Automatic efficient data layout for multithreaded stencil codes on CPUs and GPUs

Julien Jaeger, Denis Barthou

▶ To cite this version:

Julien Jaeger, Denis Barthou. Automatic efficient data layout for multithreaded stencil codes on CPUs and GPUs. High Performance Computing conference, Dec 2012, India. pp.1-10. hal-00793201

HAL Id: hal-00793201

https://hal.archives-ouvertes.fr/hal-00793201

Submitted on 28 Feb 2013

Automatic efficient data layout for multithreaded stencil codes on CPUs and GPUs

Julien Jaeger*, Denis Barthou†
*University of Versailles St Quentin, FR
julien.jaeger@prism.uvsq.fr
†University of Bordeaux / INRIA Bordeaux Sud-Ouest, FR
denis.barthou@labri.fr

***Abstract*—Stencil based computation on structured grids is a kernel at the heart of a large number of scientific applications. The variety of stencil kernels used in practice make this computation pattern difficult to assemble into a high performance computing library.**

With the multiplication of cores on a single chip, answering architectural alignment requirements became an even more important key to high performance. In addition to vector accesses, data layout optimization must also consider concurrent parallel accesses.

In this paper, we develop a strategy to automatically generate stencil codes for multicore vector architectures, searching for the best data layout possible to answer architectural alignment problems. We introduce a new method for aligning multidimensional data structures, called multipadding, that can be adapted to specificities of multicores and GPUs architectures. We present multiple methods with different level of complexity. We show on different stencil patterns that generated codes with multipadding display better performances than existing optimizations.

I. Introduction

The petaflop era has given rise to architectures of increasing complexity. Modern architectures combine many different levels of parallelism and a large memory hierarchies. SIMD instructions, such as those proposed in Intel SSE and AVX ISA for instance, and multi-thread programming offer opportunities to use this parallelism to reach high level performance. This comes however at the cost of a careful data layout organization in order to match memory alignment constraints. Combining vectorization and memory bank conflicts limitation with a proper data layout is a key to performance in current multicores and GPUs.

Stencil based computations represent a large class of applications, ranging from image processing, computational electromagnetics, hydrodynamics, lattice QCD or other physics simulations requiring the resolution of PDEs using finite difference or volume discretization. However, the variety of stencil kernels used in practice make this computation pattern difficult to assemble into a high performance computing library. Besides, the low flop/byte ratio that most common stencil exhibit requires to precisely tune the data layout and optimize memory accesses according to the architecture features.

Automatic transformations for every stencil pattern is important, as the variety of stencil kernels used in practice is very large. Generating efficient code for CPUs and GPUs, taking into account alignment requirements, is paramount. Stream alignment conflicts are a fundamental algorithmic issues, as shown by Henretty *et al.* [9].

The contributions of this paper are:

- A data-layout transformation so as to handle CPU and GPU alignment issues.
- A compact code generation for stencils.

In this paper, we develop a novel strategy to automatically generate stencil code for CPUs and GPUs, searching for the best data layout to answer alignment issues, without breaking the stencil structure and the data locality. We introduce a new data layout transformation, called *multi-padding*, extending the usual padding so as to maximize the number of aligned loads for vectorization. We present several methods to find the best paddings, with different levels of complexity. We show on stencil codes, in particular Jacobi and Laplacian computations, that generated codes compare well with hand-tuned codes, and that multi-padding can bring significant performance gains compared to the usual padding.

The rest of the paper is organized as follows. We first present related works in Section II. Section III presents architectural alignment requirements for efficient memory access. In Section IV, we focus on the SIMD alignment issue on stencil computations to propose our multipadding method. Details on the code generation on any architecture are presented in Section V. Experimental validation is described in Section VI. Then we conclude in Section VII.

II. Related Works

For generalist multicore processors, there has been considerable interest on vectorization issues. Many papers focus on loop transformations to improve the vectorization quality, playing with data locality to decrease the amount of unaligned loads or shuffles ([17], [25], [19], [8], [18], [20], [27], [2], [7]). Similarly, data locality has been the focus of optimizations for stencil computation, through blocking ([11], [17], [3], [26], [24], [16], [5], [6]), tiling ([22], [12], [14]) or skewing ([28]), improving single and multi-core performance. These works however do not directly consider alignment issues, only focusing on data locality and register reuse.

For GPUs, generating efficient code able to close the gap with the peak performance, is quite challenging [13]. Several auto-tuning models applying on both CPU and GPU architectures have been proposed, to ease code development on

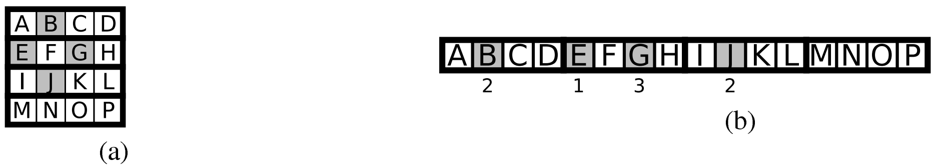


Fig. 1. Representation of a Jacobi 2D pattern, using vector of four elements (simple data in SSE). **(a)** Logical view, update of F requires elements B, E, G, J. **(b)** Memory view, the four elements are not in the same place in vectors. E is in the first position, B and J are both in the second position, and G is in the third. Choosing alignment of B and J as the valid one, two shuffles or unaligned loads are required.

multiple target architectures ([1], [16], [26], [23]). Neither of these models however consider alignment issues.

Some recent works propose data layout transformations ([15], [9]) in order to improve vectorization for stencil codes. The work of Henretty *et al.* [9] focuses for instance on data layout reorganization, through dimension lift and transpose, removing all non-aligned loads. However, they do not consider multithreaded execution. Array elements initially contiguous may be spread accross the whole array, and this will probably result, for a multithreaded execution, in memory sharing situation inducing performance slowdown.

The works of Datta *et al.* [4] and Kamil *et al.*[10] describe an auto-tuning framework to optimize stencil computations on multicore architectures, including GPU. The optimizations proposed take advantage of the different levels of memory hierarchy through blocking. Alignment issues are not addressed directly though. Performance figures bring some comparison basis that will be used in section VI.

III. Efficient Data Layout

In current multicore architectures, high performance can be achieved through careful data layout and memory management.

For multicore architectures supporting hardware vectorization, one has to deal with a set of entangled hardware constraints, both coming from vector memory accesses and from memory structured into multiple banks. We provide in the following an overview of these constraints for both CPU and GPU architectures.

