Algorithm XXX: Semi-stencil

RAÚL DE LA CRUZ, Barcelona Supercomputing Center
MAURICIO ARAYA-POLO, Repsol USA

Finite Difference (FD) is a widely used method to solve Partial Differential Equations (PDE). PDEs are the core of many simulations in different scientific fields, e.g., geophysics, astrophysics, etc. The typical FD solver performs stencil computations for the entire computational domain, thus solving the differential operators. In general terms, the stencil computation consists of a weighted accumulation of the contribution of neighbor points along the cartesian axis. Therefore, optimizing stencil computations is crucial in reducing the application execution time.

Stencil computation performance is bounded by two main factors: the memory access pattern and the inefficient reuse of the accessed data. We propose a novel algorithm, named Semi-stencil, that tackles these two problems. The main idea behind this algorithm is to change the way in which the stencil computation progresses within the computational domain. Instead of accessing all required neighbors and adding all their contributions at once, the Semi-stencil algorithm divides the computation into several updates. Then, each update gathers half of the axis neighbors, partially computing at the same time the stencil in a set of closely located points. As the Semi-stencil progresses through the domain, the stencil computations are completed on precomputed points. This computation strategy improves memory access pattern and efficiently reuses the accessed data.

Our initial target architecture was the Cell/B.E., where the Semi-stencil in a SPE was 44% faster than the naive stencil implementation. Since then, we have continued our research on emerging multi-core architectures in order to assess and extend this work on homogeneous architectures. The experiments presented combine the Semi-stencil strategy with space and time-blocking algorithms used in hierarchical memory architectures. Two x86 (Intel Nehalem and AMD Opteron) and two POWER (IBM POWER6 and IBM BG/P) platforms are used as testbeds, where the best improvements for a 25-point stencil range from 1.27 to 1.76× faster. The results show that this novel strategy is a feasible optimization method which may be integrated into auto-tuning frameworks. Also, since all current architectures are multi-core based, we have introduced a brief section where scalability results on IBM POWER7, Intel Xeon and MIC based systems are presented. In a nutshell, the algorithm scales as well as or better than other stencil techniques. For instance, the scalability of the Semi-stencil on MIC for a certain testcase reached 93.8× over 244 threads.


General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: stencil computation, Semi-stencil, Blocking, Time-skewing, Cache-Oblivious, HPC, code optimization, numerical algorithms, performance model

This work was supported by project TIN2007-60625 of Spanish Government’s Science and Innovation Ministry.

Authors’ address: Raúl de la Cruz, CASE Department, Barcelona Supercomputing Center, Barcelona, Spain; email: delacruz@bsc.es; Mauricio Araya-Polo, Repsol USA, The Woodlands, TX, USA; email: araya.mauricio@repsol.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 0098-3500/YYY/01-ARTA $15.00
D01 : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Astrophysics [Brandenburg 2003], Geophysics [McMechan 1989; Operto et al. 2006], Quantum Chemistry [Alonso et al. 2008; Castro et al. 2006] and Oceanography [Korrmann et al. 2008; Groot-Hedlin 2008] are examples of scientific fields where large computer simulations are frequently carried out. These simulations have something in common, the physical models are represented by Partial Differential Equations (PDE), which are mainly solved by the Finite Difference (FD) method. This method uses stencil computations to obtain values for discrete differential operators. Large scale simulations may consume days of supercomputer time, further, most of this execution time is spent on the stencil computation if they are PDE+FD based. For instance, Reverse-Time Migration (RTM) is a seismic imaging technique from geophysics used in the oil and gas industry, where up to 80% of the RTM kernel execution time [Araya-Polo et al. 2008] is spent on stencil computation.

The basic structure of stencil computation is that the central point accumulates the contribution of neighbor points in every axis of the Cartesian system. The number of neighbor points in every axis relates to the accuracy level of the stencil, where more neighbor points lead to higher accuracy. The stencil computation is then repeated for every point in the computational domain, thus solving the spatial differential operator.

Two inherent problems can be identified from the structure of the stencil computation:

— First, the non-contiguous memory access pattern. In order to compute the central point of the stencil, a set of neighbors have to be accessed. Some of these neighbor points are distant in the memory hierarchy, requiring many cycles in latencies to be accessed [Kamil et al. 2005; Kamil et al. 2006]. Furthermore, with increasing stencil order, it becomes more expensive to access the required data points. Hardware prefetchers can hide the latency issue, but only if the cache lines used do not exceed the number of streaming prefetchers.

— Second, the low computational-intensity and reuse ratios. After gathering the set of data points, just one central point is computed and only the accessed data points in the sweep direction might be reused for the computation of the next central point [Frigo and Strumpen 2005]. Thus, some of the accessed data are not reused and the current hierarchical memory structures are poorly exploited.

In order to deal with these issues and improve the stencil computation, we introduce the Semi-stencil algorithm. This algorithm changes the structure of the stencil computation, but it can be generally applied to most stencil based problems. Indeed, the Semi-stencil algorithm computes half of the axis contributions required by several central points at the same loop iteration. By just accessing the points required to compute half of the stencil axis, this algorithm reduces the number of neighboring points loaded per iteration. At the same time, the number of floating point operations remains the same, but because the number of loads is reduced, the computation-access ratio is increased.

This approach has already been shown to be effective in the heterogeneous Cell/B.E. architecture [de la Cruz et al. 2009] to optimize real scientific applications such as RTM [Araya-Polo et al. 2008]. The Cell/B.E. stencil implementation of the RTM includes the Semi-stencil strategy, the SIMDization of the code and other pipeline optimizations. The single-SPE performance on a IBM Bladecenter QS22 is 12.44 GFlops.
Algorithm XXX: Semi-stencil

for an 8th order stencil (25-point stencil). This gives 49% of the SPE peak performance (25.6 GFlops) compared to 34% (8.61 GFlops) obtained with the naive implementation.

In the current work, we present a comprehensive study of the Semi-stencil strategy on homogeneous multi-core architectures with hierarchical memories. Unlike other stencil optimization works [Datta et al. 2008; Datta et al. 2009; Datta et al. 2010] where a 7-point academic stencil (single Jacobi iteration) is tackled through autotuning environments, we perform an evaluation of 25 and 43-point stencils (from academic up to widely used in the industry). Furthermore, the aim of this work is not to achieve the fastest stencil implementation on the chosen platforms, but to prove the soundness of the Semi-stencil algorithm when scientific codes are optimized. Hence, a future goal may be to include the Semi-stencil into auto-tuners to take advantage of hardware dependent optimization techniques such as: register blocking, SIMDization or cache-bypassing.

The remainder of this paper is organized as follows: Section 2 introduces the stencil algorithm analyzing its main characteristics and challenges. In Section 3, the principal techniques that help to cope with the stencil problem are reviewed. Section 4 introduces the novel algorithm, its internals, features and considerations. Section 5 depicts the performance modeling methods available in the literature in order to characterize the performance of a specific stencil algorithm. In Section 6, we present the testbed platforms and, the serial and parallel performance are evaluated for each algorithm with the Semi-stencil strategy. Finally, Section 7 presents the conclusions of this work.

2. THE STENCIL PROBLEM

The stencil computes the spatial differential operator, which is required to solve PDEs numerically on FD schemes. A multidimensional structured grid (often a huge 3D data structure) is traversed and the elements \( \chi_{i,j,k}^t \) are updated with weighted contributions. Figure 1.b depicts a generic 3D stencil pattern, where the \( \ell \) parameter represents the number of neighbors to be used in each direction of the Cartesian axis. Hence, the computation of a \( \chi_{i,j,k}^t \) point at time-step \( t \) requires \( \ell \) weighted neighbors in each axis direction at the previous time-step \( (t-1) \). The domain contains interior points (the solution domain) and ghost points (outside of the solution domain). This operation is repeated for every point in the solution domain, finally approximating the spatial differential operator.

The first problem identified from this computation is the sparse memory access pattern. Data is stored in Z-major order (unit-stride dimension), and therefore accesses across the other two axes (X and Y) may be significantly more expensive latency-wise (see Figure 1.a). The \( \ell \) parameter has a direct impact on this problem, the larger the \( \ell \) value the more neighbors at each axis have to be loaded to compute the \( \chi_{i,j,k}^t \) point.

The second problem to deal with is the low floating-point operations to data cache accesses ratio, which is related to the poor data reuse. In this work, we use the \( \text{FP}/\text{Cache} \) metric to quantify these issues; this is the ratio of floating-point operations to the number of loads and stores issued to L1 data cache for one \( \chi^t \) point computation:

\[
\text{FP}/\text{Cache}_{\text{Classical}} = \frac{\text{FloatingPoint Operations}}{\text{Data Cache Accesses}} = \frac{2 \times \text{MultiplyAdd Instructions}}{\chi^t - 1 \text{ Loads} + \chi^t \text{ Stores}} = \frac{2 \times \text{MultiplyAdd Instructions}}{4 \times \text{dim} \times \ell - 2 \times \ell + 2}
\]

(1)
Equation 1 states this metric for the classical stencil computation shown in Algorithm 1. \( \text{dim} \) is the number of dimensions of our PDE (3 in a 3D problem), where 2 * \( \ell \) Multiply-Add instructions are required at each dimension. Furthermore, one extra multiplication operation must be considered for the self-contribution (\( \mathcal{X}_{t,i,j,k}^{t-1} \)). The number of loads needed to compute a stencil point differs depending on the axis. Those axes that are not unit-stride dimension (X and Y) require 2 * \( \ell \) loads at each loop iteration. However, the \( Z \) unit-stride dimension tends to require just one load due to the load reuse from preceding loop iterations. Finally, to conclude the \( \mathcal{X}_{t,i,j,k} \) point computation requires one store to save the result.

As shown in Equation 1, the \( \text{FP/Cache} \) ratio depends on the \( \text{dim} \) and \( \ell \) variables. Taking into account that \( \ell \) is the only variable that may increase, the \( \text{FP/Cache} \) ratio tends to a factor of \( \approx 3 \) flops per data cache (d-cache) access for 3D problems. This relatively low ratio along with some previous works’ results [Rivera and Tseng 2000; McCalpin and Wonnacott 1999; Frigo and Strumpen 2005] shows that the stencil computation is usually memory-bound. In conclusion, the execution time of the stencil computation is mainly dominated by the memory access latency.

These concerns force us to pay special attention to how data is accessed during the computation. It is crucial to improve the memory access pattern (by reducing the overall number of data transfers) and exploit the memory hierarchy as much as possible (by reducing the overall transfer latency). Section 3 reviews the main approaches that can be found in the literature to address these issues.

3. RELATED WORK

Most of the contributions for stencil computations can be divided into three dissimilar groups: space blocking, time blocking and pipeline optimizations. Space and time blocking are related to tiling strategies widely used in multi-level cache hierarchy architectures. Pipeline optimizations refer to those optimization techniques that are used...
Algorithm 1 The classical stencil algorithm pseudo-code for a 3D problem. $X^t$ is the space domain for time-step $t$, where $Z$, $X$ and $Y$ define the dimensions (ordered from unit to least-stride) of the datasets including ghost points (points outside of the solution domain). $t$ denotes the number of neighbors used for the central point contribution. $C_{Z1...Zt}$, $C_{X1...Xt}$, $C_{Y1...Yt}$ are the spatial discretization coefficients for each dimension and $C_0$ for the self-contribution. $z_s$, $z_c$, $x_s$, $x_c$, $y_s$ and $y_c$ denote the area in $X^t$ where the stencil operator is computed. In order to compute the entire dataset, the stencil pseudo-code must be called as follows: STENCIL($X^t$, $X^{t-1}$, $t$, $Z - t$, $X - t$, $Y - t$). Notice that in this work the discretization coefficients are considered constant.

```plaintext
1: procedure STENCIL($X^t$, $X^{t-1}$, $t$, $z_s$, $z_c$, $x_s$, $x_c$, $y_s$, $y_c$)
2: for $k = y_s$ to $y_c$ do
3: for $j = x_s$ to $x_c$ do
4: for $i = z_s$ to $z_c$ do
5: $X_{i,j,k}^t = C_0 * X_{i,j,k}^{t-1}$
6: + $C_{Z1} * (X_{i-1,j,k}^{t-1} + X_{i+1,j,k}^{t-1}) + ... + C_{Zt} * (X_{i-t,j,k}^{t-1} + X_{i+t,j,k}^{t-1})$
7: + $C_{X1} * (X_{i,j-1,k}^{t-1} + X_{i,j+1,k}^{t-1}) + ... + C_{Xt} * (X_{i-t,j,k}^{t-1} + X_{i+t,j,k}^{t-1})$
8: + $C_{Y1} * (X_{i,j,k-1}^{t-1} + X_{i,j,k+1}^{t-1}) + ... + C_{Yt} * (X_{i,j,k-t}^{t-1} + X_{i,j,k+t}^{t-1})$
9: end for
10: end for
11: end for
12: end procedure
```

at the CPU pipeline level to improve the instruction throughput performance (IPC). In addition, these three groups contain incremental optimization techniques which can be combined with techniques from other groups.

