# A Multi-level Optimization Strategy to Improve the Performance of Stencil Computation

Gauthier Sornet, Fabrice Dupros, Sylvain Jubertie

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 1083–1092

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017,
Zurich, Switzerland

# A Multi-level Optimization Strategy to Improve the Performance of Stencil Computation

Gauthier Sornet[1][2], Fabrice Dupros[2], and Sylvain Jubertie[1]

[1] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France.
{gauthier.sornet, sylvain.jubertie}@univ-orleans.fr
[2] BRGM, BP 6009, 45060 Orléans Cedex 2, France.
f.dupros@brgm.fr

**Abstract**

Stencil computation represents an important numerical kernel in scientific computing. Leveraging multi-core or many-core parallelism to optimize such operations represents a major challenge due to both the bandwidth demand and the low arithmetic intensity. The situation is worsened by the complexity of current architectures and the potential impact of various mechanisms (cache memory, vectorization, compilation). In this paper, we describe a multi-level optimization strategy that combines manual vectorization, space tiling and stencil composition. A major effort of this study is to compare our results with the Pochoir framework. We evaluate our methodology with a set of three different compilers (Intel, Clang and GCC) on two recent generations of Intel multi-core platforms. Our results show a good match with the theoretical performance models (i.e. roofline models). We also outperform Pochoir performance by a factor of x2.5 in the best case.

*Keywords:* Stencil computation, Vectorization, Performance model

## 1 Introduction

Stencil computation is widely used in numerical simulations. For instance, many physical models based on PDE (Partial Differential Equation) heavily rely on it. This is coming from the grid-based computation induced by finite-difference or finite-volume methods that are routinely used to solve various equations (elastodynamics, heat equation, shallow water equation, etc). This numerical kernel can represent a major part of the total elapsed time, particularly when an explicit method is implemented. In this case, the numerical kernel should be evaluated hundreds or thousands of times making critical any performance improvement.

Regarding the computation workflow, each numerical stencil could be described as a set of weighted neighbor cells that should be loaded and combined. The stencil morphology (i.e. the number of cells and the directions involved in the computation) has an impact on the overall performance of the kernel. For instance, the ratio between the number of memory accesses

10.1016/j.procs.2017.05.217

and the number of operations on each cell should be minimized to improve the computational efficiency. This memory-bound situation is a concern on multi-core architectures regarding the limited improvement at the bandwidth level. The situation is worsened on many-core platforms and recent work underline significant efforts to extract the best level of performance of such platforms (e.g. Kalray MPPA-256 chips for seismic wave propagation stencil in [3]).

In this paper we consider the classical 7-point and 27-point stencils that exhibit different characteristics (arithmetic intensity, memory bandwidth demand). These stencils have already been widely studied in the scientific literature providing some good references [5]. The main contributions of this paper are the following. First, we underline the effectiveness of each level of optimization (vectorization, tiling, stencil composition) based on a set of representative compilers (Clang, Intel and GCC). Moreover, we show the portability of our methodology on two recent dual-socket Intel platforms (Intel Ivy Bridge and Broadwell). Finally, our contributions are analyzed with respect to the expected peak performance of each numerical kernel learned from the roofline prediction model [16]. Stencils studied in this paper have also been implemented with the Pochoir reference compiler [15] in order to discuss our results.

The paper proceeds as follows. Section 2 describes the related work. Section 3 discusses the fundamentals of stencil computation. We motivate the choice of each kernel by underlying their theoretical behavior. Section 4 underlines the main challenges for optimal performance on multi-core architectures and presents our contributions. Section 5 discusses our experimental results and Section 6 concludes this paper.

## 2 Related work

Stencil-based computation is the building block of many numerical models. One of the critical aspects is to minimize the cost of the memory movements required to retrieve the data from the main memory to the fast local memories. There have been significant efforts on new techniques designed to improve the performance of these memory-bound kernels on different types of architectures [10, 7, 14]. Obviously, these efforts mainly tackle the improvement of data-reuse. Among them, space blocking techniques provide a first level of optimization but these approaches suffer from several limitations. First of all, the expected efficiency is bounded because of the very low reuse opportunity inherent to stencils. In [13], the limited speedup obtained from such strategies is described in a three-dimensional Jacobi loop (17%). Additionally, the impact of low-level mechanisms such as prefetching, vectorization or cache bypass are underestimated as described in [5, 1].