A. Thread Warps and Memory Banks

In modern CPUs and GPUs architectures, a set of threads can run simultaneously on the same chip. To simplify, we will use the term s *warp* to refer to such set of simultaneous threads. A warp is a basic notion in GPU programming. We generalize it to multicore CPUs, considering a warp as the collection of threads running on a single chip, with a maximum of one thread per core (no hyperthreading is considered). On GPU, due to the SIMD nature of the architecture, threads of a same warp are synchronous. On CPU, there is no such limitation.

Vertical parallelization consists in slicing an iteration space in several parts, which will be swapped by a thread, and is used very often to parallelize linear algebra or stencil codes. It implies that simultaneous threads perform the same instruction at nearly the same time, issuing multiple memory requests simultaneously. Whenever these requests target the same memory bank, this is called a bank conflict. Memory accesses to the same bank are then serialized instead of being executed in parallel (for different banks). Requests on a busy memory bank are delayed, and memory accesses, often critical for the performance of stencil codes, take more time than usual to fetch the required data.

To avoid this problem, data accessed by different thread can be mapped to different memory banks. This implies that these data have different memory alignments modulo the number of memory banks. However, this is not convenient since the number of threads can exceed the number of memory banks. In this case, it is better to divide memory requests equally between all memory banks, than to aggregate numerous requests on the same bank.

B. SIMD Memory Accesses

In order to obtain high performance and tip the balance between memory accesses and computation, array accesses have to be vectorized. So far, many architectures (including Intel AVX) exhibit different performance depending on whether the accesses are aligned or unaligned on a x byte boundary, with x the size of the vector. A vector is composed of several elements, depending on the element size and most vector operations are element-wise (this changes in recent vector ISA). For a stencil, vectorizing the computation implies that all elements of the access pattern are at the same index in their respective vector.

At first sight, GPUs do not seem to have this problem, since an element is composed of multiple registers, whereas on CPUs, a vector register is composed of several elements, causing alignment issue. On GPUs, a vector register always includes only one element. In CUDA, built-in vector types are proposed, providing programmers with a set of vectors of different sizes, up to 128 bits or a vector of four elements. In the CUDA Programming Guide [21], these built-in types are said to enforce specific alignment of data, sometimes with vector requirements being different from its base type requirement. Among others, a char1 has no memory alignment constraint, a char2 must be in memory at an address aligned on a 2 bytes, a char3 has no alignment constraint and a char 4 has 4-byte alignment constraint. Using these built-in types will cause the same alignment issues as for CPU.

IV. Multi-dimensional Multi-Padding

We study in this section alignment issues for stencil computations, which are representative of many linear algebra computations. For the following, the general stencil pattern studied is on a n-dimensionnal array $A[V_1, \ldots, V_n]$ of the form:

$$\ldots = f(\mathbf{A}[i_1 + d_{11}, \ldots, i_n + d_{1n}], \ldots, \mathbf{A}[i_1 + d_{m1}, \ldots, i_n + d_{mn}])$$

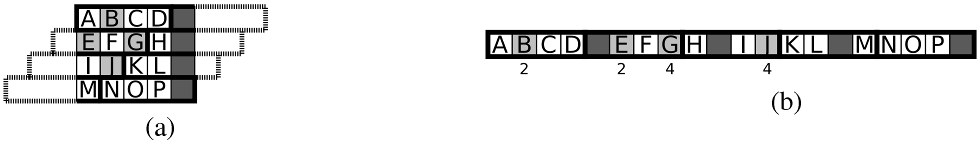


Fig. 2. Simple padding applied on the Jacobi 2D pattern with vector of four elements. **(a)** Logical view, alignment of elements have changed. Dotted lines represent unfinished vectors on a line, being completed with elements of the next line.**(b)** Memory view, two distinct alignments. Do not improve compared to no padding, since here again two shuffles or unaligned loads are required.

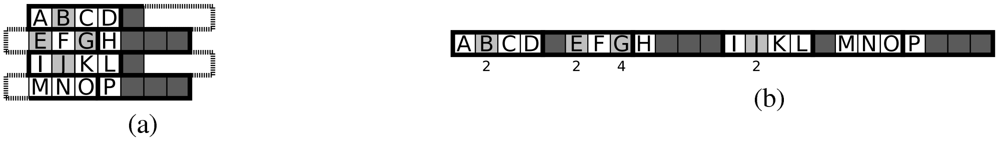


Fig. 3. Multipadding 1-3 applied on the Jacobi 2D pattern with vector of four elements. **(a)** Logical view, alignment of elements have changed. **(b)** Memory view, still two distinct alignment.B, E and J are in the same position in vectors. Now only one shuffle or unaligned load required.

This computation reads all elements of A belonging to an access pattern, defined by $(i_1, \ldots, i_n)$ and the matrix of integers $(d_{hk})_{hk}$. We assume in the rest of the paper that all elements of A are mapped contiguously in memory, and to simplify notations, we assume elements of A are 1 byte long. Results of this section can be generalized to stencil computations on several statements, involving multiple arrays.

Vectorizing such expression requires to load into SIMD registers, at the same position in the vectors, all the elements of the access pattern. Optimization of this code for GPU is discussed in Section V.

We first discuss alignment issues for an SIMD implementation, illustrated on a Jacobi stencil, then introduce the multi-dimensional padding. Finally we extend it further into multi-dimensional, multi-padding (*MDMP*).

A. Alignment Issues

A particular case of stencil, used in many application, is a 2D Jacobi stencil:

$$B[i,j] = A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]$$

Fig.1**(a)** presents the pattern of access: The stencil uses the elements B, E, G, J. Fig. 1**(b)** shows the offsets of these elements in the vectors of 4 elements $(A, B, C, D), (E, F, G, H)$ and (I, J, K, L). Here, the elements are not in the same positions in the vectors. These elements have to be shuffled or loaded from unaligned addresses.

More generally, we propose to describe the conditions when the elements of a stencil pattern are all aligned: For SIMD vectors of size l, this implies that the addresses of these elements are the same modulo l. The memory address of an element $A[i_1, \ldots, i_n]$ is defined by:

$$\&A[i_1, \ldots, i_n] = A + \sum_{k=1}^{n} i_k S_k$$

where S_k define the strides separating consecutive elements of A in the k^{th} dimension. Thus, given a vector $(i_1, \ldots, i_n)$, the elements of A required for the stencil computation have the same alignment if and only if:

$$\exists c, \forall h, A + \sum_{k=1}^{n} (i_k + d_{hk}) S_k \equiv_l c$$

where $\equiv_l$ is the identity modulo l and c is a constant.