Figure 2 categorizes these three groups by complexity of implementation (effort), benefit improvement (performance) and implementation tightness regarding hardware (dependency). Even though each algorithm property (effort, performance and dependency) can differ widely depending on the code complexity and the underlying hardware, this classification can be treated as the big picture of the state-of-the-art. In this diagram, an interesting optimization algorithm would be classified in the top-left corner, whereas an inefficient one would be in the bottom-right corner. The next sections will review the advantages and disadvantages of these optimizations methods.

3.1. Space Blocking

Space blocking algorithms promote data reuse by traversing data in a specific order. Space blocking is especially useful when the dataset structure does not fit into the memory hierarchy. The most representative algorithms of this kind are:

— Tiling or blocking techniques: Rivera and Tseng [2000] propose a generic blocking scheme for 3D stencil problems. The entire domain is divided into small blocks of size $TI \times TJ \times TK$ which must fit into the cache. Rivera and Tseng showed that a good blocking scheme configuration can be achieved when a $TI \times TJ$ 2D block is set along the less-stride dimension. Later, Kamil et al. [2005] found out that the best configuration is usually given when $TJ$ is equal to the grid size $I$, as shown in Figure 3. This traversal order reduces cache misses in less-stride dimensions ($X$ and $Y$), which may increase data locality and overall performance. Note that a search of the best block size parameters ($TI \times TJ$) must be performed for each problem size and architecture.

— Circular queue: Datta et al. [2008] employ a separate queue data structure to perform the actual stencil calculations. This structure stores only as many 2D planes as are
Fig. 2. Characterization of different optimization schemes. This diagram sorts each optimization method according to the three properties: programming effort required to implement it, performance boost by optimizing the code and the dependency of the implementation with respect to the underlying hardware.

Fig. 3. Left: 3D cache blocking proposed by Rivera, where $TI \times TJ$ cuts are computed sweeping through the unblocked dimension $K$. Right: $I \times TJ \times K$ blocking configuration suggested by Kamil, where the $I$ and $K$ dimensions are left unblocked. In this example, $I$ and $K$ are the unit-stride and the least-stride dimensions respectively.

needed for the given stencil. After completing a plane, the pointer to the lowest read plane is moved to the new top read plane, thus making a circular queue. In general, the circular queue stores $(t-1)$ sets of planes, where $t$ is the number of iterations performed.
3.2. Time Blocking

Time blocking algorithms perform loop unrolling over time-step sweeps to exploit the grid points as much as possible, and thus increase data reuse. Such techniques have shown some effectiveness in real infrastructures [ANAG 2012], but they require careful code design. However, due to the time dependency scenario, these techniques may pose implementation issues when other tasks must be carried out between stencil sweeps. For instance, boundary condition computation, intra- and extra-node communication or I/O may occur in many scientific applications during time-step updates. Time blocking techniques can be divided into explicit and implicit algorithms:

— Time-skewing: McCalpin and Wonnacott [1999] try to reuse data in cache as much as possible. As a result, memory transfers are reduced and execution stalls are minimized. Essentially, several cache blocks of size \( T_1 \times T_J \times T_K \) are generated over space dimensions of the grid and each of those blocks is unrolled over time, as shown in Figure 4(a). To keep data dependencies of stencil computations, block computations must be executed in a specific order. This constraint makes the time-skewing algorithm only partially effective in parallel executions. However, time-skewing has been already parallelized [Wonnacott 2000; Kamil et al. 2006] with reasonable results. Wonnacott devised the parallel version of the time-skewing algorithm by performing space cuts in the least-stride dimension. Each thread block is divided into three parallelepipeds, which are computed in a specific order to preserve data dependencies and enable parallel computation. In this algorithm, as in space tiling, a search of the best block size parameters must be performed prior to the start of the computation.

— Cache-oblivious: the Frigo and Strumpen [2005] time blocking algorithm tiles both space and time domains. In contrast with time-skewing, the cache-oblivious algorithm does not require explicit information about the cache hierarchy. As Figures 4(b,c,d) show, an \((n+1)\) dimensional space-time trapezoid is considered, where all the spatial dimensions plus time are represented. Cuts can be performed recursively over the trapezoid in space or time to generate two new, smaller trapezoids \((T_1\text{ and }T_2)\), where the stencil can be computed in an optimal way due to size constraints of the cache hierarchy. Cutting in space produces a left side trapezoid \((T_1)\) where there is no point depending on the right side trapezoid \((T_2)\), thus allowing \(T_1\) to be fully computed before \(T_2\). In addition, recursive cuts can be taken over the time dimension. In this cut, the original time region \((t_0, t_1)\) is split into a lower \(T_1\) trapezoid \((t_0, t_n)\) and an upper \(T_2\) trapezoid \((t_n, t_1)\). As in space cut, \(T_1\) does not depend on any point of \(T_2\), and \(T_1\) can be computed in advance. Cache-oblivious has been also parallelized [Frigo and Strumpen 2006] by creating space cuts (either in the \(Z\) or \(Y\) dimensions) of inverted and non-inverted trapezoids, which are computed in order to preserve dependencies.

3.3. Pipeline Optimizations

Low level optimizations at the CPU pipeline level include several well-known techniques. These techniques may be categorized into three groups: loop transformations, data access and streaming optimizations. Loop unrolling, loop fission and loop fusion are part of the loop transformations group. They can reduce the overall execution time of scientific codes by reducing register pressure and data dependency as well as improving temporal locality [McKinley et al. 1996; Manjikian and Abdelrahman 1997].

Data access optimizations include techniques such as software prefetching, software pipelining and register blocking [Callahan et al. 1991; Mowry and Gupta 1991; Allan et al. 1995; Rogers and Li 1992; Lam et al. 1991], all of these relate to improving data access latency within the memory hierarchy (from register to main memory level). Fi-
nally, Symmetric Multi-Processing (SMP), Single-Instruction Multiple-Data (SIMD) or Multiple-Instruction Multiple-Data (MIMD) are some programming paradigms which are included in the streaming optimizations group. The multiprocessor term refers to the ability to execute multiple processes, threads and instructions concurrently in a hardware system.

All the previous techniques have been successfully used in many computational fields to improve the processing throughput and increase the IPC metric (Instructions Per Cycle). Furthermore, the performance of the codes can be improved significantly by combining several of these, although collateral effects (performance issues) can appear from time to time. Some modern compilers are able to generate code automatically that takes advantage of some of these techniques, thus relieving the developer of the tedious implementation work. However, some pipeline optimizations may increase both instruction code size and register pressure. Therefore, they must be used carefully, since overuse can lead to register spilling, which would produce slower code due to additional saves and restores of register code from the stack.

The stencil code representation in terms of arithmetic instructions is another issue to bear in mind. Depending on the hardware, one type of stencil code codification could perform better than others. Table I shows two different ways of stencil representation: factored and expanded. The former uses add and multiplication instructions, while the latter maps more naturally to fused multiply-add and multiplication instructions.

4. THE SEMI-STENCIL ALGORITHM

The Semi-stencil algorithm involves noticeable changes to the structure as well as the memory access pattern of the stencil computation. This new computation structure (depicted in Figure 5) consists of two phases: forward and backward updates, which are described in detail in Section 4.1. Additionally, due to this new structure, three parts of code, called head, body and tail need to be developed to preserve the numerical soundness; these are described at the end of this section. To conclude this section, it
Algorithm XXX: Semi-stencil

Table 1. Operation and instruction cost of two representations of stencil codes. Depending on the instruction set and the latency (in cycles) of each arithmetic instruction in the CPU pipeline, one specific representation (factored or expanded) may outperform the other.

<table>
<thead>
<tr>
<th>Method</th>
<th>Stencil code</th>
<th>Operations</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factored (Muls and Add</td>
<td>[X_{i,j,k}^{t} = C_0 \cdot X_{i-1,j,k}^{t-1} + C_{Z1} \cdot (X_{i-1,j,k}^{t-1} + X_{i+1,j,k}^{t-1}) + \ldots + C_{Zl} \cdot (X_{i-l,j,k}^{t-1} + X_{i+l,j,k}^{t-1}) ]</td>
<td>3 * dim * \ell + 1</td>
<td>Muls: 3 * dim * \ell + 1</td>
</tr>
<tr>
<td></td>
<td>(Add inst. based)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expanded (Fused Multiply-</td>
<td>[X_{i,j,k}^{t} = C_0 \cdot X_{i-1,j,k}^{t-1} + C_{Z1} \cdot X_{i+1,j,k}^{t-1} + \ldots + C_{Zl} \cdot X_{i+l,j,k}^{t-1} ]</td>
<td>4 * dim * \ell + 1</td>
<td>Muls: 2 * dim * \ell + 1</td>
</tr>
<tr>
<td></td>
<td>Add inst. based)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

will be shown how Semi-stencil is orthogonal and may be combined with any other optimization technique.

As shown in Section 2, there are two main issues in stencil computations. First, the low floating-point operation to data cache access ratio and, second, the high latency memory access pattern especially for the last two dimensions. Basically, the Semi-stencil algorithm tackles these two issues in the following ways:

— It improves data locality since less data is required on each axis per loop iteration. This may have an important benefit for hierarchical cache architectures, where the non-contiguous axes (X and Y) of a 3D domain are more expensive latency-wise.

— The new memory access pattern reduces the total number of loads per inner loop iteration, while keeping the same number of floating-point operations. Thereby, increasing the data reuse and thus the FP/Cache ratio, relative to the classical estimate of Equation 1.

As mentioned before, two phases (forward and backward updates) are needed to carry out the new memory access pattern of the Semi-stencil algorithm. These are the core of the algorithm, and the following subsection elaborates them.

4.1. Forward and Backward Updates

Line 5 of Algorithm 1 shows, in computational terms, the generic spatial operator for FD codes. This line of pseudo-code updates the value of \(X_{i,j,k}^{t}\) in one step (iteration \(\{i, j, k\}\) of the loop). The essential idea of the Semi-stencil is to break up this line into two phases, thereby partially updating more than one point per loop iteration. To carry out this parallel update, the factored add and mul operations \((c_1 \cdot (x_{-i} + x_{+i}))\) must be decomposed into multiply-add instructions \((c_1 \cdot x_{-i} + c_1 \cdot x_{+i})\) in order to split up the classical computation into forward and backward updates.

The forward update is the first contribution that a \(X^{t}\) point receives at time-step \(t\). During this phase, when the \(i\) iteration of the sweep axis is being computed, the \(X_{i+\ell}^{t}\) point is updated with \(\ell\) \(X^{t-1}\) rear contributions (as depicted in Figure 5.a). Through this operation we obtain a precomputed \(X_{i+\ell}^{t-1}\) value. Recall that \(\ell\) represents the number of neighbors used in each axis direction. Equation 2 shows, in mathematical terms, a summary of the forward operations performed over \(X_{i+\ell}^{t}\) point in a one-dimensional (1D) problem,

\[
X_{i+\ell}^{t} = C_1 \cdot X_{i+\ell-1}^{t-1} + C_2 \cdot X_{i+\ell-2}^{t-1} + \ldots + C_{\ell-1} \cdot X_{i+1}^{t-1} + C_{\ell} \cdot X_{i}^{t-1}
\] (2)
where the prime character (′) denotes that the point has been partially computed, and some contributions are still missing. In addition, only \( \ell \) neighbor points of \( X_{t-1} \) have been loaded and one \( X_t \) point, \( X_t' \), stored so far.

In a second phase, called backward, a previously precomputed \( X_t' \) point in the forward update is completed by adding in the front axis contributions (\( X_{t-1} \) to \( X_{t+\ell} \)). More precisely,

\[
X_t = X_t' + C_0 \cdot X_{t-1} + C_1 \cdot X_{t+1} + C_2 \cdot X_{t+2} + \cdots + C_{\ell-1} \cdot X_{t+\ell-1} + C_\ell \cdot X_{t+\ell}
\]

(3)

(see also Figure 5.b).

Only two loads are required in this process to complete the backward computation, since most of the \( t-1 \) time-step points were loaded during the forward update of the \( X_t \) computation and hence they can be reused. The two loads required for this second phase are: the \( X_{t-1} \) point and the precomputed \( X_t' \) value. This phase also needs one additional store to write the final \( X_t \) value.

Finally, in order to carry out both updates, \( 2 \cdot \ell + 1 \) floating-point operations are issued in the inner loop (\( \ell \) Multiply-Add instructions for neighbor contributions and one multiplication for the self-contribution). Moreover, in particular scenarios the computation can be reused; if \( \ell \) is a multiple of two, the \( C_{\ell/2} \cdot X_{t+\ell/2} \) operation may be used for forward and backward updates when performing the \( X_t \) and \( X_{t+\ell} \) computation. These scenarios may lead to a slight reduction of the inner loop instructions and a further improvement in the execution time.