To overcome space tiling limitations, efforts have been focused on tiling the computational domain both in space and time directions. In this case, the algorithms exploit the iterative nature of the kernel. Time-skewing or cache-oblivious algorithms [8] rely on a similar idea to perform several time steps inside each subdomain. The main difference comes from the explicit blocking criterion in the first case whereas the oblivious approaches exploit a recursive cut in space in order to fit into cache memory. The search for the optimal spacetime tile shape is also a critical point, the so-called diamond tile shape exhibits the best performance [11, 12]. Regarding the growing importance of vectorization capabilities, the space-time decomposition could exhibit a high level of complexity for the compilers. Moreover, implementations at the application level is difficult as the organization of the code must be deeply revisited.

Much work has also been done on auto-tuning techniques. Several frameworks have been introduced to optimize stencil computations on modern architectures (e.g. PATUS [4] or PLUTO [2]). Amongst them, the Pochoir stencil compiler [15] aroused much interest. It implements stencil computations in a domain-specific language (DSL) and exploits a hyperspace

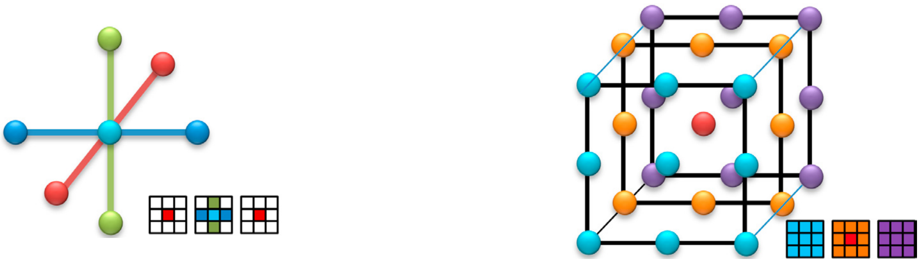

Figure 1: Visual representation of the 7-point and 27-point stencils.

cut algorithm in order to optimize cache reuse and parallelism. The Pochoir framework relies on the Cilk multithreading library and on the Intel compiler to vectorize.

# 3 Standard Stencils

## 3.1 7-point and 27-point Stencil Description

We consider two classical stencils commonly studied in the scientific literature [13, 6]. The first stencil is a 7-point 3D stencil which includes a central cell with its six nearest neighbors as described in figure 1. The second stencil is a 27-point 3D stencil composed of a total of 27 cells: 6 direct plus 20 indirect neighbors included in the $3 \times 3 \times 3$ cube around the central cell as described in figure 1. Both stencils are of great interest because they are representative of compute-bound and memory-bound situations. Indeed, we define the *Reuse Intensity* (RI) as a factor of reused floats per loaded bytes per stencil iteration. It is determined by the number of cells of the stencil divide by the number of bytes of a floating point value. The 7-point stencil performs the accumulation of 7 cells and each cell contains a 32 bits (4 bytes) floating point value. In this case, the reuse intensity factor is $7/4 = 1.75$ for 7-point stencil that is likely to be memory-bound. It means that each float per byte is reused 1.75 times. For the 27-point stencil, the RI is 6.75, indicating that this stencil is less memory-bound.

## 3.2 Classic Stencil Implementation

Stencil algorithms consist in traversing a grid and performing computation at each grid point. In the case of a three-dimensional Cartesian grid, the computation consists in using the neighbor points in the upward-downward, left-right and forward-backward directions to evaluate the value of the current grid point. The algorithm then iterates to the next point applying the same computation until the entire grid has been traversed. Considering the 7-point Jacobi stencil, one needs to deal with a domain of $n^3$ single precision values and adds neighbor weights to compute the result for the next iteration. On multi-core platforms, a classical way to extract parallelism is to exploit the nested loops coming from the spatial dimensions of the problem. One of the main advantages is a straightforward use of OpenMP directives that are applied to the external loop. A first level of optimization of this implementation is to use the collapse OpenMP option in order to extract the parallelism from the three spatial loops.