Thus, all elements are aligned if and only if the vector of strides (S_k) checks the following constraint:

$$\exists c, \begin{pmatrix} d_{11} & \ldots & d_{1n} \\ \vdots & \ddots & \\ d_{m1} & & d_{mn} \end{pmatrix} \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix} \equiv_l \begin{pmatrix} c \\ \vdots \\ c \end{pmatrix} \qquad (1)$$

When Equation (1) is not checked, some elements are misaligned in memory. Misaligned elements would have to be aligned in registers for the computation to be correct. The main methods to align misaligned elements are:

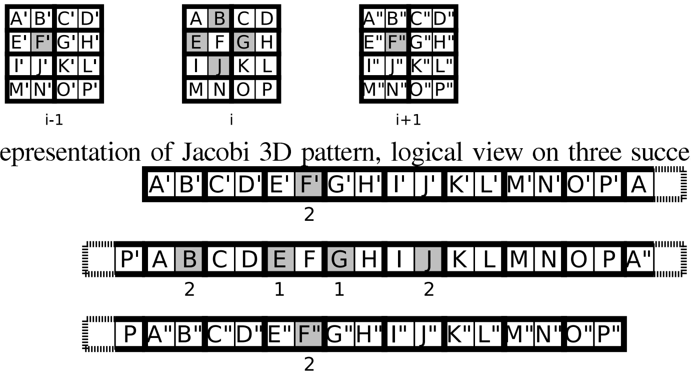
- **Misaligned Load**: Load data from a unnatural alignment in a vector register. On many architectures, this comes at the expense of a performance penalty.
- **Memory Duplication**: Duplicate the array as many times as there are different alignments between required neighbors. Each of these arrays will be aligned differently. This method is not to be used on memory bound computations.
- **Shuffle**: Create with two naturally aligned vectors a third vector with the required elements. This method adds instructions and increases register pressure.
- **Padding**: Increase the size of the strides so as to align in memory elements of the pattern.

While the three first are time consuming or consume significantly more memory, padding has the advantage of only requiring a minor change in the data layout. If padding occurs only for large strides, most of the spacial locality of the code is kept unchanged. No additional instruction is inserted in the code and, although there is an extra memory consumption, it is negligible compared to the Memory Duplication method.

For instance, in the Jacobi 3D presented in Fig.4**(a)** (showing three planes of the volume) and **(b)** (linearized memory), the elements required by the computation (F', B, E, G, J, F'') have not the same alignment (resp. $(2, 2, 1, 1, 2, 2)$). E and G are unaligned compared to the other elements.

B. Formulation of Multi-dimensional Padding

Let us assume, with no loss of generality, that the array is row-major (as in C). We describe in this section a simple padding for a 2D array, and then generalize this to a padding for a multi-dimensional array.



(a) Representation of Jacobi 3D pattern, logical view on three successive planes.

(b) Representation of Jacobi 3D pattern, logical view.

Fig. 4. Representation of a Jacobi 3D pattern, using vector of two elements (double data in SSE). **(a)** Logical view, update of F requires elements B, E, G, J, F' and F". **(b)** Memory view, the three lines are contiguous in memory (P' in the first and second lines are the same one). The six elements are spread through the two possible alignment. Choosing alignment of B, J, F' and F" as the valid one, two shuffles or unaligned loads are required.

A simple padding for a 2D array consists in adding elements at the end of each row, in order to align elements needed in a stencil pattern. The number of elements is the same for each row. For instance, Fig.2**(a)** and **(b)** describe the access pattern of a Jacobi 2D when rows are padded with one element. The elements B, E, G, J needed by the computation have two different alignments. While better than the first vectorization (with no padding), there are still 2 elements unaligned. Likewise in 3D, Fig.5**(a)** and **(b)** show that padding a single dimension is not sufficient to improve the alignment.

A *multi-dimensional padding* consists in adding elements at the end of each dimension. The number of elements added is constant per dimension, but may vary from one dimension to another. This flexibility allows more elements in a stencil pattern to be aligned. To define the number of elements p_k to add in each dimension k, we describe the relation between the value of the padding and the stride separating two consecutive elements of the array. For any dimension k, the stride S_k separating two consecutive elements in the k^{th} dimension has to take into account the size taken by all elements in dimensions h, $h < k$, and any padding on these dimensions. This provides a recurrent definition of the stride for dimension k: it is equal to the stride of the $k-1^{th}$ dimension times the number of elements in this dimension, plus the padded elements for dimension $k-1$. Formally, the stride is defined by:

$$k > 1, S_k = S_{k-1}V_{k-1} + p_{k-1},$$

and S_1 is equal to the size in byte of the element. In a matrix form, this defines a relation between paddings and strides:

$$\begin{pmatrix} p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} \equiv_l \begin{pmatrix} -V_1 & 1 & & \\ & \ddots & \ddots & \\ & & -V_{n-1} & 1 \end{pmatrix} \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix}, \tag{2}$$

where only non-null elements are represented in the matrix. The value of each padding is bounded by the length of the SIMD vector (due to the relation $\equiv_l$). Besides, as soon as strides of two successive dimensions are multiple of the vector length (maybe thanks to padding of these dimensions), Equation 2 shows that there is no need for padding in outer dimensions.

C. Finding a Multi-dimensional padding

Equations (1) and (2) provide the constraints on the padding so as to obtain potentially an alignment for all elements of a stencil pattern. Since padding consists in adding extra, useless elements, the total count of such elements should be minimized in order to reduce the impact of this method.