4.2. Floating-Point Operations to Data Cache Access Ratio

For the purpose of comparing the classical and the Semi-stencil algorithms, we calculate the \( FP/Cache \) ratio for the latter algorithm as follows,

\[
FP/Cache_{semi} = \frac{\text{FloatingPoint Operations}}{\text{Data Cache Accesses}} = \frac{2 \cdot \text{Multiply-Add Instructions}}{X_{t-1} \text{ Loads} + X_t \text{ Loads} + X_t \text{ Stores}}
\]

\[
= \frac{2 \cdot 2 \cdot \text{dim} \cdot \ell + 1}{\left( (\text{dim} - 1) \cdot \ell + 1 \right) + \left( \text{dim} - 1 \right) + \left( \text{dim} \right)} = \frac{4 \cdot \text{dim} \cdot \ell + 1}{\text{dim} \cdot \ell - \ell + 2 \cdot \text{dim}}
\]

(4)
where the total number of floating-point operations remains constant with respect to the classical stencil implementation (see Equation 1). Equation 4 also shows that $X^{t-1}$ loads to the L1 cache have decreased substantially, by almost a factor of 2. Nevertheless, the $X^t$ stores have increased and a new term ($X^t$ loads) has appeared in the equation due to the partial computations applied to the $X$ and $Y$ axes. Reducing the number of loads results in less cycles to compute the inner loop, as well as a lower register pressure. The lower register use means the compiler has the opportunity to perform more aggressive low level optimizations, such as: loop unrolling, software pipelining and software prefetching.

However, as shown in Figure 5, the Semi-stencil algorithm updates two points per iteration leading to double the number of $X^t$ stores. Depending on the architecture, this could result in loss of performance. For instance, in caches with write-allocate policy, a store miss produces a load block action followed by a write-hit, which will produce cache pollution and a pipeline stall. Nowadays, some cache hierarchy architectures implement cache-bypass techniques to address this issue.

Reviewing Equation 4, we see that the new computation structure of the Semi-stencil allows the $\text{FP}/\text{Cache}$ ratio to increase to $\approx 5$ flops per d-cache access for 3D problems. For $\ell$ in the range of 4 to 14, the $\text{FP}/\text{Cache}$ ratio is a factor of between 1.3 and 1.7 times better than the classical stencil implementation, which will have a clear effect on performance.

### 4.3. Head, Body and Tail computations

FD methods require what are called interior points (inside the solution domain) and ghost points (outside the solution domain). The new algorithm structure of Semi-stencil takes this feature into account to preserve the numerical soundness.

To obtain consistent results with the two-phase update on border interior points, the algorithm must be split into three different sections: head, body and tail. The head section updates the first $\ell$ interior points with the rear contributions (forward phase). In the body section, the interior points are updated with back and forth neighbor interior elements (both forward and backward phases are carried out). Finally, in the tail section, the last $\ell$ interior points of the axis are updated with the front contributions (backward phase). Figure 6 shows a 1D execution example of Semi-stencil where the three sections are clearly depicted.

### 4.4. Orthogonal Algorithm

As stated in Section 3, the state-of-the-art in stencil computations has improved in recent years with the publication of several optimization techniques. Some of these can significantly increase the execution performance under specific circumstances. However, most of them can not be combined due to traversing strategy incompatibilities. In other words, some state-of-the-art techniques share a particular data traversal method when the stencil operator is computed, for example: space blocking or space-time blocking based. Therefore, not all combinations of these techniques are feasible.

In contrast to the previous group of optimization algorithms, the Semi-stencil specifies the manner in which the stencil operator is calculated and how data is accessed in the inner loop (see Algorithm 2). This kind of algorithm, which we term a structural-based optimization algorithm, is orthogonal and can easily be implemented without modifying the sweeping order of the computational domain. Thus, the Semi-stencil structure is complementary with traversing optimization algorithms such as Rivera, Time-skewing or Cache-oblivious. At the same time, Semi-stencil can be combined with low level pipeline optimizations such as loop-level transformations, software prefetching or software pipelining.
Fig. 6. Execution example of Semi-stencil algorithm for a 1D problem, where $\ell = 4$. $F$ stands for a forward update and $B$ stands for a backward update. The horizontal axis represents the space domain ($Z = 16$ including ghost points) of the problem. The vertical axis shows the execution time (example completed in 12 steps) for each algorithm section.

Note that Semi-stencil can be applied to any axis of an $n$-dimensional stencil problem, from unit-stride to least-stride dimension. Our recent studies in 3D stencil problems have shown that the Semi-stencil algorithm is most suitable for the least-stride dimensions of 3D problems ($X$ and $Y$ in our Cartesian axes). There are four main reasons for this behavior. First, most modern compilers can take advantage of unit-stride accesses by reusing previous loads and fetching one value per iteration in a steady state. Second, the programmer may easily add some pipeline optimizations that would improve performance for the unit-stride dimension in the same way that Semi-stencil does. Third, current architectures usually sport hardware prefetchers which may especially help to reduce unit-stride latency accesses. And fourth, our memory access model for stencil computations (see the summary in Table II) has shown that a full-axis Semi-stencil algorithm implementation has a slightly higher penalty for $X^t$ loads and $X^t$ stores than a partial Semi-stencil implementation.

Algorithms 4, 5 and 6 show the pseudo-code implementation of Rivera, Time-skewing and Cache-oblivious algorithms respectively with a partial Semi-stencil implementation (detailed in Algorithms 2 and 3). All these implementations are freely available in the micro-benchmark that accompanies the software bundle.
Algorithm XXX: Semi-stencil

Table II. Dataset accesses per stencil implementation in order to compute one point of the domain. The figures show the structure of each stencil while dark boxes and numbers represent the updated points and their computation order respectively. Recall that data reuse for load operation is considered for the unit-stride dimension \((Z)\) through compiler optimizations.

<table>
<thead>
<tr>
<th>Dataset accesses</th>
<th>Classical Stencil ((Z, X \text{ and } Y \text{ axis}))</th>
<th>Full Semi-stencil ((X \text{ and } Y \text{ axis}))</th>
<th>Partial Semi-stencil ((X \text{ and } Y \text{ axis}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure of the stencil</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>(X^{t-1}) loads</td>
<td>(2 \times (\text{dim} - 1) \times \ell + 1)</td>
<td>((\text{dim} - 1) \times \ell + 1)</td>
<td>((\text{dim} - 1) \times \ell + 1)</td>
</tr>
<tr>
<td>(X^t) loads</td>
<td>0</td>
<td>(\text{dim})</td>
<td>(\text{dim} - 1)</td>
</tr>
<tr>
<td>(X^t) stores</td>
<td>1</td>
<td>(\text{dim} + 1)</td>
<td>(\text{dim})</td>
</tr>
</tbody>
</table>

5. PERFORMANCE MODELING

The performance characterization of a kernel code is difficult but it may help us to forecast the maximum expected performance for a given architecture. Nevertheless, the modeling approach has a drawback: it relies heavily on the ability to capture an algorithm’s behavior in an accurate fashion, which is not simple to achieve. Thus, we must focus on effectively modeling how a kernel performs and behaves on different platforms.

For stencil computations, a number of suitable performance models for finite difference problems may be found in the literature. The next subsections provide details of Roofline, a generic performance model for any scientific kernel along with a stencil computation specific model for multi-level cache architectures. In this regard, we also acknowledge the works [Treibig and Hager 2009] and [Treibig et al. 2009]. Although these models are not covered in this work, they devised a simplified approach to understand the performance of bandwidth-limited loop kernels which might predict stencil computation behavior.

5.1. The Roofline Model

An easy-to-understand and visual performance model is proposed through the Roofline model [Williams et al. 2008] in order to improve parallel software and hardware for floating point computations by relating processor performance to off-chip memory traffic. To achieve such a goal, the term operational intensity (OI) is defined as the operations carried per byte of memory traffic (Flops/Byte). The total bytes accessed are those that hit the main memory after being filtered by the cache hierarchy. Therefore, OI measures traffic between the caches and memory rather than between the processor and the caches. Thus, OI quantifies the DRAM bandwidth needed by a kernel on a particular architecture.

As shown in Figure 7, floating-point performance, OI and memory performance can be tied together in a two-dimensional graph with this model. The horizontal line shows peak floating-point performance of the given architecture, and therefore no kernel can exceed this since it is a hardware limit. The X axis depicts the GFlops/byte, whereas
Algorithm 2 The Semi-stencil algorithm pseudo-code (Part 1). $\mathcal{X}^t$ is the space domain for time-step $t$, where $Z$, $X$ and $Y$ are the dimensions of the datasets including ghost points. $\ell$ denotes the neighbors used for the central point contribution. $C_{Z \ell \ldots X \ell}$ and $C_{Y \ell \ldots Y \ell}$ are the spatial discretization coefficients for each direction and $C_0$ for the self-contribution. $z_a, z_e, x_a, x_e, y_a$ and $y_e$ denote the area of $\mathcal{X}^t$ where the stencil operator is computed.

1: procedure SEMI-Stencil($\mathcal{X}^t$, $\mathcal{X}^{t-1}$, $\ell$, $z_a, z_e, x_a, x_e, y_a, y_e$) \hfill \text-comment{Head Y / Forward Y}
2: for $k = y_a - \ell$ to MIN($y_e, y_e - \ell$) do \hfill \text-comment{Head Y / Forward Y}
3: for $i = z_a$ to $z_e$ do \hfill \text-comment{Body Y}
4: $\mathcal{X}_{i,j,k}^{t+1} = C_{Y \ell} * X_{i,j,k}^{t-1} + C_{Y \ell-1} * \mathcal{X}_{i,j,k+1}^{t-1} + \ldots + C_{Y 1} * \mathcal{X}_{i,j,k+\ell-1}^{t-1}$ \endfor
5: endfor
6: endfor
7: for $j = x_a - \ell$ to MIN($x_e, x_e - \ell$) do \hfill \text-comment{Head X / Forward X}
8: for $i = z_a$ to $z_e$ do \hfill \text-comment{Body X}
9: $\mathcal{X}_{i,j,k}^{t+1} = C_{X \ell} * \mathcal{X}_{i,j,k}^{t-1} + C_{X \ell-1} * \mathcal{X}_{i,j+1,k}^{t-1} + \ldots + C_{X 1} * \mathcal{X}_{i,j+\ell-1,k}^{t-1}$ \endfor
10: endfor
11: endfor
12: for $j = x_a - \ell$ to MAX($x_e - \ell, x_a$) to $x_e$ do \hfill \text-comment{Backward X,Y / Forward X,Y}
13: for $i = z_a$ to $z_e$ do \hfill \text-comment{Body X,Y}
14: $\mathcal{X}_{i,j,k}^{t+1} = C_{0} * \mathcal{X}_{i,j,k}^{t-1}$ \hfill \text-comment{Tail X}
15: + $C_{Z 1} * (\mathcal{X}_{i,j-1,k}^{t-1} + \mathcal{X}_{i,j+1,k}^{t-1}) + \ldots + C_{Z \ell} * (\mathcal{X}_{i,j-\ell,k}^{t-1} + \mathcal{X}_{i,j+\ell,k}^{t-1})$
16: + $C_{X 1} * \mathcal{X}_{i,j+1,k}^{t-1} + \ldots + C_{X \ell} * \mathcal{X}_{i,j+\ell-1,k}^{t-1}$
17: + $C_{Y 1} * \mathcal{X}_{i,j,k+1}^{t-1} + \ldots + C_{Y \ell} * \mathcal{X}_{i,j,k+\ell}^{t-1}$ \endfor
18: endfor
19: for $j = \text{MAX}(x_a - \ell, x_a)$ to $x_e$ do \hfill \text-comment{Backward X,Y / Forward Y}
20: for $i = z_a$ to $z_e$ do \hfill \text-comment{Tail X}
21: $\mathcal{X}_{i,j,k}^{t+1} = C_{0} * \mathcal{X}_{i,j,k}^{t-1}$ \hfill \text-comment{Tail X}
22: + $C_{Z 1} * (\mathcal{X}_{i,j-1,k}^{t-1} + \mathcal{X}_{i,j+1,k}^{t-1}) + \ldots + C_{Z \ell} * (\mathcal{X}_{i,j-\ell,k}^{t-1} + \mathcal{X}_{i,j+\ell,k}^{t-1})$
23: + $C_{X 1} * \mathcal{X}_{i,j+1,k}^{t-1} + \ldots + C_{X \ell} * \mathcal{X}_{i,j+\ell-1,k}^{t-1}$
24: + $C_{Y 1} * \mathcal{X}_{i,j,k+1}^{t-1} + \ldots + C_{Y \ell} * \mathcal{X}_{i,j,k+\ell}^{t-1}$ \endfor
25: endfor
26: endfor
27: endfor
28: \hfill \text-comment{Continues in part 2}

the Y axis the GFlops/second. The 45° angle of the graph represents the bytes per second metric, the ratio of (GFlops/second)/(GFlops/byte). As a consequence, a second line can be plotted, which gives the highest floating-point performance that the memory
Algorithm 3  Continuation of the Semi-stencil algorithm pseudo-code (Part 2). This algorithm must be called as follows to compute the entire data set: \( \text{SEMI-STENCIL}(\lambda^t, \lambda^{t-1}, \ell, \ell, Z - \ell, \ell, X - \ell, \ell, Y - \ell) \). Note that, in this work, the coefficients are considered constant.

