High Level Transforms for SIMD and low-level computer vision algorithms

Lionel Lacassagne
Daniel Etiemble
Ali Hassan Zahraee
LRI Univ. Paris-Sud
firstname.name@lri.fr

Alain Dominguez
Intel Corporation
alain.dominguez@intel.com

Pascal Vezolle
IBM Corporation
pascal.vezolle@fr.ibm.com

Abstract

This paper presents a review of algorithmic transforms called High Level Transforms for IBM, Intel and ARM SIMD multi-core processors to accelerate the implementation of low level image processing algorithms. We show that these optimizations provide a significant acceleration. A first evaluation of 512-bit SIMD Xeon-Phi is also presented. We focus on the point that the combination of optimizations leading to the best execution time cannot be predicted, and thus, systematic benchmarking is mandatory. Once the best configuration is found for each architecture, a comparison of these performances is presented. The Harris points detection operator is selected as being *representative* of low level image processing and computer vision algorithms. Being composed of five convolutions, it is more complex than a simple filter and enables more opportunities to combine optimizations. The presented work can scale across a wide range of codes using 2D stencils and convolutions.

Keywords High Level Transforms, SIMD, Intel SSE & XeonPhi, IBM Altivec, ARM Neon, code optimization, 2D stencil, low-level computer vision and image processing algorithms.

1. Introduction

Graphic Processing Units (GPU) are efficient for High Performance Computing (HPC) [17] where the operations involved lend themselves to massive parallelization [7]. Some papers claim “orders-of-magnitude performance increase” versus General Purpose Processors (GPP). Recently, papers from Intel [11] and IBM [3] claim that GPPs can match GPUs if optimizations are applied. Some papers propose a fair benchmarking, by optimizing as much as possible the implementations on these architectures and compare them rigorously to find out the applications that really achieve significant speedups (like n-body [2] or stencil [4]).

The aim of this paper is to present some high level transforms (HLT) for SIMD applied to low-level image processing and computer vision algorithms. We focus on the fact that combining SIMD with OpenMP is not enough to reach and sustain a high level of performance. Optimizing memory accesses is an issue and HLT should be cache-aware and also *external-memory*-aware when data do not fit in the caches. In our study, the Harris operator [8] for point of interest detection is chosen. Widely used for image stabilization, velocity analysis or visual tracking, this operator is also a representative example of the regular low-level image processing algorithms class. As it is composed of 8 operators (Fig. 1), it enables more opportunities for optimizations and parallelization than a unique convolution kernel. HLT can be also efficiently applied to any code using 2D stencils.

The paper is organized as follows: the first section details the Harris operator, the software optimizations and HLT that can be applied to the Harris detector. The second section presents the targeted SIMD machines (with Altivec, SSE or Neon instruction set extensions) and a multistep benchmark that evaluates the impact of these optimizations. A first evaluation of XeonPhi is also presented. Then performance of GPPs and GPUs is compared.

2. Harris detector and HLT

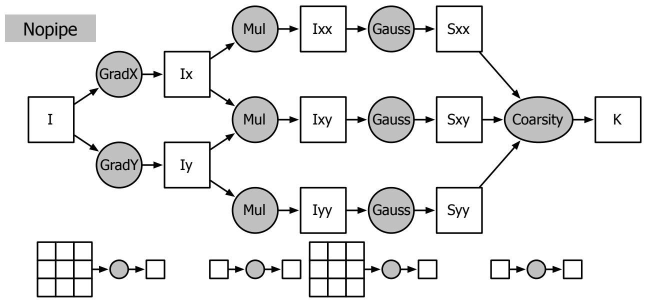

Figure 1. Harris detector, *Nopipe* version

The Harris detector computation relies on a set of point-to-point operations such as products and additions, along with (3×3) convolutions (the Sobel gradients and Gaussian smoothing). Given an input image I, the first derivatives I_x and I_y of Sobel gradients are computed by GradX and GradY. The cross-products $I_{xx} = I_x \times I_x$, $I_{xy} = I_x \times I_y$ and $I_{yy} = I_y \times I_y$ performed by the Mul operator. These cross-products are smoothed by a Gaussian filter (Gauss) and finally, “coarsity” K is a linear combination of the smoothed cross-products. A producer-consumer model is also provided under each figure with explicit input and output patterns: $(3 \times 3) \rightarrow (1 \times 1)$ for Grad and Gauss and $(1 \times 1) \rightarrow (1 \times 1)$ for Mul and coarsity. Moreover, considering the four stages of its classical computation, the so called *Nopipe* version (Fig.1) is an interesting candidate for higher level optimizations such as fusion of operators.