Besides, for some stencil patterns, even a multi-dimensional padding cannot ensure that all elements are aligned. For each unaligned element, either a shuffle or a misaligned load will be generated. To count these necessary shuffles, according to the padding chosen, we reformulate Equation (1) so that a solution can always be found, using slack variables w_k:

$$\begin{pmatrix} d_{11} & \dots & d_{1n} \\ \vdots & \ddots & \\ d_{m1} & & d_{mn} \end{pmatrix} \cdot \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix} \equiv_l \begin{pmatrix} c \\ \vdots \\ c \end{pmatrix} + \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix} \tag{3}$$

Here, w_h stands for an additional shift on the element h of the stencil pattern, corresponding to an alignment change. As for any shift, $0 \leq w_h < l$. In order to count and minimize the number of elements for which an alignment change is necessary, we add the constraint for any element:

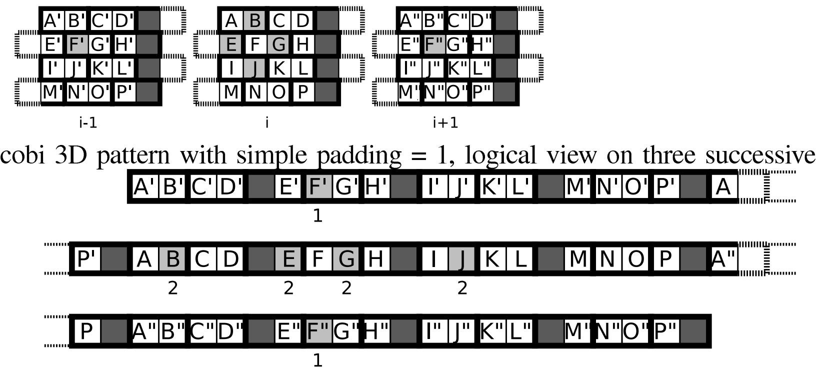
$$0 \leq w_h < (1 - u_h)M \tag{4}$$

with M a big constant. u_h is a $0-1$ variable equal to 1 whenever a shuffle or an unaligned load is needed.

An objective function to minimize for the multi-padding problem, combining the minimization of the number of unaligned loads/shuffles and the memory consumption due to padding can be defined as:

$$\min(\sum_{h=1}^{m} u_h + \sum_{k=1}^{n} p_k), \tag{5}$$

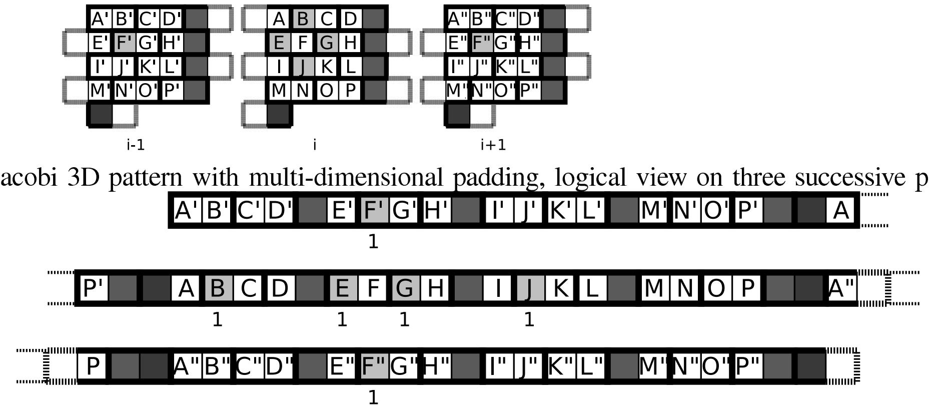
with Equations (2),(3),(4) on the variables w_h, S_h and p_h. A different objective function may be formulated, depending on the architecture and on the cost associated to memory consumption and shuffles.


A' B' C' D'
E' F' G' H'
I' J' K' L'
M' N' O' P'
A B C D
E F G H
I J K L
M N O P
A" B" C" D"
E" F" G" H"
I" J" K" L"
M" N" O" P"
i-1
i
i+1
A' B' C' D' E' F' G' H' I' J' K' L' M' N' O' P' A
1
P' A B C D E F G H I J K L M N O P A"
2 2 2 2
P A" B" C" D" E" F" G" H" I" J" K" L" M" N" O" P"
1

(a) Jacobi 3D pattern with simple padding = 1, logical view on three successive planes.

(b) Jacobi 3D pattern with simple padding = 1, memory view.

Fig. 5. Simple padding applied on the Jacobi 3D pattern with vector of two elements. **(a)** Logical view, alignment of elements have changed. Dotted lines represent unfinished vectors on a line, being completed with elements of the next line.**(b)** Memory view, two distinct alignments. Do not improve compared to no padding. There is again four elements on one alignment (second position), and two elements on the other possible alignment. Again, two shuffles or unaligned loads are required.


A' B' C' D'
E' F' G' H'
I' J' K' L'
M' N' O' P'
A B C D
E F G H
I J K L
M N O P
A" B" C" D"
E" F" G" H"
I" J" K" L"
M" N" O" P"
i-1
i
i+1
A' B' C' D' E' F' G' H' I' J' K' L' M' N' O' P' A
1
P' A B C D E F G H I J K L M N O P A"
1 1 1 1
P A" B" C" D" E" F" G" H" I" J" K" L" M" N" O" P"
1

(a) Jacobi 3D pattern with multi-dimensional padding, logical view on three successive planes

(b) Jacobi 3D pattern with multi-dimensional padding, memory view

Fig. 6. Multi-dimensional padding applied on the Jacobi 3D pattern with vector of two elements. **(a)** Logical view, alignment of elements have changed. In addition to the simple padding, an extra element is added at the end of each planes.**(b)** Memory view, all six required elements have the same alignment (first position in vectors). Only aligned loads will be performed, no additional shuffles or unaligned loads are required.

For the example in Figure 6**(a)** and **(b)**, the multidimensional padding (1 in each dimension) aligns all elements of the stencil pattern. There is no need for unaligned access or shuffle.

D. Multi-dimensional Multi-Padding

To further reduce the number of unaligned accesses in a stencil pattern, we describe now an extension of the multi-dimensional padding. The principle can be easily explained for a 2D array.

For a 2D array, each row can be padded by a number of elements. Instead of padding each row with the same number of elements, rows are now padded with a number of elements that can differ from row to row. In order to keep the code generation with this padding manageable, the padding is cyclic: every T rows, the sequence of padding values starts over again. This technique can be generalized to any number of dimensions. The Jacobi 2D in Fig.3 shows that by padding even rows with one element, and odd rows with 3 elements, all elements of the stencil pattern can be aligned but one. Note that the remaining unaligned access is a compulsory alignment conflict, since the two elements E and G are in the same row. Padding inside the row would remove this conflict, but also it would degrade spacial locality and increase significantly memory requirements. We choose in this paper not to consider padding between elements of the same row. Hence, under this assumption, the padding proposed for the Jacobi 2D minimized the number of unaligned accesses.