# 4 Optimizations

This section details the different techniques we propose to optimize stencil computation.

## 4.1 Manual Vectorization

Available in all modern processors, SIMD units are able to apply a single instruction on a vector of values.In our study, we consider using AVX units supported by our Intel platforms which are able to process vectors of eight floating point single precision values. Compilers may automatically vectorize some codes, but there is no guarantee that the process works nor that it provides the best vectorized code in terms of performance. Thus, we propose a hand-written vectorized version of the stencil computation to ensure that our code is correctly vectorized. We compared it with actual auto-vectorized codes produced by compilers. This version is written using compiler intrinsics and consists in computing eight contiguous cells at a time.

## 4.2 Tiling

Tiling consists in virtually decomposing the domain into tiles and adapting the traversal accordingly. In this case, we traverse the domain blockwise with a given tile size $(tx; ty; tz)$. When the whole domain does not fit into the cache, the standard stencil implementation makes the processor reload data from DRAM several times by iteration. If we traverse this domain by tiles fitting into the cache, we can expect to get closer to the theoretical RI. Therefore, the challenge is to determine the size of the tile that minimizes the amount of cache misses and provides the best performance. Optimized tile sizes have been experimentally found. Their larger dimensions are in the unit-stride direction which follows the literature [5, 12].

## 4.3 Stencil Composition

The standard stencil description underlines the impact of the Reuse Intensity (RI). For instance, the 7-point stencil exhibits poor theoretical performance. In this part, we introduce a strategy to build an equivalent of the 7-point stencil but with a higher density. This approach could be described as a similar strategy of spacetime stencil decomposition. Figure 2 provides an overview of the composition of two 7-point stencils. Regarding the reuse intensity, this new stencil is close to the 27-point stencil and is likely to improve the performance of the original 7-point stencil. Each evaluation of our composed stencil corresponds to two iterations of the original 7-point kernel, thus we perform twice more floating point operations. However, we expect to reach a higher level of performance. We remind that the RI of the 7-point stencil is 1.75 whereas the RI of the reformulated stencil with 25-point is 6.25.

The stencil composition is defined as follows. Let us consider two stencils $S_A$ and $S_B$ that must be applied consecutively, then it is possible to build a stencil $S_C$ that is the composition of $S_A$ and $S_B$. If we consider that $S_A$ and $S_B$ are of dimension $N$ where the size of each dimension are respectively $S_A.r_i * 2 + 1$ and $S_B.r_i * 2 + 1$ $(1 \leq i \leq N)$ then the composed stencil $S_C$ is also in size $N$ where the size of each dimension is $(S_A.r_i + S_B.r_i) * 2 + 1)$. We can determine that the extra-compute ratio between the successive application of $S_A$ and $S_B$ in comparison

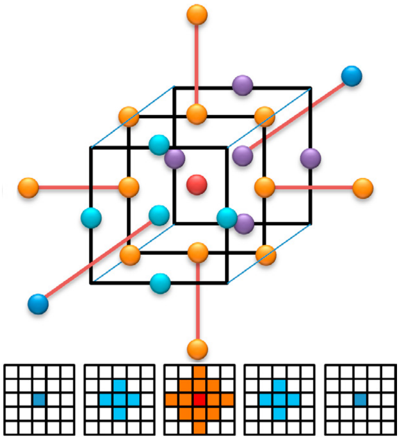

Figure 2: The 25-point stencil corresponding to the composition of two 7-point kernels.

with the application of $S_C$ is given by the following approximation :

$$\frac{(\prod_{i=1}^{N} 2(S_A.ri + S_B.ri) + 1) - 1}{(\prod_{i=1}^{N} 2S_A.ri + 1 + \prod_{i=1}^{N} 2S_B.ri + 1) - 2} \tag{1}$$