\[
\begin{align*}
\text{Algorithm 3} & : \text{Semi-stencil algorithm pseudo-code (Part 2).} \\
29: & \text{for } k = \max(y_e - \ell, y_e) \text{ to } y_e \text{ do} \\
30: & \text{for } j = x_s - \ell \text{ to } \min(x_s, x_e - \ell) \text{ do} \quad \text{\( \text{Head} \ X / \text{Forward} \ X \)} \\
31: & \text{for } i = z_s \text{ to } z_e \text{ do} \\
32: & \quad \lambda_{i,j+\ell,k}^t = C_X \lambda_{i,j,k}^{t-1} + C_{Xt} \lambda_{i,j+1,k}^{t-1} + \ldots + C_{Yt^*} \lambda_{i,j+\ell-1,k}^{t-1} \\
33: & \qquad \text{end for} \\
34: & \text{end for} \\
35: & \text{for } j = x_s \text{ to } x_e - \ell \text{ do} \\
36: & \text{for } i = z_s \text{ to } z_e \text{ do} \quad \text{\( \text{Backward} \ X,Y / \text{Forward} \ X \)} \\
37: & \quad \lambda_{i,j,k}^t = C_0 \lambda_{i,j,k}^{t-1} + C_{Zt} \lambda_{i-1,j,k}^{t-1} + \ldots + C_{Zt^*} \lambda_{i-\ell,j,k}^{t-1} \\
38: & \quad \text{end for} \\
39: & \text{end for} \\
40: & \text{for } j = \max(x_e - \ell, x_e) \text{ to } x_e \text{ do} \quad \text{\( \text{Tail} \ X \)} \\
41: & \text{for } i = z_s \text{ to } z_e \text{ do} \quad \text{\( \text{Backward} \ X,Y \)} \\
42: & \quad \lambda_{i,j,k}^t = C_0 \lambda_{i,j,k}^{t-1} + C_{Zt} \lambda_{i-1,j,k}^{t-1} + \ldots + C_{Zt^*} \lambda_{i-\ell,j,k}^{t-1} \\
43: & \quad \text{end for} \\
44: & \text{end for} \\
45: & \text{end for} \\
46: & \text{end for} \\
47: & \text{end procedure}
\]

Algorithm 4  Pseudo-code for the space tiling implementation of the Semi-stencil algorithm. \( TI \) and \( TJ \) define the \( TI \times TJ \) block size. The less-stride dimension (\( Y \)) is left uncut.

\[
\begin{align*}
\text{Algorithm 4} & : \text{Semi-stencil algorithm pseudo-code (Part 3).} \\
1: & \quad \text{procedure RIVERA}(\lambda^t, \lambda^{t-1}, Z, X, Y, TI, TJ, \text{timesteps}, \ell) \\
2: & \quad \text{for } t = 0 \text{ to } \text{timesteps} \text{ do} \quad \text{\( \text{Compute blocks of size} \ TI \times TJ \)} \\
3: & \quad \text{for } jj = \ell \text{ to } X, TJ \text{ do} \\
4: & \quad \quad \text{for } ii = \ell \text{ to } Z, TI \text{ do} \\
5: & \quad \quad \quad \text{SEMI-STENCIL}(\lambda^t, \lambda^{t-1}, \ell, \ell, \min(ii + TI, Z - \ell), \\
6: & \quad \quad \quad \text{MIN}(jj + TJ, X - \ell, \ell, Y - \ell) \\
7: & \quad \quad \text{end for} \\
8: & \quad \text{end for} \\
9: & \text{end procedure}
\]

system can achieve for a given OI. Through these two lines, the model is completely roof-limited and therefore the performance limit can be obtained with the following formula:
Algorithm 5 Pseudo-code for the time-skewing implementation of the Semi-stencil algorithm.

1: procedure TIME-SKEWING($\mathbf{X}^t, \mathbf{X}^{t-1}$, $Z$, $X$, $Y$, $TI$, $TJ$, $TK$, timesteps, $\ell$)
2: for $kk = \ell$ to $Y - \ell$, $TK$ do
3: negY slope = $\ell$
4: posY slope = $-\ell$
5: $tyy = \text{MIN}(TK, Y - kk - \ell)$
6: if $kk = \ell$ then
7: negY slope = 0
8: end if
9: if $kk = Y - tyy - \ell$ then
10: posY slope = 0
11: end if
12: for $jj = \ell$ to $X - \ell$, $TJ$ do
13: negX slope = $\ell$
14: posX slope = $-\ell$
15: $txx = \text{MIN}(TJ, X - jj - \ell)$
16: if $jj = \ell$ then
17: negX slope = 0
18: end if
19: if $jj = X - txx - \ell$ then
20: posX slope = 0
21: end if
22: for $ii = \ell$ to $Z - \ell$, $TI$ do
23: negZ slope = $\ell$
24: posX slope = $-\ell$
25: $tzz = \text{MIN}(TI, Z - ii - \ell)$
26: if $ii = \ell$ then
27: negZ slope = 0
28: end if
29: if $ii = Z - tzz - \ell$ then
30: posX slope = 0
31: end if
32: $pX^t = \mathbf{X}^t$
33: $pX^{t-1} = \mathbf{X}^{t-1}$
34: for $t = 0$ to timesteps do /* Compute stencil on 3D trapezoid */
35: $\text{SEMI-STALLZIL}(pX^t, pX^{t-1}, \ell,$
36: MAX($\ell$, $ii - t \times \text{negZ slope}$), MAX($\ell$, $ii + tzz \times \text{posX slope}$),
37: MAX($\ell$, $jj - t \times \text{negX slope}$), MAX($\ell$, $jj + txx \times \text{posX slope}$),
38: MAX($\ell$, $kk - t \times \text{negY slope}$), MAX($\ell$, $kk + tyy \times \text{posY slope}$))
39: $\text{tmpPtr} = pX^t$
40: $pX^t = pX^{t-1}$
41: $pX^{t-1} = \text{tmpPtr}$
42: end for
43: end for
44: end for
45: end for
46: end procedure

Attainable GFlops/second = $\text{MIN}(\text{Peak Floating Point Performance},$
$\text{Peak Memory Bandwidth} \times \text{Operational Intensity})$
Algorithm 6 Pseudo-code for the cache-oblivious implementation of the Semi-stencil algorithm. The trapezoid to compute is defined by \( \tau(t_0, t_1, z_0, d_0, z_1, d_1, x_0, d_x, x_1, d_x, y_0, d_y, y_1, d_y) \).

1: procedure Cache-oblivious\\(\(X[t, t-1]\), Z, X, Y, Cutoff, ds, t_0, t_1, \\
z_0, d_0, z_1, d_1, x_0, d_x, x_1, d_x, y_0, d_y, y_1, d_y)\\n2: \begin{align*}\\n3: & dt = t_1 - t_0\\n4: & wz = ((z_1 - z_0) + (d_1 - d_0) * dt * 0.5) / 2 \quad \text{//Compute 3D trapezoid volume}\\n5: & wx = ((x_1 - x_0) + (d_x - d_x) * dt * 0.5) / 2\\n6: & wy = ((y_1 - y_0) + (d_y - d_y) * dt * 0.5) / 2\\n7: & \text{if } dt = 1 \text{ or vol < cutoff then}\\n8: & \quad \text{for } t = t_0 \text{ to } t_1 \text{ do} \quad \text{//Base case: compute stencil on 3D trapezoid}\\n9: & """"\\n10: & \quad \text{else if } dt > 1 \text{ then}\\n11: & \quad \quad \text{if } wxy \geq 2 * ds * dt \text{ then} \quad \text{//Space Y-cut}\\n12: & \quad \quad \quad \text{ym} = ((x_0 + y_0) + (2 * d_0 + d_y) * dt) / 4\\n13: & \quad \quad \text{Cache-oblivious}(X[t, t-1], Z, X, Y, Cutoff, ds, t_0, t_1, \\
z_0, d_0, z_1, d_1, x_0, d_x, x_1, d_x, y_0, d_y, y_1, d_y)\\n14: & \quad \quad \text{else if } wxy \geq 2 * ds * dt \text{ then} \quad \text{//Space X-cut}\\n15: & \quad \quad \quad \text{xm} = ((x_0 + x_1) + (2 * d_x + d_x) * dt) / 4\\n16: & \quad \quad \text{Cache-oblivious}(X[t, t-1], Z, X, Y, Cutoff, ds, t_0, t_1, \\
z_0, d_0, z_1, d_1, x_0, d_x, x_1, d_x, y_0, d_y, y_1, d_y)\\n17: & \quad \quad \text{else} \quad \text{//Time cut}\\n18: & \quad \quad \quad s = dt / 2\\n19: & \quad \quad \text{Cache-oblivious}(X[t, t-1], Z, X, Y, Cutoff, ds, t_0, t_0 + s, \\
z_0, d_0, z_1, d_1, x_0, d_x, x_1, d_x, y_0, d_y, y_1, d_y)\\n20: & \quad \text{Cache-oblivious}(X[t, t-1], Z, X, Y, Cutoff, ds, t_0 + s, t_1, \\
z_0 + d_0 * s, d_0, z_1 + d_1 * s, d_1, \\
x_0 + d_x * s, d_x, x_1 + d_x * s, d_x, \\
y_0 + d_y * s, d_y, y_1 + d_y * s, d_y)\\n21: & \quad \text{end if}\\n22: & \text{end if}\\n23: \end{align*}\\n
The point where the two lines intersect is the peak computational performance and peak memory bandwidth. It is important to bear in mind that the Roofline limits are fixed for each architecture and do not depend on the kernel being characterized.

A point on the X axis can be found for any scientific kernel through its OI. Drawing a vertical line through that point, the maximum attainable performance for the tested architecture can be estimated from the intersection of this line and the roofline.

The Roofline sets an upper bound on a kernel’s performance depending on its OI. If the vertical projection of the line intersects the flat part of the roof, the kernel performance is compute bound, whereas it is memory bound if it intersects the sloped part of the roof. In addition, the model shows what the authors call the ridge point, where the horizontal and the 45° roof lines meet. The x-coordinate of this point provides an estimate of the minimum OI required by a kernel to achieve the maximum performance on an architecture. A ridge point shifted to the right implies that only kernels with a high
OI can achieve the maximum performance. On the other hand, a ridge point shifted to the left on the axis means that most of the kernels can potentially obtain the maximum performance. Hence, the ridge point is related to the difficulty for programmers and compilers to achieve the peak performance in a given architecture.

5.2. Multi-Level Cache Model

In order to predict properly the behavior of stencil computations in mainstream architectures, a flexible and accurate model for a wide range of stencil computations has been proposed [de la Cruz and Araya-Polo 2011]. In this work, the authors claim that the multi-level cache hierarchy has to be considered to forecast the behavior of stencil computations.

Only memory transactions between the CPU and memory subsystem are considered, and computation bottlenecks are assumed to be negligible due to the memory bound aspect of stencil computation [Datta et al. 2009; de la Cruz et al. 2009]. Finally, interference between data and instructions at cache level is not taken into account.

The model is composed of a base model, a read and write miss policy; and additional special features such as a hardware prefetching predictor mechanism.

5.2.1. Base Model. Consider a grid of $I \times J \times K = N^3$ elements (each of size word) which is traversed with a naive implementation, where $I$ is the unit-stride dimension and $J$ and $K$ are the least-stride dimensions excluding ghost points. A 3D stencil computation of order $\ell$ requires $2 \times \ell + 1$ Z-X input planes in order to compute one Z-X output plane [de la Cruz et al. 2009]. The size of these planes (in words) differs depending on the operation being performed (read or write), and therefore altering the memory requirements to compute one $k$ iteration of the sweep (outermost loop of Algorithm 1). Let $P$ be the number of planes and $S$ the plane size in words; the following terms for read and write operations are obtained,

\begin{align}
P_{\text{read}} &= 2 \times \ell + 1 \\
S_{\text{read}} &= (I + 2 \times \ell) \times (J + 2 \times \ell) \\
S_{\text{write}} &= I \times J \\
S_{\text{writeback}} &= P_{\text{read}} + P_{\text{write}} \\
S_{\text{total}} &= P_{\text{read}} \times S_{\text{read}} + P_{\text{write}} \times S_{\text{write}}
\end{align}

The stencil computation involves back and forth dataset transfers, which may produce conflicts depending on cache policies (write-through or write-back). For instance, during a write operation with write-back policy, a cache line allocation is produced (write allocate), which shares memory resources among input data and leads to cache pollution.
Algorithm XXX: Semi-stencil

Given an architecture with a memory hierarchy of \( n \) cache levels, it can be assumed that the total time \( (T_{\text{total}}) \) required to compute a stencil is based on the following formula,

\[
T_{\text{total}} = T_{L_1} + \cdots + T_{L_i} + \cdots + T_{L_n} + T_{\text{Memory}}
\]

(6)

where \( T_{L_i} \) and \( T_{\text{Memory}} \) are the times to access data in \( L_i \) cache level and main memory respectively. In order to estimate \( T_{L_i} \) and \( T_{\text{Memory}} \), the number of cache line hits \( (\text{Hits}_{L_i}) \) and cache line misses \( (\text{Misses}_{L_i}) \) must be estimated. Depending on the level, the way to compute hits and misses differs due to hardware policies. To deal with this aspect, three memory hierarchy groups are established to calculate hits and misses: first \( (L_1) \), intermediate \( (L_2 \text{ to } L_n) \) and last \( (\text{Memory}) \). When the CPU issues a word load instruction, the data, if present, is brought from the closest cache level \( (L_1) \); otherwise, a miss is flagged and passed to the next level of the hierarchy, which will finally fetch an entire cache line containing the data. The remaining levels also follow this mechanism.