Optimizing a code is a two-step process. First, the code is optimized to reach the highest level of performance in a stressless context, when data fit in the cache. Next, performance is *sustained* in a stressed context when data do not fit in the cache. We first describe the classical compilation optimizations like *loop-unrolling*, *scalarization* and *reduction* for High Performance Computation. These techniques are then adapted to the low level operators in image processing: some high level algorithmic transformations named *Halfpipe* and *Fullpipe* are introduced. These optimizations are then improved by considering the application domain: both the separability of the 2D filters and the convolutions overlapping are considered and combined to loop-unrolling and scalarization to add *reduction* in *Halfpipe* and *Fullpipe* transforms. Their complexity is finally evaluated in terms of arithmetic operations and memory access. The second part presents an advanced memory layout transform, circular buffers and modular addressing associated with another level of operator pipelining to optimize spatial and temporal data locality. Finally SIMD *reduction* is detailed.

2.1 Classical transformations, decomposition and reduction

$$B_{3\times3} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{16}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \quad (1)$$

Algorithm 1: 1-pass implementation of the 3×3 binomial filter with the 2D-filter corresponding to equation (1)

```
for $i = 1$ to $n-1$ do
    for $j = 1$ to $n-1$ step 3 do
        $a_0 \leftarrow X(i-1,j-1), b_0 \leftarrow X(i-1,j), c_0 \leftarrow X(i-1,j+1)$
        $a_1 \leftarrow X(i,j-1), b_1 \leftarrow X(i,j), c_1 \leftarrow X(i,j+1)$
        $a_2 \leftarrow X(i+1,j-1), b_2 \leftarrow X(i+1,j), c_2 \leftarrow X(i+1,j+1)$
        $s \leftarrow 1a_0+2b_0+1c_0+2a_1+4b_1+2c_1+1a_2+2b_2+1c_2$
        $Y(i,j) \leftarrow s/16$
```

To facilitate the presentation of different algorithms, we assume that computations use local variables and that memory accesses correspond to transfers between image pixels and local variables. This assumption fits with RISC processors, for which the computations operate on register values and memory accesses correspond to LOAD and STORE instructions. In that case, local variables are held in processor registers. The situation is different for CISC processors using the IA-32 or Intel 64/AMD64 instruction sets for which memory operands are used by computing instructions. In that case, due to the limited numbers of registers, some local variables will be held in the memory hierarchy. However, this difference does not change the impact of the optimizations that are presented. In the rest of the paper, we present the transformations assuming a RISC instruction set.

The classical software optimizations for optimizing compilers [1] aim to improve the operation of the processor pipeline. Given a 3×3 binomial filter aka Gaussian filter (Eq. 1), one can perform *scalarization* (to put data into registers), and *register rotation* to avoid reloading data from an iteration of the filter to another. The 9 LOADs of the original algorithm (Alg. 1) are replaced by only 3 (Alg. 2). It is an efficient optimization, considering that many algorithms are memory-bound, but the algorithm complexity remains the same.

In this presentation of optimization techniques, we assume general filters with unknown coefficients: this is why we indicate multiplications by 1, 2 or 4, that are transformed into a set of additions ($4x = t+t$ with $t = x+x$, i.e. the *strength reduction* optimization) or replaced by a shift ($4x =$ `x<<2`) for integer computations. Note also that the apron processing problem is not considered in order to simplify the explanation.