If we apply Equation 1 on the 27-point stencil, the half dimension is 1 and the extra-computation ratio is 2.29. In our case, for the 7-point stencil extra-computing ratio is 2.0 (2).

$$\frac{(25pts - 1)Additions}{2time\_steps * (7pts - 1)Additions} \tag{2}$$

# 5 Performance Analysis

## 5.1 Experimental Setup

We used two dual-socket platforms for our experiments:

- the first one with Intel Xeon E5-2697 v2 Ivy Bridge processors, for a total of twenty-four cores at 2.7Ghz;
- the second one with Intel Xeon E5-2697 v4 Broadwell processors, for a total of eighteen cores at 2.3Ghz.

Our codes are compiled with three different compilers: Clang 3.8, GCC 6.2 and ICC 17, with `-O3 -march=native` optimization flags, the second one implicitly enables AVX and FMA support on capable architectures. Multi-threading is performed using OpenMP. All results represent the maximum of ten runs to guarantee relevant values. The computation domain is composed of $512^3$ single precision floating point values. Therefore, the memory footprint is one order of magnitude larger than the size of the last level of cache memory. For each run, 100 iterations are performed. The following versions are implemented for both the 7-point and the 27-point stencils, except for the composition which only concerns the first stencil:

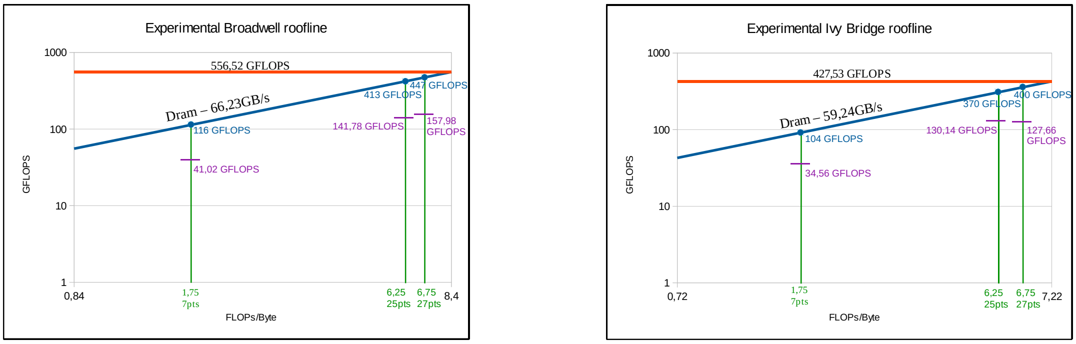

Figure 3: Experimental Stream and Linpack roofline models on Broadwell and Ivy Bridge platforms.

1. No vectorization: vectorization is disabled by the `-fno-tree-vectorize` compilation flag.
2. Automatic vectorization: compilers may automatically vectorize the code.
3. Manual vectorization: manual vectorization is performed as described in Section 4.
4. Manual vectorization with tiling: this version adds tiling to the previous version.
5. 7-point stencil composition: this version applies the 25-point stencil resulting from the self composition of the 7-point stencil and consequently performs only 50 iterations.
6. Pochoir: this version requires the Intel compiler since it is based on the Cilk threading library instead of OpenMP.