More generally, we consider some periodic functions $S_k(j)$, defining the strides separating consecutive elements of A in the dimension k. The address of an element of the array A is defined by:

$$\&A[i_1,\ldots,i_n] = A + \sum_{k=1}^{n}\sum_{j=0}^{i_k} S_k(j) \qquad (6)$$

If T is the period of the functions S_k, then S_k is characterized entirely by the set of values $\{S_k(0),\ldots,S_k(T-1)\}$. In Equation 6, $S_k(j)$ can be replaced by $S_k(j \mod T)$. For a given value h between 0 and $T-1$, there are $\lfloor \frac{i_k-j}{T} \rfloor$ occurrences of $S_k(h)$ in the sum. Hence formula (6) can be rewritten:

$$\&A[i_1,\dots,i_n] = A + \sum_{k=1}^{n}\sum_{h=0}^{T-1} \lfloor \frac{i_k - h}{T} \rfloor S_k(h).$$

Considering the elements of the stencil for the computation in $(i_1,\dots,i_n)$, they are all aligned if and only if $\exists c, \forall j$,

$$\sum_{k=1}^{n}\sum_{h=0}^{T-1}\lfloor \frac{i_k-h+d_{jk}}{T}\rfloor S_k(h) \equiv_l c$$

for some constant c. Let us denote $a_{jkh}(i) = \lfloor \frac{i-h+d_{jk}}{T}\rfloor - \lfloor \frac{i}{T}\rfloor$. The value of $a_{jkh}(i)$ can only take two values at most when i changes, and $a_{jkh}(i)$ is a periodic function of period (at most) T. The condition for alignment then becomes $\exists c, \forall j$,

$$\sum_{k=1}^{n}\sum_{h=0}^{T-1}(\lfloor\frac{i_k}{T}\rfloor + a_{jkh}(i_k))S_k(h) \equiv_l c$$

Writing this constraint in matrix form:

$$\begin{pmatrix} a_{110}(i_1) & \dots & a_{1n,T-1}(i_n) \\ \vdots & \ddots & \\ a_{m10}(i_1) & & a_{mn,T-1}(i_n) \end{pmatrix} \cdot \begin{pmatrix} S_1(0) \\ \vdots \\ S_1(T-1) \\ \vdots \\ S_n(T-1) \end{pmatrix} \equiv_l \begin{pmatrix} c \\ \vdots \\ c \end{pmatrix} \tag{7}$$

where the values of $S_k(h)$ are the unknown. This constraint of alignment is similar to Equation (1). The periodicity of the strides leads to consider a larger stencil pattern (matrix has size $m \times nT$). Similarly to Equation (2), strides between elements and padding are connected through the equation:

$$\forall k > 1, \forall h, S_k(h) = \sum_{j=0}^{T-1}\lfloor \frac{V_{k-1}-j}{T}\rfloor S_{k-1}(j) + p_{k-1}(h). \tag{8}$$

Equations (7) and (8) define the constraints for the periodic padding.

E. Padding for Bank Conflicts

For multithreaded stencil codes, we assume that the parallelization is such that one or several of the dimension of the stencil correspond to parallel loops. This means for instance that some indices i_k can be written as $i_k = b.t + ii_k$ where t is a thread number, ii_k an index enumerating the block in this dimension allocated to a given thread. Other partitioning of an array dimension can be considered, as long as the partitioning correspond to an affine transformation. However, the parallelization is an input for the padding and the exact distribution of iterations among threads has to be known before padding.

To minimize bank conflicts, each elements accessed simultaneously by each thread must hit a different memory bank. Hence, for each couple of threads, the address of their first element must not be on the same alignment. For a given address m, the number b of the memory bank corresponding to it corresponds to some contiguous bits of m: $b = m/B \mod NB$ where B and NB depend on the architecture. Thus, following the representation of the addresses in previous section, we can express that each thread accesses different memory banks: if $b(t_1, ii)$ is the memory bank accessed by thread 1 through a memory access, and $b(t_2, ii)$ is the memory bank corresponding to the same access in the block for thread 2, then the following equation corresponds to the constraint:

$$b(t1, ii) - b(t2, ii) \equiv_{NB} db,$$

with NB the number of banks and $db \geq 1$.

As for the padding for vectors, slack variables following the same constraint as in equation 4 can be added in order to ensure the existence of a solution.

F. Integer Linear Program Formulation

Finding multi-dimensional multi-padding for a particular architecture and stencil boils down to the resolution of an integer linear program. Indeed, provided the period T is given and constant, equations are similar to those considered in IV-C and are affine. Modulo equations are handled by the use of a new variable (for the modulo), possible compulsory unalignment can be handled by the introduction of slack variables to minimize. This is used for satisfying alignment constraints for SIMD and for bank conflict.

For MDMP, The number of equations has increased due to the periodic values and due to the fact that the coefficients of the matrix in formula 7 are also periodic (in $(i_1,\dots,i_n)$). The formulation of the function to minimize, depending on the number of shuffles, is similar to the one presented in section IV-C.

The period T can be chosen either by the user, or iteratively found (trying 1, 2, ... and each time computing a solution). Other metrics, such as the size of the code, the number of registers used, can be used to bound the search for a appropriate padding period.

V. Code generation

Code generation for stencils on CPUs or GPUs consists in two phases:

1) **Memory allocation and data transfer**: The resolution of the system of constraints relative to multi-padding provides the necessary amount of memory to allocate (or re-allocate) for effectively pad the different dimensions of the data structures. Since the data layout transformation consists in translating data in memory, any required copy (for instance for GPU) is easily generated.
2) **Instruction generation**: Once data layout is computed and memory is allocated, the code is generated. Memory instructions are generated taking into account the new data layout. Aligned and unaligned instructions can be necessary, and the two different ways to generate these instructions are presented in Section V-B.

A. Efficient data layout

Algorithm 1 describes the different steps to remove a maximum number of inefficient memory accesses (bank conflicts or unaligned aligned data accesses).

Algorithm 1 Efficient Multi-Padding and Memory Management

```
1: Find Multidimensional Multi-padding to remove bank
   conflicts while aligning data for vectorization.
2: Memory Allocation
3: Data transfer, and Copy from High Latency Memory
   (HLM) to Low Latency Memory (LLM)
```

1) Padding for Vectors: Padding for vector is used to align a maximum of required elements in the computation pattern. Aligned elements will be accessed through more efficient memory transactions. The complete procedure is describe in Section IV. The constraints are the same for CPUs and GPUs when using built-in vector types in CUDA.