Several values are required in the internal loop to compute a single point of the domain: grid points \( (X^{1-1}) \), weights \( (C_{Z,X,Y,0}) \), and indices \( (i, j, k) \). In the register bank, grid points and weights are kept in Floating-Point Registers \( (\text{FPR}) \), while indices use General-Purpose Registers \( (\text{GPR}) \). Leaving aside optimizations, grid points must be fetched in every sweep of the loop, whereas weights and indices might be partially reused. However, note that depending on: the order of the stencil \( (\ell) \), the problem dimension \( (\text{dim}) \) and the available registers \( (\text{FPR}_{\text{free}} \text{ and } \text{GPR}_{\text{free}}) \), data reuse becomes burdensome, leading to register spilling and increased CPU-L1 traffic. To obtain such information for the first level, the register pressure \( (\text{Reg}_{\text{grid}}, \text{Reg}_{\text{weight}} \text{ and } \text{Reg}_{\text{index}}) \) required to compute the whole domain \( (I \times J \times K) \) is estimated by considering the available registers,

\[
\begin{align*}
T_{L_1}^{\text{word}} &= \frac{\text{word}}{Bw_{L_1}^{\text{read}}} \\
\text{Hits}_{L_1}^{\text{word}} &= (\text{Reg}_{\text{grid}} + \text{Reg}_{\text{weight}} + \text{Reg}_{\text{index}}) \ast I \ast J \ast K - \text{Misses}_{L_1}^{\text{word}} \\
T_{L_1} &= \text{Hits}_{L_1}^{\text{word}} \ast T_{L_1}^{\text{word}}
\end{align*}
\]

(7)

where the number of hits \( (\text{Hits}_{L_1}^{\text{word}}) \) is estimated by the difference between loads issued by the CPU and misses triggered by the first hierarchy \( (\text{Misses}_{L_1}^{\text{word}}) \). Let word be the data granularity and \( Bw_{L_1}^{\text{read}} \) the \( L_1 \) read bandwidth; the \( L_1 \) cost \( (T_{L_1}) \) is calculated as the cost of transferring a word from \( L_1 \) to CPU \( (T_{L_1}^{\text{word}}) \) times the hits in \( L_1 \) \( (\text{Hits}_{L_1}^{\text{word}}) \). Following a similar scheme, the cost for any intermediate cache level can be approximated as,

\[
\begin{align*}
T_{L_i}^{\text{cacheline}} &= \frac{\text{cacheline}}{Bw_{L_i}^{\text{read}}} \\
\text{Hits}_{L_i}^{\text{cacheline}} &= \text{Misses}_{L_i-1}^{\text{cacheline}} - \text{Misses}_{L_i}^{\text{cacheline}} \\
T_{L_i} &= \text{Hits}_{L_i}^{\text{cacheline}} \ast T_{L_i}^{\text{cacheline}}
\end{align*}
\]

(8)

where the number of hits \( (\text{Hits}_{L_i}^{\text{cacheline}}) \) depends on misses issued in the previous cache level \( (L_i - 1) \) and misses flagged in the current one \( (L_i) \). Then, the time cost in level \( i \) can be estimated straightforwardly as the time required to transfer a cache line (data granularity) from level \( i \) \( (T_{L_i}^{\text{cacheline}}) \) times the number of transfers performed \( (\text{Hits}_{L_i}^{\text{cacheline}}) \). Finally, the accessing data cost for the last level (main memory) can be similarly computed. Given that data is allocated in the last hierarchy level, any miss issued at the previous level \( (L_n) \) will necessarily generate a hit in main memory \( (\text{Hits}_{L_n}^{\text{cacheline}} = \text{Misses}_{L_n}^{\text{cacheline}}) \). Thus, the access cost \( (T_{\text{Memory}}) \) is directly based on

5.2.2. Read Misses. The model covers four cases of miss estimation (see Figure 8) where these cases depend on three distinct types of cache misses (compulsory, capacity and conflict misses) [Temam et al. 1994]. Ordered from lowest to highest penalty, the four cases proposed by the model are:

1. The best possible scenario (lower bound) is obtained when all the required Z-X planes ($O_{total}^{writepolicy}$) for one $k$ iteration fit completely in the $L_i$ cache. Thus, one plane per $k$ iteration is fetched. In this case, only compulsory misses (cold-start misses) are flagged. Cold misses arise due to loads of new data which are not in the hierarchy. Let $II$, $JJ$ and $KK$ be the extended dimensions (including ghost points) of $I$, $J$ and $K$, and let $W$ be the words per cache line ($W = \text{cacheline/word}$); the misses of $L_i$ are,

$$Misses_{L_i}^{cacheLine} = \left\lceil \frac{II}{W} \right\rceil \times JJ \times KK \times nplanes_{L_i} \text{ where } nplanes_{L_i} = 1. \quad (9)$$

2. The second scenario occurs when all required planes do not fit in $L_i$, leading to capacity misses. However, the $k$-central plane, with a higher temporal reuse than the rest (1 vs $2 \times \ell + 1$), is probably not replaced from the cache due to conflict misses. Hence, it might be partially reused in following iterations, giving $nplanes_{L_i} = P_{read} - 1$ for Equation 9.

3. On the third scenario, it is also assumed that all planes do not fit in $L_i$. Nevertheless, in this case the $k$-central plane overwhelms a significant part of the $L_i$ cache (i.e., half the capacity) reducing the possibility of temporal reuse. Therefore, $nplanes_{L_i} = P_{read}$ planes must be read when traversing the $Y$ axis of the stencil loop.

4. Finally, in the worst scenario (upper bound) neither the planes nor the columns of the $k$-central plane fit in the cache. Due to capacity and conflict misses, all planes must be read at every $k$ iteration ($P_{read}$), whereas the columns of the $k$-central plane must be read at each $j$ iteration ($P_{read} - 1$). This scenario gives $nplanes_{L_i} = 2 \times P_{read} - 1$.

5.2.3. Prefetching. The prefetching modeling is complex; in particular recognizing if a prefetched stream flags a miss. In this model a simple approach is devised. Misses of the model are divided into two groups: prefetched and non-prefetched. First, the $X$-$Y$ planes that can be prefetched ($nplanes_{Li}^{stream}$) and not prefetched ($nplanes_{Li}^{Nstream}$) in cache level $i$ are calculated,
Algorithm XXX: Semi-stencil

\[ n_{\text{planes}}^{\text{Stream}}_{Li} = \max(n_{\text{planes}}_{Li} - \text{pref}_{Li}, 0) \quad n_{\text{planes}}^{\text{NStream}}_{Li} = n_{\text{planes}}_{Li} - n_{\text{planes}}^{\text{Stream}}_{Li} \]

\[ T_{Li} = \text{Hits}_{Li}^{\text{cacheline}} \ast T_{Li}^{\text{cacheline}} + \text{Hits}_{Li}^{\text{Stream}} \ast T_{Li}^{\text{Stream cacheline}} \]

where \( \text{pref}_{Li} \) refers to the number of stream channels supported by the current hierarchy level \( i \). Second, prefetched and non-prefetched cache line misses within planes are computed using Equation 9 and its specific \( n_{\text{planes}}_{Li} \) value. Finally, \( T_{Li} \) is computed by adding the contribution of the two types of planes using their respective hit ratio and transfer cost to the base model rule for \( Li \). In order to compute the specific bandwidths for streamed and non-streamed planes, a modified version of the STREAM2 [de la Cruz and Araya-Polo 2011] benchmark is used.

6. EXPERIMENTS

In this section, experimental results of the Semi-stencil algorithm are presented. First, we introduce the Stencil Probe, an existing parameterized micro-benchmark developed to test stencil codes. Second, the platforms used to conduct the experiments are described, and the tools employed to obtain the profiling information are introduced. Then, we prove that our \( FP/Cache \) model is consistent for classical and Semi-stencil codes. Finally, the performance results are shown and evaluated for each testbed architecture, including some multi-core scalability results.

6.1. Stencil Micro-benchmark

In order to evaluate the performance of our implementations, the Stencil Probe [Kamil et al. 2005], a compact and self-contained serial micro-benchmark has been used. This micro-benchmark was developed to explore the behavior of 3D stencil-based computations without having to modify any application code. Furthermore, the Stencil Probe has been extended with more features to evaluate the new algorithms.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem sizes</td>
<td>512 \times 512 \times 512 (in double-precision floating-point format)</td>
</tr>
<tr>
<td>Stencil sizes (( \ell ))</td>
<td>1, 2, 4, 7 and 14 (7, 13, 25, 43 and 85-point respectively)</td>
</tr>
<tr>
<td>Time-steps</td>
<td>1, 2, 4 and 8</td>
</tr>
<tr>
<td>Algorithms</td>
<td>{Naive, Rivera, Time-skew, Cache-oblivious} \times {Classical, Semi-stencil}</td>
</tr>
<tr>
<td>Block sizes</td>
<td>{16, 32, 64, 128, 256, 512} \times {16, 32, 64, 128, 256, 512} \times 512</td>
</tr>
<tr>
<td>CUTOFF</td>
<td>{256, 512, 1k, 2k, 4k, 8k, 16k, 32k}</td>
</tr>
<tr>
<td>Inner loop opts.</td>
<td>{loop fusion (1 loop), loop fission (2 loops), loop fission (3 loops)}</td>
</tr>
</tbody>
</table>

We made four changes to the micro-benchmark. First, two Semi-stencil algorithms have been introduced: a full-axis (Z, X and Y) and a partial-axis (X and Y) version. Second, as well as the original stencil shape, four new cross-shape stencils have been included for each algorithmic implementation. The currently available stencil sizes range from low-order to high-order \( \ell = \{1, 2, 4, 7, 14\} \). Third, some straightforward pipeline optimizations related to the inner loop of the stencil computation have been coded. These optimizations include loop fusion and fission in two and three segments. Finally, the Stencil Probe micro-benchmark has been extended to include the parallel implementation of each algorithm in order to analyze their multi-core scalability. The shared-memory OpenMP API is used as the parallel programming paradigm.
It is worth noting that, with the new benchmark features, the parameter space to be explored becomes quite large. In order to bound the search area for the experiments, we set some parameters as either constants or within a reasonable range of values (e.g., problem size and blocking parameters). Table III shows a summary of all the parameters used in the micro-benchmark.

6.2. Testbed Architectures

The following leading platforms, available in the PRACE\(^1\) project, were used to carry out our study for serial runs (detailed platform characteristics are presented in Table IV):

— **Intel Nehalem**: Juropa supercomputer, hosted at Jülich Research Center (Germany), sports 4416 Intel Xeon Nehalem-EP processors.

— **IBM POWER6**: Sara supercomputing center (Netherlands) hosts Huygens, an IBM pSeries 575, clustered SMP system. The system consists of 104 nodes, each with 16 dual core POWER6 processors.

— **IBM Blue Gene/P**: Jugene supercomputer, hosted at Jülich Research Center, is based on the Blue Gene/P processor. Jugene reaches 1 PFlops of peak performance with 294912 PowerPC 450 cores.

— **AMD Opteron**: Louhi is a Cray XT4/XT5 supercomputer with 1012 XT4 and 852 XT5 compute nodes, which is hosted at IT Center for Science in Finland.