## 5.2 Performance Characterization

### 5.2.1 Roofline Models

Figure 3 shows the roofline models obtained on Broadwell and Ivy Bridge platforms. It is a good visual tool to discuss both the performance of our baseline implementations but also to compare the impact of the reuse intensity (RI) of each stencil. First, we can observe that the achievable performance is strongly related to the RI. In the first case, the 7-point kernel is higtly memory-bound and the achievable peak performance is limited to an average upper-bound of 100 GFLOPS on both platforms. For the 27-point stencil, the RI is higher (6.25 vs 1.75 for the 7-point) leveraging from the higher density of this kernel. Therefore, the upper-bound achievable performance is 413 GFLOPS on Broadwell and 368 GFLOPS on Ivy Bridge. Nevertheless, these stencils are still memory-bound as their RI are below the compute-bound limit measured on each platform (8.4 on Broadwell and 7.2 Ivy Bridge platforms). Secondly, if we combine manual vectorization and space tiling optimization we almost reach 30% of the achievable peak performance for the 7-point and 27-point stencils on both platforms. In fact, the 7-point and 27-point stencils perform respectively 41 GFLOPS over 116 GFLOPS and 158 GFLOPS over 447 GFLOPS on the Broadwell platform. Moreover, the Ivy Bridge platform achieves 35 GFLOPS over 104 GFLOPS and 128 GFLOPS over 400 GFLOPS with respectively the 7-point and 27-point stencils. These results are rather good as we do not take into account advanced ceiling for our roofline models. For instance, one may consider that with only additions, processors may no use all its operator ports. Whereas a theoretical peak performance can be determined from all the available ports. As a result, our kernels have a significantly reduced achievable peak performance.

#### 5.2.2 Comparison with the theoretical Results

The reuse intensity is a tool to check the level of performance of our implementations. The theoretical ratio between the RI of the 27-point and the 7-point stencil is $6.75/1.75 \approx 3.85$. We use this ratio to discuss the performance observed on the different architectures. Regarding the Clang compiler auto vectorization version, the ratio measured is 1.97 on Broadwell and 2.04 on Ivy Bridge. These values are far from the theoretical value. Additional investigations demonstrate that Clang compiler is unable to vectorize the standard OpenMP implementations. Combination of the complexity of the nested loops and side effects from the use of OpenMP directives could explain this situation [9]. GCC and ICC compilers vectorize the code leading to better results. Another issue is the DRAM memory access delay. Any loaded data from DRAM into caches have to be reused as mush as possible in order to avoid some DRAM access delays. As we supposed a 3D grid bigger than caches, each grid cell is loaded at least once by iteration. At the best, they are loaded only once by iteration. To get close to it, it needs the spacial cache reuse to be optimized. This is the aim of the tile optimization. Table 1 shows our tiled vectorized peak performance ratios 27-point/7-point. The performance ratios of the manually vectorized and tiled versions (version 4) are closer to 3.85 than the ones for the automatic vectorized versions (version 2). Thus, we observe a good match between the theoretical RI ratio and these experiments. This version shows a reliable ratio of 3.13.

| Machine | Clang compiler | | Intel Compiler | | GCC Compiler | |
|---|---|---|---|---|---|---|
| | 2-Auto | 4-ManuTiled | 2-Auto | 4-ManuTiled | 2-Auto | 4-ManuTiled |
| Broadwell | 1.97 | 3.45 | 4.28 | 3.88 | 4.18 | 3.74 |
| Ivy Bridge | 2.04 | 2.59 | 4.24 | 4.14 | 4.20 | 3.72 |

Table 1: Ratios between 7-point and 27-point kernels of automatically vectorized version 2 and our best optimized version 4. The theoretical value ration is about 3.85.

### 5.3 Multi-level Optimization

We evaluate the improvement coming from our optimizations including a new algorithmic formulation introduced for the 7-point stencil. Figure 4 shoes the results we obtained.

#### 5.3.1 Impact of Vectorization

The comparison between manual and automatic vectorization underlines the difficulty for the compiler to fully exploit this level of parallelism. As described in the previous section, the Clang compiler is struggling to be efficient on the 27-point stencil that exhibits complex multi-threaded nested loops. Indeed, the compiler uses a lot of uload/ustore and does not detect that the buffers are aligned. Moreover, in our case, the OpenMP transformations may inhibit the Clang automatic vectorization. For these reasons, the scalar version and the automatic vectorization version of Clang have the same performance. This is the reason why only the scalar version is shown on figures 4. Our manual vectorization strategy provides a significant