However, on GPUs, hardware computations use only on element at a time. If one uses its own vector structure instead of CUDA built-in types, no specific alignment is required, and this step may be skipped, remaining only to avoid bank conflicts between the threads.

2) GPU parallelism: As explained in Section III-A, simultaneous memory accesses from concurrent threads should target different memory banks.

GPU threads can be considered as CPU threads, and the GPU code can be designed for each thread to access an independent block of data. However, warps often access contiguous data in the shared memory. To generate such a GPU code, one has to know if the whole warp will access contiguous data (in that case there is nothing to do), or if slices of warp will be spread across several lines/blocks. For example, a warp being a set of 32 threads, the code can be designed for the warp to access 32 contiguous data, or for four quarter of warps to access 8 contiguous data on different lines. These slices will then be considered as meta-vectors (4 in our example), which should not have any memory banks in common . The multipadding method will be applied with these meta-vectors, using their specific constraints.

3) Memory Allocation: Once all the necessary paddings found, one must combine them to know the total amount of memory to allocate.

On GPU, memory will be allocated to the shared memory of a chip, according to the padding found in the first step of the algorithm. Then, in a next step, necessary data will be transfer from global memory to the shared memory, closer and easier to access efficiently.

On CPU, a global memory allocation will be made. This global allocation will include the padding for vectorization, and the padding for threads at end of logical thread block.

4) From HLM to LLM: Data transfer consists in putting all needed data for computation on the correct device memory. It will be a copy to the global memory if GPU is considered. In order to reduce the cost of memory transactions, or to simplify memory accesses pattern, a first copy from the High Latency Memory, in which the data are originally stored, to a Low Latency Memory can be useful.

On GPU, data will be transfered from the global memory to the shared memory. Global memory has a very restrictive access, since all threads of a warp should access contiguous data with a specific alignment to perform efficient memory movement (coalesced loads). On the other hand, threads can load data from shared memory very easily, and the concurrent memory transactions can be spread across the entire array without penalties (if bank conflicts are avoided).

CPU memory hierarchy is simpler since the wherever the data are, they are fetched to the nearest memory to the computing unit each time they are read. The only needed data transfer is from the original array to the new allocated region, modifying the data layout.

B. SIMD code generation

For CPU code generation, we rely on state-of-the art compiler vectorization techniques to generate SIMD code. Two approaches are taken:

- Vector extensions proposed by compilers (`icc` and `gcc`, with attribute directives)
- Intrinsic functions

We favor in the code generation the first approach, whenever aligned elements are accessed. The advantage of this method is that instruction selection is actually performed by the compiler, depending on the target architecture and target vector ISA. However, this method is limited to basic algebra operators and all memory accesses to a vector have to be all aligned to be efficient. These limitations narrow the number of stencil patterns that can be handled, and only a Jacobi pattern in double precision can be efficiently generated with this method. For other cases, we use the intrinsics approach (with SSE2), with some possible remaining unaligned vector elements.

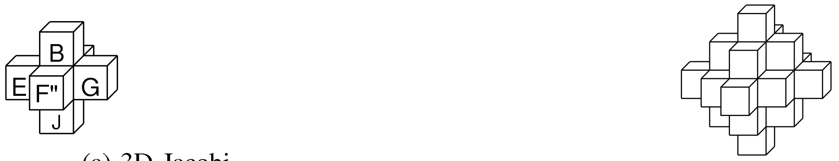
VI. Benchmark results

To evaluate our method, we run several tests of stencil computations (Jacobi, Laplacian and the convex envelope of a Von-Neumann neighbourhood), on several Intel architectures along with a GPU environment, and all tests have been done with icc 12.1.0 and gcc 4.6.1 compilers.

The multipadding method was also applied on a 9-points 2D stencil, and a 25-points 3D stencil using Moore's neighbourhood, and on a diamond-shape pattern in 2D using a Von-Neumann neighbourhood of range 2 Performance results are not presented since the best possible padding for the 9-points and 25-points stencil is no padding at all. For the diamond-shape pattern, the best possible padding is a simple padding.

Performance results in section VI-B are for computations in simple precision, as it is more difficult to align data when vectorizing such codes (four possible positions in vectors). Tested codes have been generated with intrinsics performing aligned loads on correctly aligned data, and unaligned loads otherwise. Graphs bars labelled ”No pad”, used as basis for comparison, are for vectorized versions of the stencil computations with no padding transformation.

Performance results in section VI-C are for computations in double precision, since the results we use for comparison have been presented only in double precision. Tested codes have been generated with vector extension, and loads are automatically managed by the compiler. Note that for all experiments, no blocking is realized to increase data locality.



(a) 3D Jacobi

(b) 3D Von-Neumann

Fig. 7. Shapes of the studied stencil. (a) Shape of a 3D jacobi stencil. (b) Shape of the convex envelope of a 3D Von-Neumann neighbourhood of range 2.

A. Target Architectures

We evaluate our approach on several architectures. We run experiments on three Intel CPUs and on a GPU NVIDIA Quadro card, plugged to the Nehalem machine:

- Nehalem Gainestown X5550, NVIDIA Quadro 5800 FX: The architecture is a 2.66 Ghz quad-core processor with a shared 8MB L3 cache. Supporting SSE4 SIMD instructions, each core can fetch and decode four instructions per cycle, for a peak performance of 10.64 GFlop/s per core in double precision. Each core has a 32KB L1 data cache, 32KB L1 instruction cache and a 256 KB L2 cache. The machine has two sockets, for a total of 8 available cores. Each socket has access to a FSB delivering a max. bandwidth of 12.8 GB/s.
  Besides, the machine considered is equipped with a NVIDIA Quadro 5800 FX card, running at 1.30 GHz and with 4GB of memory. This card is used to perform GPU tests presented in section VI.
- Westmere Gulftown X5650: The machine used for the benchmarks is a 2.66Ghz four socket architecture, and each socket is an hexa-core with a shared 12MB L3. This processor, a successor of Nehalem, has the same characteristics concerning SIMD. Each core has a peak performance of 10.64 GFlops/s in double precision.
- Sandy Bridge E3-1245: The processor used for the benchmarks is a 3.3Ghz quad-core processor with a shared 8MB L3 cache. Supporting SSE4 and AVX SIMD instructions, each core can fetch and decode up to eight instructions per cycle, for a peak performance of 26.4 GFlops/s per core in simple precision when using SSE instructions. Each core has a 32KB L1 data cache, 32KB L1 instruction cache and a 256 KB L2 cache. Only one processor is available. The maximum memory bandwidth achievable is 21 GB/s.