<table>
<thead>
<tr>
<th>System name</th>
<th>Juropa</th>
<th>Huygens</th>
<th>Jugene</th>
<th>Louhi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Intel Xeon Core i7 (Nehalem-EP)</td>
<td>IBM Power6</td>
<td>IBM BlueGene/P PowerPC 450</td>
<td>AMD Opteron 2356 (Barcelona)</td>
</tr>
<tr>
<td>Chips × Cores</td>
<td>2 × 4</td>
<td>16 × 2</td>
<td>1 × 4</td>
<td>1 × 4</td>
</tr>
<tr>
<td>Clock</td>
<td>2.93 GHz</td>
<td>4.7 GHz</td>
<td>850 MHz</td>
<td>2.3 GHz</td>
</tr>
<tr>
<td>SP GFlops(^†)</td>
<td>93.76 (SSE)</td>
<td>75.2 (VMX)</td>
<td>13.6 (SIMD)</td>
<td>73.6 (SSE)</td>
</tr>
<tr>
<td>DP GFlops(^†)</td>
<td>46.88 (SSE)</td>
<td>37.6</td>
<td>13.6 (SIMD)</td>
<td>36.8 (SSE)</td>
</tr>
<tr>
<td>L1 Cache (D+I)</td>
<td>32 kB + 32 kB</td>
<td>64 kB + 64 kB</td>
<td>32 kB + 32 kB</td>
<td>64 kB + 64 kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 kB per core</td>
<td>4096 kB per core</td>
<td>1920 B per core</td>
<td>512 kB per core</td>
</tr>
<tr>
<td>L3 Cache (shared)</td>
<td>8 MB (inclusive)</td>
<td>32 MB (victim)</td>
<td>2 × 4 MB</td>
<td>2 MB (victim)</td>
</tr>
<tr>
<td>Main memory</td>
<td>24 GB</td>
<td>128 GB</td>
<td>2 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>32 GB/s</td>
<td>51.2 GB/s</td>
<td>13.6 GB/s</td>
<td>42.7 GB/s</td>
</tr>
<tr>
<td>Watts × hour</td>
<td>95</td>
<td>100</td>
<td>39</td>
<td>75</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel/GCC (v11.1/v4.1.2)</td>
<td>IBM XL (v12.1/v10.01)</td>
<td>IBM XL (v9.0/v11.01)</td>
<td>Portland/GCC (v10.2/v4.1.2)</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3, -fast,</td>
<td>-O3, -O4, -O5,</td>
<td>-O3, -O4, -O5,</td>
<td>-O3,</td>
</tr>
<tr>
<td></td>
<td>-mcpu=core2,</td>
<td>-qarch=power6</td>
<td>-qarch=450d</td>
<td>-mcpu=opteron</td>
</tr>
</tbody>
</table>

Two sets of data have been collected for evaluation from these platforms: execution times and hardware counters. The former allows to compare the performance of our implementations with the classical versions; while the latter gives an accurate profiling picture of the underlying events, such as the CPU pipeline resource usage and the cache hierarchy transfers. Some of the counters measured are: floating-point operations, total cycles & instructions, load & store instructions issued, and cache misses, hits & accesses for L1, L2 and L3 levels. These counters permit measurement of some interesting metrics such as: GFlops, \(FP/\text{Cache}\) ratio and actual memory traffic, among others. The results of these metrics are presented along with the \(FP/\text{Cache}\) models in Subsection 6.3. Subsection 6.5 shows a comparison of the performance results.

\(^1\)Partnership for Advanced Computing in Europe - http://www.prace-project.eu/
Algorithm XXX: Semi-stencil

In order to gather the hardware counter profiling, the following tools and frameworks have been deployed in the testbed systems:

— **PapiEx (PAPI)**: PapiEx is a performance analysis tool designed to measure, transparently and passively, the hardware performance counters of an application using the PAPI [Mucci et al. 1999] framework.

— **LikwidPerfCtr (Likwid)**: LikwidPerfCtr [Treibig et al. 2010] is a lightweight command tool to measure hardware performance counters on Intel and AMD processors using the Linux `msr` module.

— **hpmcount/libhpm (HPCT/HPM)**: hpmcount and libhpm provide hardware metric and resource utilization statistics after application execution. They are developed by IBM to support Power-based systems.

The information gathered through these tools is used as reference data in the validation and evaluation process. However, some steps are necessary to ensure reliable and accurate data. First, tests must be executed a number of times in order to minimize collateral effects from the OS or other artifacts. Second, cache hierarchy must be cleared during each test repetition to avoid cache-line hits due to hot cache effects. And finally, to prevent the pollution of load and store metrics in high order stencils due to register spilling, the same value is used for all coefficient terms ($C_{Z,X,Y}$).

### 6.3. Data Cache Accesses

As Equations 1 and 4 convey, the proposed algorithm performs increasingly well in $FP/Cache$ and data access terms with respect to the classical approach. Figure 9 (top) plots the three models: the classical and the two Semi-stencil (partial-axis $X$-$Y$ and full-axis $Z$-$X$-$Y$). We see that the classical and the Semi-stencil models tend to a factor of $\approx 3$ and 5 flops per d-cache accesses respectively.

To evaluate the robustness of our results, we proceed to compare the model with real performance data. Unfortunately, if the measured data and the projected $FP/Cache$ model are compared, we note a slight difference. The main reason for this is due to interference from other loads and stores not related to $X^\ell$ and $X^\ell-1$ datasets; coefficients terms, loop control variables and register spilling are all involved in this cache traffic increase. The larger the stencil length ($\ell$), the worse the traffic becomes.

In order to achieve an accurate comparison, a new metric must be defined. Taking into account that this issue affects any stencil computation and the artifact should remain constant for a specific stencil size ($\ell$); it would be a good idea to compare data cache accesses between different implementations of the same stencil length. Therefore, results are compared based on the accesses in terms of gain or loss using a reduction factor between $FP/Cache$ metrics. This new metric is defined as follows:

$$\text{AccessesGain} = \frac{(\text{AccessesClassical} - \text{AccessesSemi})}{\text{AccessesClassical}}$$

and is measured as a percentage. A negative value signifies higher data cache traffic, while a positive percentage signals a lower traffic, and, hence, better performance.

Figure 9 (bottom) shows the comparison of data cache accesses for best performance cases using Semi-stencil and classical implementations. Examining the figure, we note that both data groups, the model and the real performance data, are quite close and the experiments follow closely a reduction model curve. In addition, the Semi-stencil algorithm performs poorly for low order stencils ($\ell = \{1,2\}$), where the negative reduction factor represents an increase in traffic. This result is also in line with the theoretical $FP/Cache$ ratio models presented at Figure 9 (top), where both Semi-stencil implementations perform worse than the classical code for $\ell \leq 2$. Nevertheless, as expected, the Semi-stencil algorithm behaves better for medium-high order stencils due to a decrease in data cache traffic.
6.4. Operational Intensity

In order to demonstrate the robustness of the Semi-stencil algorithm, we conducted the Roofline model on the Intel Nehalem and AMD Opteron architectures where memory traffic counters (DRAM bytes accessed) were gathered using likwidperfctr. We can then use the Roofline model to assess how far our empirical results are from the attainable peak performance.

To build the roof part of the model, the DRAM bandwidth and the theoretical peak performance were measured. The DRAM bandwidth measurements were conducted using a modified version of STREAM2 [de la Cruz and Araya-Polo 2011] to estimate stream and non-stream bandwidths. These bandwidths determine how far to the left or right the ridge point is on the X-axis. To obtain the Y-axis ceilings, theoretical peak performance, the processor frequency and the floating-point units on a single core were...
considered. The AMD Opteron and Intel Nehalem cores have two floating-point units, one multiplication and one add unit. Hence, applications must be multiplication/add instruction balanced in order to reach the maximum peak performance. Table V shows the gathered ceiling data on both architectures.

### Table V. Computational and bandwidth ceilings for Roofline model. Notice that only one core is being considered.

<table>
<thead>
<tr>
<th>Roofline ceilings</th>
<th>Intel Nehalem</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak Stream DRAM (GBytes/s)</td>
<td>8.2</td>
<td>4.6</td>
</tr>
<tr>
<td>Peak Non-stream DRAM (GBytes/s)</td>
<td>3.4</td>
<td>1.4</td>
</tr>
<tr>
<td>Peak 2 FP units (GFlops/s)</td>
<td>5.86</td>
<td>4.6</td>
</tr>
<tr>
<td>Peak 1 FP unit (GFlops/s)</td>
<td>2.93</td>
<td>2.3</td>
</tr>
</tbody>
</table>

To gain a better understanding of the Roofline model, the theoretical Operational Intensity (OI) for each stencil algorithm and size were also estimated. As described in section 5.2, three OI groups are devised depending on the stencil 3C's misses: only compulsory, compulsory + capacity and compulsory + capacity + conflict misses. Compulsory misses set the upper bound while the compulsory + capacity + conflict misses set the lower bound in the X-axis of the Roofline model. Table VI shows the estimated OIs using write-back and write-through policies.

### Table VI. Theoretical OIs depending on their 3C's misses. The gray section depicts where Semi-stencil obtains a better ratio compared to the classical. Values in parenthesis are obtained using write-through policy.

<table>
<thead>
<tr>
<th>Stencil sizes</th>
<th>Classical</th>
<th>Semi-stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compulsory</td>
<td>Compulsory + Capacity</td>
</tr>
<tr>
<td>7-point</td>
<td>0.54 (0.81)</td>
<td>0.32 (0.40)</td>
</tr>
<tr>
<td>13-point</td>
<td>1.04 (1.56)</td>
<td>0.44 (0.52)</td>
</tr>
<tr>
<td>25-point</td>
<td>2.04 (3.06)</td>
<td>0.55 (0.61)</td>
</tr>
<tr>
<td>43-point</td>
<td>3.50 (5.31)</td>
<td>0.62 (0.66)</td>
</tr>
<tr>
<td>85-point</td>
<td>7.04 (10.56)</td>
<td>0.68 (0.70)</td>
</tr>
</tbody>
</table>

Furthermore, we calculated the actual computational peak performance to provide a realistic bound on the Roofline ceilings. Such information was obtained by running the Stencil Probe micro-benchmark several times using a small dataset as input ($32^3$), but without clearing the cache (warm cache effect). Then, the fastest execution time was selected as the reference for the stencil algorithm and size. Using this technique, the optimal computational performance was obtained disregarding DRAM accesses and considering only traffic between processor and cache.

Figure 10 combines all the gathered data in the Roofline model for the Intel Nehalem and AMD Opteron architectures to predict the attainable performance. Graphs are on log2-log2 scale, where the Y-axis is the attainable GFlops per second and the X-axis is Flops per DRAM byte accessed. Memory bandwidths (stream and non-stream) and computational peaks (mul/add instruction balanced and imbalanced) determine the optimization regions within the graphs. Remember that the memory measurement is the steady state bandwidth potential of the memory in only one core, and it is not the pin bandwidth of the DRAM chips. The model also suggests, through three regions, which optimizations would be appropriate. The darkest trapezoid suggests working mainly on computational optimizations, the lightest parallelogram proposes trying just memory optimizations, and the region in the corner suggests both types of the above
Fig. 10. Roofline models for Intel Nehalem and AMD Opteron. Computational and bandwidth ceilings bound the theoretical limits for the architecture, whereas the vertical 3C's lines and the horizontal actual peak lines set the limits for each stencil kernel.
Algorithm XXX: Semi-stencil optimizations. The vertical lines on the graphs show bounds for 3C's regions obtained for a 25-point stencil computation, and hence their theoretical OI limits. Additionally, the remaining horizontal lines depict the actual peak performance for each stencil size. Finally, the crosses and circles on the graphs mark the performance achieved by Naive and Naive+Semi-stencil runs respectively. Numbers shown next to the marks represent their stencil size.

The peak double precision performance for the Intel Nehalem is the highest of the two architectures. However, Figure 10 (top) shows that this performance can only be achieved with an OI greater than 0.71 in the best scenario (stream bandwidth) and 1.72 Flops per byte in the worst scenario (non-stream bandwidth). Reviewing the achieved performances, we can note that they are all in the bottom part of the memory bandwidth (45° non-stream line). This behavior may be a consequence of a front side bus limitation and the ineffective work of the snoop filter, which carries coherency traffic and may consume half of the bus bandwidth. Despite the memory bandwidth limitation, the 25 and 43-point stencil computations of Naive+Semi-stencil get close to the actual computational peak thanks to their higher OI.

Figure 10 (bottom) shows the Roofline model for the AMD Opteron architecture. In this architecture the ridge point of the model is at an OI of 1 Flop per byte for stream bandwidth and at 3.28 Flops per byte for non-stream bandwidth. The model clearly shows that the 7 and 13-point results of Naive+Semi-stencil are already limited by peak memory and their performance could only be improved by increasing OI. On the other hand, Naive results fall into the lightest region, indicating not so efficient memory access. Notice that if a vertical line is projected from each Naive result, these lines meet the actual stencil peak performance approximately at the point that diagonal roof part is reached. This behavior confirms the accuracy of the gathered data for the Roofline model. In addition, large stencil size computations (25 and 43-point) of Semi-stencil lead the GFlops/second results despite only showing slightly better OI ratios with respect to their Naive competitors. This metric gap is because OI depends on traffic between the caches and memory, whereas the Semi-stencil algorithm mostly reduces traffic between the processor and caches (see Equation 4).

6.5. Performance Evaluation and Analysis

This work not only claims that the Semi-stencil approach improves the stencil computation performance, but also that it can be combined with others stencil optimization techniques to achieve even higher performance. Due to the sheer number of possible experiments (see Table III) the most relevant results across all testbed platforms are highlighted and analyzed in this section. All computations have been carried out in double precision, since this is used by most scientific codes.

In order to support our claims, the results of our experiments are shown combined in four arrangements. First, the execution times of classical and Semi-stencil versions are compared when the stencil size ($\ell$) and the problem size are changed. Second, hardware counter profilings are given and discussed for each algorithm. Third, a general view of algorithm speed-ups is outlined, and finally, performance results are grouped by varying blocking parameters ($TI \times TJ \times TK$).

