3%. It is interesting to note that GSCL-none has the best ratio, since there the computation does not use buffers at all, while GSCL-1-2 exhibits the best ration between cache accesses and operation counts, which explains why Intel-1-2, even though with a similar operation count, is slower than the GSCL one.

5 Conclusions

We presented a generic method to optimize complex stencil applications by expressing the computation using a functional approach to fuse otherwise distinct loops, and buffer intermediate results for best memory hierarchy exploitation. The implementation can be tuned by selecting for which levels to use buffering and for which to compute on the fly, thus trading computation for memory bandwidth. We shown that we can achieve the performance of hand tuned implementations of the same computations.

In future we also plan to employ auto-tuning techniques to select tile sizes and to select which nodes to buffer. For the latter case it is possible to work at the finer granularity of single nodes instead of levels.

References

1. Bandishti, V., Pananilath, I., Bondhugula, U.: Tiling stencil computations to maximize parallelism. In: Proc. of the 2012 ACM/IEEE Conference on Supercomputing, SC 2012, pp. 40:1–40:11. IEEE Computer Society Press, Los Alamitos (2012)
2. Bianco, M., Varetto, U.: A generic library for stencil computations. CoRR, abs/1207.1746 (2012)
3. Christen, M., Schenk, O., Cui, Y.: Patus for convenient high-performance stencils: Evaluation in earthquake simulations. In: SC, p. 11 (2012)
4. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Rev. 51, 129–159 (2009)
5. Doms, G., Schätter, U.: A description of the nonhydrostatic regional model lm, part i, dynamics and numerics (2002)
6. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proc. of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 361–366. ACM, New York (2005)
7. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An auto-tuning framework for parallel multicore stencil computations. In: IPDPS, IPPS 2010, pp. 1–12 (2010)
8. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In: Proc. of 2011 ACM/IEEE Conference on Supercomputing, SC 2011, pp. 11:1–11:12. ACM, New York (2011)
9. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In: Proc. of the 2010 ACM/IEEE Conference on Supercomputing, SC 2010, pp. 1–13. IEEE Computer Society, Washington, DC (2010)
10. Rivera, G., Tseng, C.-W.: Tiling optimizations for 3D scientific computations. In: Proc. of the 2000 ACM/IEEE Conference on Supercomputing, SC 2000. IEEE Computer Society, Washington, DC (2000)
11. Rojas, O., Dunham, E.M., Day, S.M., Dalguer, L.A., Castillo, J.E.: Finite difference modelling of rupture propagation with strong velocity-weakening friction. Geophysical Journal International 179(3), 1831–1858 (2009)
12. Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., Matsuoka, S.: Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In: Proc. of 2011 ACM/IEEE Conference on Supercomputing, SC 2011, pp. 3:1–3:11. ACM, New York (2011)
13. Strzodka, R., Shaheen, M., Pajak, D., Seidel, H.-P.: Cache oblivious parallelograms in iterative stencil computations. In: Proc. of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 49–59. ACM, New York (2010)