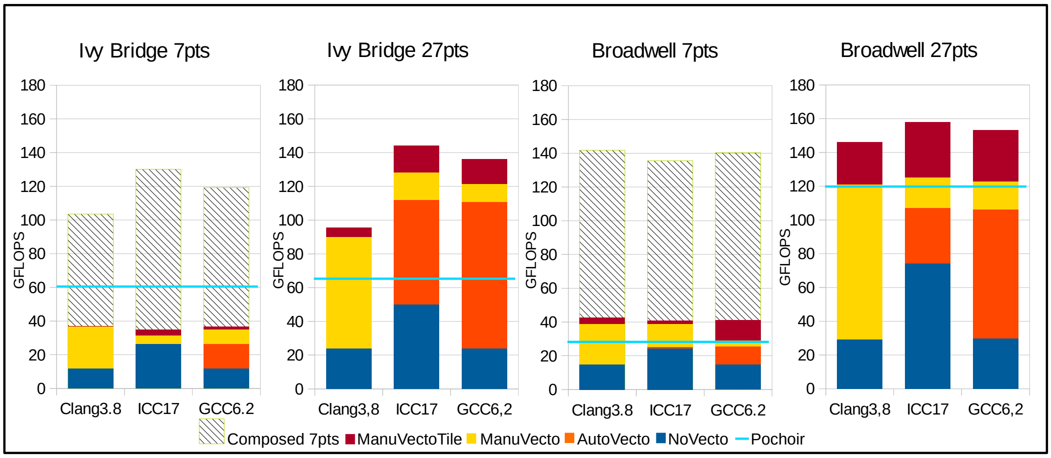

Figure 4: Impact of the multi-level optimization strategy on the 7-point and the 27-point stencils with the Clang, Intel and GCC compilers on a dual-socket Intel Xeon Broadwell (E5-2697v4) and a dual-socket Intel Xeon Ivy Bridge platform.

improvement in comparison with the results obtained with the three compilers. We observe an average gain of 1.8× for Intel and GCC compilers and a maximum gain of a factor 4.2 for Clang compiler. As a matter of fact, the 7-point stencil can hardly benefit from these improvements as this kernel quickly saturates the memory bandwidth available.

### 5.3.2 Impact of Tiling

Tiling is a strategy to enhance data-reuse.We learn from figures 4 that the impact of this optimization is rather strongly dependent on the stencil characteristics.For instance, we observe limited gains for the 7-point stencil (less than 10% on Broadwell and about 0.3% on Ivy Bridge) whereas the 27-point stencil is able to benefit from this strategy (more than 25% on Broadwell and more than 12% on Ivy Bridge). This is mainly coming from the shape of the stencil and the ratio between the computation and the memory movement that offers more opportunities to reuse data in the 27-point case. The optimal tile size selection is another issue. We solve it for each stencil and architecture thanks to an experimental approach guided by the literature [5, 12].

### 5.3.3 Impact of Stencil Composition

The last stage of our optimization strategy corresponds to a new formulation of the numerical kernel. We evaluate this strategy for the 7-point example as this kernel is more likely to leverage this improvement.

In terms of GFLOPS, we observe good performances with Clang, Intel and GCC compilers as the measured speedup on Broadwell is respectively 3.34×, 3.33× and 3.42×. On the Ivy Bridge, we also have respectively 2.8×, 3.74× and 3.26× speedups. The reuse intensity of the 25-point stencil (new formulation of the 7-point stencil) allows us to compute with a higher efficiency whatever the compiler or the platform. Additionally, we maintain the coherency with the theoretical analysis. Figure 4 shows the peak performance measured with the 25-point and the 27-point stencil. The measurements are very similar as both stencils exhibit very close data reuse capabilities. Elapsed times are reported in table 2. As explained in the subsection 4.3, the

25-point stencil is the result of the 7-point stencil self-composition but it requires 4 times more additions by iteration (from 6 to 24 additions). However, one iteration of the 25-point stencil is equivalent to 2 iterations of the 7-point stencil. Thus, to obtain the same numerical results, we only need half the number of iterations. As a result, we perform twice more additions for the whole computation. Then, in terms of elapsed time, the measured speedup on Broadwell is respectively divided by 2 and are equal to 1.67×, 1.67× and 1.7×. On the Ivy Bridge, we also have respectively 1.4×, 1.87× and 1.63× speedups. We are still investigating these odd Pochoir performances.