Figure 11 (top) clearly shows how efficient the Semi-stencil algorithm performs as $\ell$ and the problem size ($N^3$) are varied. Here, all parameters have been fixed, except for the stencil length. Each pair of classical and Semi-stencil runs was conducted for a particular problem size, ranging from unusually small sizes ($128^3$ and $256^3$) to realistic cases ($512^3$, $768^3$ and $1024^3$). In order to obtain a fair and clear comparison, the reduction in time with respect to the classical algorithm is presented in Figure 11 (bottom).

As expected, Semi-stencil versions perform worse for low order stencil computations ($\ell \leq 2$) where the elapsed time is slightly higher compared to the classical implemen-
Fig. 11. Comparison between classical and Semi-stencil runs using different stencil lengths and problem sizes. Top: elapsed time comparison in seconds. Lower is better. Bottom: reduction of time in percentage. A negative value represents a slower Semi-stencil, whereas a positive value depicts a faster one. Notice how important the problem size is in order to obtain further improvement in low-medium order stencils.

However, for medium and high order stencils ($\ell \geq 4$), our proposed algorithm performs extremely well. Nevertheless, as the problem size grows, the elapsed time difference becomes negligible when $\ell = 2$.

Considering precision and performance requirements, the stencil computation depends on the number of neighbor points. A large number of neighbors will provide high order results at increased computational cost. Given the previous statement and the results shown in Figure 11, a Semi-stencil algorithm is a valid option when high precision is required in large scientific problems.

Figures 12 and 13 show a detailed collection of hardware counters obtained by profiling on all four platforms. Performance statistics shown are: Memory traffic (GBytes), L2/L3 cache misses, GFlops and $FP/Cache$ ratios. Each matrix column represents an algorithm implementation and each matrix row a specific stencil length, $\ell$. Generally, most of these show a better performance when $\ell \geq 2$ (13-point). In terms of the $FP/Cache$ ratio and cache misses, our proposed algorithm performs increasingly well with respect to the classical algorithm.

The stacked bar graphs in Figure 14 show the speed-up of each implementation and execution times in seconds. Data is gathered by platform, time-steps and stencil length. In these graphs, the baseline for each platform and set of parameters represents the Naive code using a classical algorithm (where the speed-up is 1). We see that, space and time-blocking techniques enhance some of the performance results, ranging from 1.05 to 1.3×. However, using the same enhancements with Semi-stencil improves the performance even further, bestowing aggregated speed-ups of up to 1.6× in some cases. The BlueGene/P outperforms the others in terms of speed-up especially for the Semi-stencil algorithm.
In order to show how important a blocking parameter is for space and time-blocking algorithms, Figure 15 collects all the execution runs sorted by blocking sizes. Results are shown for each platform by modifying the blocking parameters in the horizontal axis. As expected, the Rivera and Time-skewing algorithms show a variation of performance when $T_I$, $T_J$, and $T_K$ are changed. In all cases, the best blocking performance is obtained when $T_I$ is left uncut [Rivera and Tseng 2000; Kamil et al. 2005]; the exception, strangely, is BG/P. This may be due to collateral cache-line conflicts. The Naive and Cache-Oblivious algorithms are plotted using their best configurations (tuning internal loop optimizations and CUTOFF parameter).

A summary of the performance results can be found in Table VII. This table shows the execution times and their speed-ups using a stencil size of 4. For three of the four platforms, the Semi-stencil versions are the best implementations. The speed-ups, with respect to the naive code, ranges from $\approx 1.20 \times$ on the Intel Nehalem to $1.60 \times$ for BG/P, which is the most favored architecture. On the Intel Nehalem, the classical Cache-oblivious version slightly outperforms the Semi-stencil version. It is likely that the fast option on the Intel compiler is quite aggressive and optimizes stencil codes rea-
sonably well. On the contrary, the AMD Opteron appears to have some kind of problem with Time-skewing, where the compiler does not appear able to generate optimized binaries for this code. The compiler used on this architecture was GCC v4.1. Finally, on both IBM platforms a substantial gain is observed. First, on BG/P a speed-up of 1.64 is obtained, basically because Semi-stencil helps to mitigate cache-line conflicts in the small L1/L2 caches (32kB and 1920B). Second, on the Power6 architecture, space and time-blocking techniques do not have any substantial effect due to the large memory hierarchy (64kB, 4MB and 32MB) and the sophisticated hardware prefetching system.
Algorithm XXX: Semi-stencil

Fig. 14. Speed-up results with respect to the baseline algorithm (Naive implementation). Numbers shown in bars represent the total time in seconds for each execution (lower is better).
Fig. 15. Execution times for all the algorithmic combinations, where the blocking parameter has been changed to discover the optimum value. The algorithms which do not support blocking have been plotted using the best time obtained.
Algorithm XXX: Semi-stencil

A widespread stencil size ($\ell = 4$, 25-point) and a time-step of 2 have been chosen. Speed-ups with respect to the Naive implementation and without the Semi-stencil strategy are shown in parenthesis and italics respectively.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Intel Nehalem</th>
<th>IBM Power6</th>
<th>IBM BlueGene/P</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>5.01 (1.00)</td>
<td>8.64 (1.00)</td>
<td>9.07 (1.00)</td>
<td>7.42 (1.00)</td>
</tr>
<tr>
<td>Rivera</td>
<td>4.82 (1.04)</td>
<td>8.74 (0.99)</td>
<td>7.46 (1.22)</td>
<td>8.63 (0.86)</td>
</tr>
<tr>
<td>Timeskew</td>
<td>4.43 (1.13)</td>
<td>8.55 (1.01)</td>
<td>7.02 (1.29)</td>
<td>11.04 (0.67)</td>
</tr>
<tr>
<td>Oblivous</td>
<td><strong>3.89 (1.29)</strong></td>
<td>9.05 (0.95)</td>
<td>8.25 (1.10)</td>
<td>5.95 (1.25)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm + Semi-stencil</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive + Semi-stencil</td>
<td>4.27 (1.17)</td>
<td>6.53 (1.32)</td>
<td>6.03 (1.50)</td>
<td>5.64 (1.32)</td>
</tr>
<tr>
<td>Rivera + Semi-stencil</td>
<td>4.25 (1.18/1.13)</td>
<td>6.92 (1.25/1.26)</td>
<td>5.73 (1.58/1.30)</td>
<td><strong>4.90 (1.51/1.76)</strong></td>
</tr>
<tr>
<td>Timeskew + Semi-stencil</td>
<td>4.25 (1.18/1.05)</td>
<td><strong>6.50 (1.33/1.22)</strong></td>
<td>5.53 (1.64/1.27)</td>
<td>5.42 (1.37/2.04)</td>
</tr>
<tr>
<td>Oblivous + Semi-stencil</td>
<td>3.93 (1.27/0.99)</td>
<td>7.04 (1.23/1.29)</td>
<td>6.68 (1.36/1.24)</td>
<td>5.65 (1.31/1.05)</td>
</tr>
</tbody>
</table>

However, Semi-stencil is able to improve the execution time by 25% due to the lower number of loads issued into the memory system.

6.6. SMP Performance

In this subsection, the parallel aspect of the algorithms is evaluated. The multiprocessing version of each stencil algorithm version has been augmented using OpenMP pragmas. We have tested certain combinations of the algorithms on three different architectures: IBM POWER7, Intel Sandy Bridge and the latest Xeon Phi (MIC).

The parallel implementation of each stencil code has been designed following specific decomposition strategies. First, the Naive parallelization has been carried out cutting the least-stride dimension ($NY$) by the number of threads in order to deal with data locality. Each thread computes thread blocks of size $NZ \times NX \times T_{KSMP}$, where $T_{KSMP} = NY/\text{Threads}$. Second, in order not to affect the Rivera-base scheme performance, the serial search of the best $TJ$ block size parameter has been respected as far as possible when decomposing the problem size. The block parameter is computed as: $T_{JSMP} = TJ$ if $TJ \times \text{Threads} \leq NX$, or $T_{JSMP} = NX/\text{Threads}$ otherwise. Hence, each thread computes blocks of size $NZ \times T_{JSMP} \times NY$. Finally, time blocking algorithms have been parallelized by applying thread synchronization methods to keep data dependencies between thread domain computations. Figure 16 shows the Timeskewing [Wonnacott 2000; Kamil et al. 2006] and Cache-oblivious [Frigo and Strumpen 2006] parallel strategies as discussed in Section 3.2.

The IBM BladeCenter PS701 node is equipped with an 8-core 64-bit POWER7 processor operating at 3.0 GHz. Each core contains 4 Floating-Point units, which are able to perform fused multiply-add (FMA) instructions in double precision. These features give an impressive 192 GFlops of peak performance per socket. Additionally, a 32KB L1 data cache and a 256KB L2 cache are included on each core, and an on-chip 32MB L3 adaptive victim cache is shared among all eight cores. The POWER7 socket processor has eight DDR3-1066 RDIMM memory channels. Each channel operates at 6.4 Gbps and can address up to 32 GB of memory. The RDIMM capacity of the PS701 system examined in our study is 128 GB ($16 \times 8$).

IBM has vastly increased the parallel processing capabilities of POWER7 cores by enabling a 4 way SMT (simultaneous multi-threading) system. Therefore, up to 32 threads can run at the same time per chip, leading to a quasi-massive parallel processor. This high thread level parallelism makes the POWER7 platform a very appealing target to test the OpenMP versions of the Stencil Probe microbenchmark. The experiments were conducted using the IBM XL Compiler and the -qsmp flag. Our PS701 node only includes one POWER7 chip; therefore no NUMA-aware code is required since all the data is allocated on the only memory bank. Otherwise, data should be initialized...
Fig. 16. Thread decomposition strategies for time blocking algorithms. Top: in Time-skewing, each thread block (size \(block\)) is divided into three parallelepipeds and executed in the next order at the same time by all threads: firstly the light gray (size \(base = 2 \times \ell \times (t - 1) + 1\)), secondly the white and finally the dark gray area. Bottom: in Cache-oblivious, each thread block is divided into inverted and non-inverted trapezoids (size \(l\)) and executed in this order.

in parallel by all threads due to the first touch page mapping policy. In this way, the dataset memory region would be pinned to its corresponding socket.

Table VIII shows the parallel results on the POWER7 architecture for a large problem \((512 \times 1024 \times 1024)\) in order to exploit the parallel capabilities (up to 32 threads per chip). This table shows the execution time and scalability for each algorithm using the classical (top) and the Semi-stencil implementation (bottom) for varying numbers of OpenMP threads. Reviewing the results, we observe clearly that Semi-stencil enhances the global performance in almost all cases, except, partially, in Cache-oblivious. In some cases, performance is nearly doubled \((1.68 \times 1.83)\) when compared with the classical algorithm running the same number of threads and base algorithm \((2 \times 4)\) threads for Naive and Timeskew implementations respectively. Besides, the Semi-stencil scalability results are mostly similar to the classical algorithm except on 8 and 16 threads runs, where a substantial degradation in scalability performance is observed.

Table VIII. POWER7 SMP results among the different algorithms. Numbers shown represent the total time in seconds (lower is better). Scalability results with respect to the 1 thread execution are shown in parenthesis.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>39.81</td>
<td>19.98 (1.99)</td>
<td>10.01 (3.97)</td>
<td>6.46 (6.15)</td>
<td>4.76 (8.35)</td>
<td>2.47 (16.08)</td>
</tr>
<tr>
<td>Rivera</td>
<td>33.48</td>
<td>16.81 (1.99)</td>
<td>8.73 (3.83)</td>
<td>8.21 (4.07)</td>
<td>5.80 (5.77)</td>
<td>2.77 (12.08)</td>
</tr>
<tr>
<td>Timeskew</td>
<td>39.50</td>
<td>19.84 (1.99)</td>
<td>9.96 (3.96)</td>
<td>6.59 (5.99)</td>
<td>4.64 (8.51)</td>
<td>3.36 (11.75)</td>
</tr>
<tr>
<td>Oblivious</td>
<td>23.86</td>
<td>11.92 (2.00)</td>
<td>5.96 (4.00)</td>
<td>4.85 (4.91)</td>
<td>2.86 (8.33)</td>
<td>2.04 (11.64)</td>
</tr>
<tr>
<td>Naive + Semi</td>
<td>23.45</td>
<td>11.83 (1.98)</td>
<td>5.95 (3.94)</td>
<td>5.87 (3.99)</td>
<td>3.37 (6.94)</td>
<td>2.36 (9.90)</td>
</tr>
<tr>
<td>Rivera + Semi</td>
<td>22.48</td>
<td>11.30 (1.99)</td>
<td>5.66 (3.96)</td>
<td>4.87 (4.61)</td>
<td>4.82 (4.66)</td>
<td>2.13 (10.53)</td>
</tr>
<tr>
<td>Timeskew + Semi</td>
<td>21.47</td>
<td>10.90 (1.97)</td>
<td>5.45 (3.94)</td>
<td>5.54 (3.87)</td>
<td>3.83 (5.59)</td>
<td>2.92 (7.35)</td>
</tr>
<tr>
<td>Oblivious + Semi</td>
<td>22.80</td>
<td>11.46 (1.99)</td>
<td>5.71 (3.98)</td>
<td>5.57 (4.09)</td>
<td>3.47 (6.55)</td>
<td>1.74 (13.06)</td>
</tr>
</tbody>
</table>

Furthermore, the POWER7 architecture scales well considering that despite having 32 threads, only 8 real cores exist per chip, achieving the impressive scalability of \(\approx 16\times\) for the Naive case and \(\approx 13\times\) for Semi-stencil case. This behaviour shows that the 4 way SMT feature maximizes the processor core throughput by offering an increase in efficiency.
This section is concluded by looking at two Intel-based platforms; a representative of the Xeon family and an early access to a Many Integrated Core (MIC) Knight Corner based system. The Xeon system is based on a E5-2670 Sandy Bridge. The system has two sockets where each socket holds 8 cores running at a clock speed of 2.6 GHz. The memory hierarchy has four levels: a 32 KB (I) + 32 KB (D) L1 per core, a 256 KB L2 per core, a 20 MB L3 shared among cores within each socket and a main memory of 32 GB.