B. Overall improvement

In order to validate our approach, the first part of our experimentation testbed consists in trying several padding values, for simple padding and multipadding, to see if the best padding(s) returned by the methods discussed in section IV are indeed the best one(s).

In Fig. 9 are shown results for a 4-point 2D jacobi stencil computation, running on only one thread. The multipadding version is compared with versions with no padding, and the usual simple padding. Fig. 9(**a**) presents results when using the gcc compiler, on different square sizes (from L1 to memory). Except for data in L1, the multipadding outperforms the other versions of the Jacobi 2D stencil, with a performance improvement up to 52% for a size of 512x512, and a gain of 28% compared to the simple padded version on the same size. The same experiment as in Fig.9(**a**) is displayed in Fig. 9(**b**), only this time with the use of icc compiler. As the compiler use more aggressive and guided optimization, the produced code without padding already performs well. The multipadding method still gives better performance when in upper hierarchy memory (L3 begins at size 256), up to a x1.24 speedup for a 1600x1600 array against the original version, and a x1.11 speedup compared to the padded version for the same size.

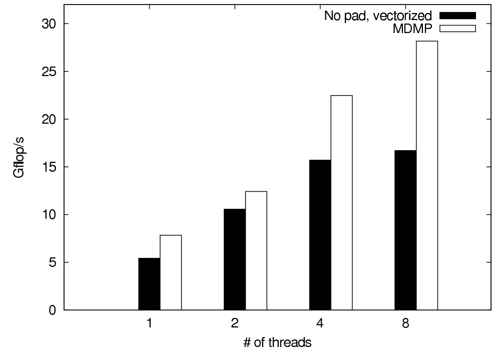
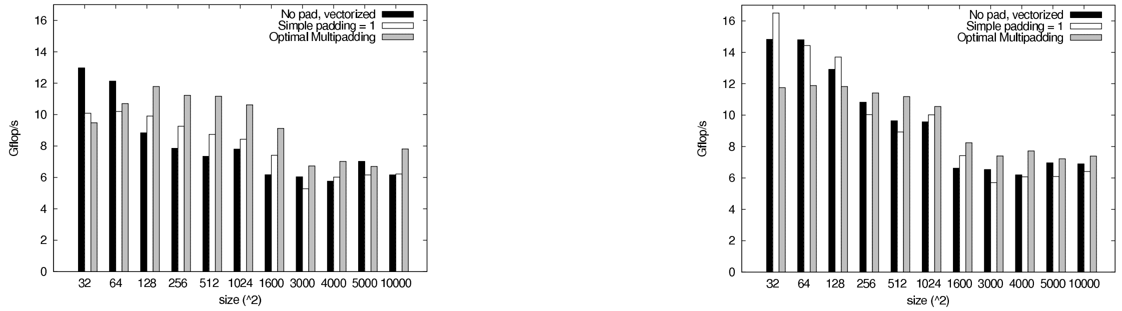


Fig. 8. Performance of a 3D 256^3 Jacobi stencil on Westmere architecture.

In Fig.10 are presented results for a stencil covering the convex envelope of a 3D Von-Neumann neighbourhood of range 2. The shape of this stencil pattern is displayed in Fig.7(**b**). As before, the multipadding version is compared with version with no padding, and the usual simple padding. Here, only a multidimensionnal padding is sufficient, i.e. each dimension has been modified with only one padding value. This multipadding allows to align 10 elements in the stencil, instead of only 6 elements for the version with no padding and with the simple padding. Figures 10(**a**) and 10(**b**) presents results when using respectively gcc and icc compilers. For each set of benchmarks, the multipadding version always outperforms the other version, up to a performance improvement of 52% compared to the original version for a size of 256^3 with gcc, and 72% for a size of 300^3 with icc.

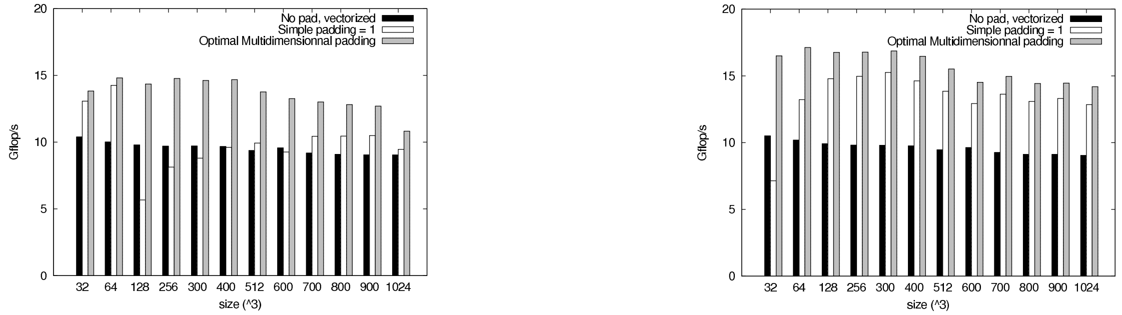
Some multi-threaded performance are displayed in Fig.8 for a 6-point 3D Jacobi compiled with icc, with simple precision data. The stencil shape is displayed in Fig.7(**b**). Each thread is pinned to a different core, and no hyperthreading is considered. The results are presented for a 256^3 matrix, since data are in memory, and this size is the one presented in most of stencil related papers. The multi-dimensional multipadding version is



(a) Results of Jacobi 2D with gcc

(b) Results of Jacobi 2D with icc

Fig. 9. Performance of a 2D Jacobi stencil on Intel's Westmere architecture for different square sizes.



(a) Results of Von-Neumann 3D with gcc

(b) Results of Von-Neumann 3D with icc

Fig. 10. Performance of stencil based on 3D Von-Neumann neighbourhood of range 2 on Intel's Sandy Bridge for different square sizes.

compared to the original version with no padding. The MDMP version outperforms the original version each time, with a 45% improvement for one thread, and up to a 69% improvement for eight threads.