| Machine | Stencil | steps | Clang | ICC | GCC | Pochoir |
|---|---|---|---|---|---|---|
| Ivy Bridge | 7-point | 100 | 2157 | 2288 | 2174 | 1316 |
| | 25-point | 50 | 1520 | 1306 | 1339 | 2534 |
| | 27-point | 100 | 3613 | 2393 | 2532 | 5289 |
| Broadwell | 7-point | 100 | 1875 | 1956 | 2117 | 2810 |
| | 25-point | 50 | 938 | 1160 | 1276 | 2301 |
| | 27-point | 100 | 2358 | 2183 | 2249 | 2893 |

Table 2: Elapsed time (in ms) of the 7-point (RI=1.75) and the 25-point (RI=6.25) stencil.

## 6 Conclusion and Perspectives

As shown in our experiments, optimizing stencil kernels is not straightforward since efficient optimization techniques depend on the stencil morphology, on the underlying architecture and on compilers. Taken independently, these optimization techniques are simple to implement, but they need to be combined in order to reach near-peak performance, and some parameters, like the tile size, may need to be tuned for each architecture. The Pochoir library relies on a complex space and time decomposition which outperforms our implementation (tiling and manual vectorization) only for the 7-point stencil on the Ivy Bridge architecture. However, our stencil composition is able to reach a similar level of performance on this architecture. In all other cases, our best implementation is faster than Pochoir but the speedup is highly variable depending on the architecture, on the stencil and on the compilers. Of course, faster implementations may be obtained since we do not reach the upper bound of the Roofline, but reducing the remaining gap may require to increase the code complexity.

Despite encouraging results, the implementation of our contributions in some real applications may not seem straightforward nor reasonable. Indeed, these optimizations are tied to the underlying architecture which may prevent code portability. To solve this problem, we plan to use the same approach as Pochoir: splitting the interface from the optimized implementation. A Domain Specific Language (DSL) would be provided to the developer who focuses on the description of its stencil computation. Optimized implementation would be obtained using either a specific compiler, like Pochoir, or meta-programming techniques.

## 7 Acknowledgments

The authors thank Philippe Thierry, senior principal engineer at Intel, for many interesting discussions and for providing us access to Broadwell and Ivy Bridge platforms.
The work of G. Sornet is co-funded by the Région Centre-Val de Loire.

# References

[1] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Optimized Stencil Computation Using In-Place Calculation on Modern Multicore Systems. In *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference*, pages 772–784, Delft, The Netherlands, August 2009.

[2] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *PPOPP*, pages 219–228, 2009.

[3] Márcio Castro, Emilio Francesquini, Fabrice Dupros, Hideo Aochi, Philippe O. A. Navaux, and Jean-François Méhaut. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108–120, 2016.

[4] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Comput. Sci.*, 26(3-4):205–210, June 2011.

[5] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[6] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Auto-tuning the 27-point stencil for multicore. In *In In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[7] Hikmet Dursun, Ken-Ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. A Multilevel Parallelization Framework for High-Order Stencil Computations. In *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference*, pages 642–653, Delft, The Netherlands, August 2009.

[8] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[9] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. Extending openmp* with vector constructs for modern multicore SIMD architectures. In *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*, pages 59–72, 2012.

[10] Marcin Krotkiewski and Marcin Dabrowski. Efficient 3d stencil computations using {CUDA}. *Parallel Computing*, 39(10):533 – 548, 2013.

[11] Tareq M. Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David E. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM J. Scientific Computing*, 37(4), 2015.

[12] Takayuki Muranushi and Junichiro Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51:1303 – 1312, 2015.

[13] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pages 32–38, Dallas, United States, 2000.

[14] Andreas Schafer and Dietmar Fey. High performance stencil code algorithms for gpgpus. *Procedia Computer Science*, 4:2027 – 2036, 2011.

[15] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[16] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.