For our experiments on the Sandy Bridge, the hyper-threading capability were turned off. Additionally, only one of the sockets was used since the micro-benchmark does not take advantage of the NUMA configuration (memory banks associated to sockets). Besides this, the testing on the Intel architectures followed the parallelism guidelines stated above. Regarding the software stack, the code was compiled and profiled with Intel tools, where the only outstanding compiler flags used were `-avx` and `-DAVX` macro in order to utilize Advanced Vector Extensions (AVX) intrinsics.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>6 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>6.77</td>
<td>3.59 (1.88)</td>
<td>2.48 (2.73)</td>
<td>2.45 (2.76)</td>
<td>2.47 (2.74)</td>
</tr>
<tr>
<td>Rivera</td>
<td>5.27</td>
<td>2.67 (1.97)</td>
<td>1.39 (3.79)</td>
<td>1.00 (5.27)</td>
<td>0.84 (6.27)</td>
</tr>
<tr>
<td>Timeskew</td>
<td>5.29</td>
<td>2.71 (1.95)</td>
<td>1.46 (3.62)</td>
<td>1.06 (4.99)</td>
<td>0.95 (5.57)</td>
</tr>
<tr>
<td>Oblivious</td>
<td>5.82</td>
<td>3.01 (1.93)</td>
<td>1.60 (3.64)</td>
<td>1.15 (5.06)</td>
<td>1.00 (5.82)</td>
</tr>
<tr>
<td>Naive + Semi-stencil</td>
<td>5.19</td>
<td>2.86 (1.81)</td>
<td>2.00 (2.89)</td>
<td>1.99 (2.60)</td>
<td>2.03 (2.56)</td>
</tr>
<tr>
<td>Rivera + Semi-stencil</td>
<td>4.35</td>
<td>2.21 (1.97)</td>
<td>1.18 (3.69)</td>
<td>0.90 (4.83)</td>
<td>0.82 (5.30)</td>
</tr>
<tr>
<td>Timeskew + Semi-stencil</td>
<td>4.48</td>
<td>2.33 (1.92)</td>
<td>1.28 (3.50)</td>
<td>0.97 (4.62)</td>
<td>0.92 (4.87)</td>
</tr>
<tr>
<td>Oblivious + Semi-stencil</td>
<td>4.72</td>
<td>2.46 (1.92)</td>
<td>1.34 (3.52)</td>
<td>1.01 (4.67)</td>
<td>0.94 (5.02)</td>
</tr>
</tbody>
</table>

The dataset sizes selected for the tests were $640^3$ and $256 \times 4096 \times 256$ on Sandy Bridge and Xeon MIC respectively. These sizes are enough to overwhelm the memory hierarchy and also to bear a reasonable load-balance among OpenMP threads. Table IX shows moderate scalability for Sandy Bridge (up to $6.27 \times$) on Rivera, Rivera+Semi-stencil, Oblivious and Oblivious+Semi-stencil. Nevertheless, Naive results lead to the worst scalability results. Clearly, the current domain decomposition (cutting the least-stride dimension) is not the most appropriate approach for this case.

Also, the Semi-stencil algorithm reduces the execution time on 8 cores by only 3% on the worst case (Rivera) to 18% on the better case (Naive). Furthermore, the larger is the number of threads, less is the advantage of the Semi-stencil strategy over the algorithms. It is likely that memory channels saturate earlier, thus achieving the peak memory bandwidth and hampering the linear scalability for a large number of threads.

The MIC Knight Corner beta 0 has 61 cores running at 1 GHz, each one of them handling 4 threads. The memory hierarchy has three levels: a 32 KB (I) + 32 KB (D) L1 per thread, a coherent 512 KB L2 per core connected through a ring bus to other L2 caches, and 8 GB of main memory. The device is attached to a host system via a PCI express bus.

The MIC runtime environment offers new scheduling options for thread affinity settings, such as: scatter, compact and balanced. Tests have been carried out with all of them, selecting only those thread schedulings that offered better performance (balanced in most cases). Also, MIC has two execution models: native (run from the MIC device) and offload (run from the host). The former is accessed via the compiler flags `-mmic` and `-DMIC` macro which produce an executable targeted specifically for MIC. The
Table X. Knight Corner beta 0 SMP (balanced scheduling) results. Numbers represent the total time in seconds (scalability is shown in parenthesis). Due to algorithmic constraints on Timeskew (thread decomposition is performed on least-stride dimension), tests were conducted using a 256 × 256 × 4096 dataset to enable parallelization with a large number of threads.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 thread</th>
<th>61 threads</th>
<th>122 threads</th>
<th>183 threads</th>
<th>244 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>49.71</td>
<td>0.888 (55.9)</td>
<td>0.658 (75.5)</td>
<td>0.632 (78.6)</td>
<td>0.627 (79.2)</td>
</tr>
<tr>
<td>Rivera</td>
<td>45.36</td>
<td>0.813 (55.8)</td>
<td>0.595 (76.2)</td>
<td>0.515 (88.0)</td>
<td>0.594 (76.3)</td>
</tr>
<tr>
<td>Timeskew †</td>
<td>29.46</td>
<td>0.575 (51.2)</td>
<td>0.465 (63.3)</td>
<td>0.599 (49.2)</td>
<td>0.694 (42.4)</td>
</tr>
<tr>
<td>Oblivious</td>
<td>32.90</td>
<td>0.593 (55.5)</td>
<td>0.456 (72.1)</td>
<td>0.571 (57.6)</td>
<td>0.694 (47.4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 thread</th>
<th>61 threads</th>
<th>122 threads</th>
<th>183 threads</th>
<th>244 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive + Semi</td>
<td>38.89</td>
<td>0.710 (54.7)</td>
<td>0.535 (72.7)</td>
<td>0.560 (69.4)</td>
<td>0.587 (66.2)</td>
</tr>
<tr>
<td>Rivera + Semi</td>
<td>44.21</td>
<td>0.780 (56.6)</td>
<td>0.511 (86.5)</td>
<td>0.457 (96.7)</td>
<td>0.471 (93.8)</td>
</tr>
<tr>
<td>Timeskew + Semi †</td>
<td>21.13</td>
<td>0.452 (46.7)</td>
<td>0.426 (49.6)</td>
<td>0.512 (41.3)</td>
<td>0.599 (35.3)</td>
</tr>
<tr>
<td>Oblivious + Semi</td>
<td>34.16</td>
<td>0.601 (56.8)</td>
<td>0.472 (72.3)</td>
<td>0.503 (67.9)</td>
<td>0.576 (59.3)</td>
</tr>
</tbody>
</table>

The latter is based on specific language pragmas. Since all versions of the code were generated through macros, the first was preferred. It was simpler to compile the benchmark with the MIC target flag rather than changing the OpenMP augmentation for a particular Intel pragma set.

It can be seen in Table X, that the scalability of the Rivera+Semi-stencil algorithm reaches 93.8× for the 244 threads case, which corresponds to a 1.26× improvement over the Rivera algorithm for a 25-point stencil. Another worthy result is the combination of Time-skewing and Semi-stencil; this case offers the best performance in execution time (0.426 seconds with 122 threads) among all algorithms. However, it should be clarified that for this case, the dataset size was rearranged and the timestep was split in order to satisfy the parallelepipeds' size constraints (size base) for a decomposition with a large number of threads (see Figure 16 (top)). Summarizing, the Semi-stencil algorithm outperforms the classical implementation for all cases presented from the MIC platform, but, for reasons to be discussed in future work, it is not the case for the Oblivious algorithm.

7. CONCLUSIONS AND FUTURE WORK

In this paper, a novel optimization technique for stencil-based computations is presented. This new algorithmic approach, called Semi-stencil, is especially well suited for scientific applications on both homogeneous and heterogeneous architectures and in particular, for large stencil sizes.

The proposed technique is relevant because it deals with a number of well known problems associated with stencil computations. The first bottleneck, due to low data re-utilization, is the poor floating-point operation to data cache access ratio (FP/Cache). Some techniques like loop unrolling, software pipelining or software prefetching can help improve the performance of stencil codes by partially hiding the dependency stalls. However, these techniques have a limiting factor, namely the number of registers available on the processor. The second identified problem is the memory access pattern. Common algorithms like Rivera or Cache-oblivious tackle this issue with partial success, reducing the overall transfer latency in cache hierarchy architectures. However, due to its orthogonal property, the Semi-stencil outperforms other techniques on many architectures, either when implemented alone or combined with space and time-blocking algorithms.

The Semi-stencil contributions to these problems are:

— reducing the minimum working data set, thus minimizing register pressure and cache memory footprint. This effect becomes more pronounced for high order stencils.
Algorithm XXX: Semi-stencil

increasing spatial and temporal data locality, owing to a reduction in the number of loads, but increasing slightly the number of stores. In homogeneous multi-core architectures, the cache coherence system and the write allocate policy (usually associated with write-back) may produce further memory traffic due to the extra store transactions. Despite the additional stores, the benefit in time of the load reduction overcomes the penalty of stores.

The experimental results show that the best classical stencil implementations for a common 25-point stencil order are typically able to deliver up to 30% of the peak performance. Under the same conditions, the Semi-stencil implementations can achieve up to $1.32 \times$ performance improvement. On multi-core systems, the Semi-stencil algorithm has also shown excellent scalability results on most of the analyzed platforms. On the POWER7 architecture, the scalability soars to $13.06 \times$ for the Oblivious+Semi-stencil case when run over 32 threads. Furthermore, when only 4 threads are used, performance is widely enhanced in most cases and the execution time almost halved with respect to the classical runs. We have also demonstrated the efficient parallel behavior of the Semi-stencil algorithm on novel architectures such as Intel MIC. For instance, on this architecture, scalability reaches $93.8 \times$ over 244 threads for certain tests cases. Additionally, through insight provided by the Roofline and the Multi-Level Cache performance models, we have revealed how the Semi-stencil algorithm performs in terms of Operational Intensity and memory traffic.

Future work will focus on further research of this novel algorithm in several aspects. First, the cost of each computational part (head, body and tail) will be broken down in respect to the global performance. Second, the benefits in performance of each forward and backward update will be analysed in detail for each Semi-stencil axis. Third, on cache-based architectures with a write allocate policy, a write miss may have a negative impact on the stencil performance. This is due to the pollution that the cache-line allocation produces for the store instruction issued. Recently, new architectures with cache-bypass techniques have appeared, which minimize cache pollution when writing data to memory. Our future efforts will also assess the performance of the Semi-stencil algorithm when combined with bypassing. For instance, on the SSE/AVX technologies used on x86 architectures, the \texttt{vmovntpd/movntpd} cache-bypass instructions can be employed for this purpose. In contrast, on the Power ISA, the non-temporal \texttt{dcbz/dclz} instructions have a dissimilar behavior to the Intel ones. Unlike the x86 architecture, these Power instructions only set to zero the contents of a cache-line without bringing it from main memory, evicting finally the cache-line. Nevertheless, the \texttt{dcbst} instruction, which enables prefetching for stores, could partially make up for this lack of bypass instruction. This store-stream prefetching instruction improves performance by anticipating the request of the cache block fetch before it is actually needed by the program. In doing so, the program can later perform stores to the block without experiencing the additional delay caused by fetching the block into the cache. Finally, regarding future work, the Semi-stencil strategy will be evaluated on Lattice Boltzmann Magnetohydrodynamics (LBMHD) applications in order to assess any benefit in performance.

ACKNOWLEDGMENT

The authors would like to express their gratitude to the Partnership for Advanced Computing in Europe (PRACE) for their grant of computing time on the Huygens, Juropa, Jugene and Louhi supercomputing facilities. The PRACE project receives funding from the EU’s Seventh Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528.

The IBM PS701 BladeCenter used in this paper was funded by the Spanish Ministry of Science and Innovation under grant 2009501052 awarded to Francisco J. Cazorla. The authors also thank to Roberto
REFERENCES


Algorithm XXX: Semi-stencil