C. Comparing with related works

In this section, results are in double precision and compared with results presented in related works, on a 7-point 3D Laplacian stencil computation. In Fig.11(**b**), we reported results shown in Kamil *et al.* [10], and presented our own performance figures obtained by applying a multipadding method (here, thread alignment and a multi-dimensional padding) for the 7-point 256^3 sized Laplacian stencil. Benchmarks for the Nehalem architecture (Fig.11(**b**)) have been realized on the exact same Gainestown architecture. Our method confirm its great results when in memory and for multiple cores. MDMP performances are more than 4x better with 2 and 4 cores than [10].

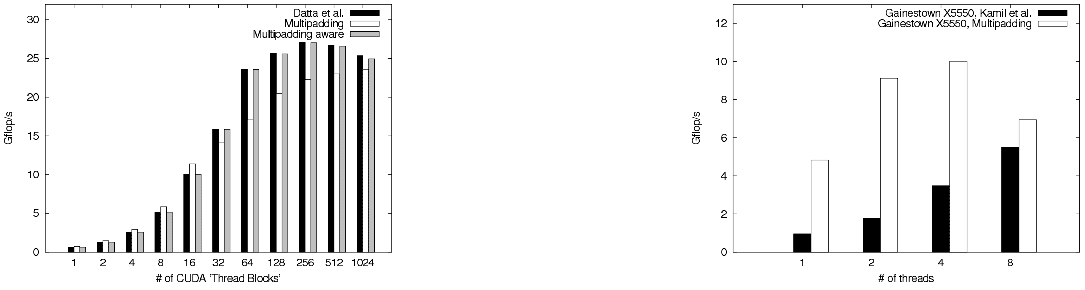
For the GPU figure, we were able to run the code from Datta *et al.* [4] on our machine, allowing to compare directly the results on the same basis. As our code generator does not produce CUDA code directly, only the code for the threads have been produced by the generator. All data transfers, and memory and thread allocations, are performed by hand. The white bars (labelled "Multipadding") give results when the code is generated taking into account CUDA programming guide's [21] recommendation: one stencil update per thread. However, in Datta *et al.* [4], it is stated to compute 4 stencil updates per thread. A code modification have been made subsequently, and performance results are displayed in grey bars (labelled "Multipadding aware"). As our first version can not withstand the comparison, the "aware" version successfully sustains around the same performance as Datta *et al.* [4], until beginning to take off for a great number of CUDA threads blocks (¿128).

VII. Conclusion

Stencil computations correspond to codes at the heart of a large class of applications. Due to the nature of stencils, alignment, memory management and vectorization are the key to performance, for GPU and CPU architectures alike. The contributions of this paper for stencil code generation are

- A data-layout transformation so as to handle CPU and GPU alignment issues for parallel stencils. This transformation called multi-dimentional multipadding introduces new elements in multi-dimensional structures and generalizes the usual padding transformation.
- A compact code generation for stencils. The data-layout transformation only requires to unroll partially loops and translate some indices of data structures.

Experimental results on multiple targets demonstrate the validity of the approach on several stencils with x1.72 speed-up on one core, and up to x1.69 speed-up on multicore. These



(a) Results on GPU Quadro 5800 FX architecture.

(b) Results on Nehalem architecture

Fig. 11. Performance of a 256^3 Laplacian stencil for multipadding method and (a) for Datta *et al.* [4] on GPU, (b) Kamil *et al.* [10], on Gulftown.Multipadding results are obtained with one computation/thread (CUDA programming guide recommendation). Multipadding aware results are obtained with 4 computations/thread (Datta *et al.* recommendation).

results outperform previous work results on multicore CPUs and are similar to those on GPUs.

References

[1] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Intl. Conf. on Compiler Construction*, pages 244–263, Paphos, Mar. 2010.

[2] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. of the VLDB Endowment*, 1(2):1313–1324, 2008.

[3] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *ACM Intl. Conf. on Supercomputing*, page 4, Austin, Texas, Nov. 2008.

[5] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par*, pages 642–653, 2009.

[6] H. Dursun, K.-I. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 533–538, 2009.

[7] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 82–93, 2004.

[8] L. Fireman, E. Petrank, and A. Zaks. New algorithms for simd alignment. In *Intl. Conf. on Compiler Construction*, Lect. Notes in Computer Science, pages 1–15, Braga, Portugal, Mar. 2007. Springer-Verlag.

[9] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Intl. Conf. on Compiler Construction*, pages 225–245, Saarbrücken, Germany, Mar. 2011.

[10] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IEEE Intl. Parallel and Distributed Processing Symposium*, pages 1–12, Atlanta, Georgia, Apr. 2010.

[11] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick. Implicit and explicit optimizations for stencil computations. In *Workshop on Memory System Performance and Correctness*, pages 51–60, San Jose, California, Oct. 2006.

[12] S. Krishnamoorthy, M. M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 235–244, San Diego, California, June 2007.

[13] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM/IEEE Intl. Symposium on Computer Architecture*, pages 451–460, Saint Malo, France, June 2010. ACM Press.

[14] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. on Programming Languages and Systems*, 26:2004, 2004.

[15] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T. fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 348–357, Raleigh, North Carolina, Sept. 2009.

[16] A. D. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *ACM Intl. Conf. on Supercomputing*, pages 1–13, New Orleans, LA, Nov. 2010. IEEE Computer Society.

[17] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor simd: Auto-vectorize once, run everywhere. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 151–160, Chamonix, France, Apr. 2011.

[18] D. Nuzman, M. Namolaru, A. Zaks, and J. H. Derby. Compiling for an indirect vector register architecture. In *ACM Computing Frontiers Conf.*, pages 199–208, Ischia, Italy, May 2008. ACM Press.

[19] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 132–143, Ottawa, Ontario, June 2006.

[20] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–11, Toronto, Ontario, Oct. 2008. ACM Press.

[21] Nvidia Corporation. Cuda programming guide. 2010.

[22] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *ACM Intl. Conf. on Supercomputing*. ACM Press, 2000.

[23] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *SPAA*, pages 117–128. ACM, 2011.

[24] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *CoRR*, abs/1004.1741, 2010.

[25] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 327–337, Raleigh, North Carolina, Sept. 2009.

[26] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *ACM Intl. Conf. on Supercomputing*, pages 244–255, Yorktown Heights, NY, June 2009. ACM Press.

[27] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(2):234–248, 2001.

[28] D. Wonnacott. Achieving scalable locality with time skewing. *Intl. J. of Parallel Programming*, 30(3):1–221, 2002.