Algorithm 2: 1-pass implementation of the 3×3 binomial filter with one 2D-filter Register Rotation

```
for $i = 1$ to $n-1$ do
    $j \leftarrow 1$
    $a_0 \leftarrow X(i-1,j-1), b_0 \leftarrow X(i-1,j)$
    $a_1 \leftarrow X(i,j-1), b_1 \leftarrow X(i,j)$
    $a_2 \leftarrow X(i+1,j-1), b_2 \leftarrow X(i+1,j)$
    for $j = 1$ to $n-1$ step 3 do
        $c_0 \leftarrow X(i-1,j+1)$
        $c_1 \leftarrow X(i,j+1)$
        $c_2 \leftarrow X(i+1,j+1)$
        $s \leftarrow 1a_0+2b_0+1c_0+2a_1+4b_1+2c_1+1a_2+2b_2+1c_2$
        $Y(i,j) \leftarrow s/16$
        $a_0 \leftarrow b_0, b_0 \leftarrow c_0$    // Rot
        $a_1 \leftarrow b_1, b_1 \leftarrow c_1$    // Rot
        $a_2 \leftarrow b_2, b_2 \leftarrow c_2$    // Rot
```

Taking into account the application domain, the 2D-filter can be replaced by two 1D-filters (Eq. 1, right part). This algorithmic transformation reduces complexity and number of memory accesses, but requires two passes on the image, which can generates cache misses, when the image is too large to entirely fit in the cache.

So we need to introduce another optimization to combine the two 1D-filters with a single pass, to factor the computations and reduce the number of memory accesses simultaneously. First, the result of the first 1D-filter is stored in a register (Alg. 3, line 5, 6 and 11). This transformation is called a *reduction*. In our case, it is a column-wise *reduction*. Then the second 1D-filter is directly applied to the *reduced* values (Alg. 3, line 12).

Algorithm 3: 1-pass implementation of the 3×3 binomial filter with two 1D-filters, with Register Rotation and *reduction*

```
1  for $i = 1$ to $n-1$ do
2      $a_0 \leftarrow X(i-1,j-1), b_0 \leftarrow X(i-1,j)$
3      $a_1 \leftarrow X(i,j-1), b_1 \leftarrow X(i,j)$
4      $a_2 \leftarrow X(i+1,j-1), b_2 \leftarrow X(i+1,j)$
5      $\mathbf{r_a} \leftarrow 1a_0 + 2a_1 + 1a_2$    // Red part #1
6      $\mathbf{r_b} \leftarrow 1b_0 + 2b_1 + 1b_2$    // Red part #1
7      for $j = 1$ to $n-1$ do
8          $c_0 \leftarrow X(i-1,j+1)$
9          $c_1 \leftarrow X(i,j+1)$
10         $c_2 \leftarrow X(i+1,j+1)$
11         $\mathbf{r_c} \leftarrow 1c_0 + 2c_1 + 1c_2$    // Red part #1
12         $s \leftarrow 1\mathbf{r_a} + 2\mathbf{r_b} + 1\mathbf{r_c}$    // Red part #2
13         $Y(i,j+0) \leftarrow s/16$
14         $\mathbf{r_a} \leftarrow \mathbf{r_b}, \mathbf{r_b} \leftarrow \mathbf{r_c}$    // Rot of reduced values
```

For a general 1D k-tap filter, there are still $(k-1)$ copies for *register rotation*. They can be entirely removed by *loop unrolling*. The order of the unrolling is usually chosen by the compiler, which may lead to suboptimal unroll. But in signal or image processing, the unrolling order has not to be chosen by heuristics: the optimal order leading to a perfect unroll is equal to the filter order. In our case, the smallest unrolling order is $k = 3$.

2.2 Operator fusion: the *Halfpipe* and *Fullpipe* transforms

The fusion of several operators is much more than a simple loop fusion (aiming to improve data locality). By pipelining operators and storing first results in registers instead of memory, the transform avoids any intermediate memory access. For that aim, each operator is described by the *producer-consumer* model with a consumption pattern and a production pattern. Such a model is derived from Synchronous Data Flow [10]. The only condition needed to fuse two operators (in the sense of mathematical composition of functions $f \circ g$) is that the patterns must be either similar or adaptive: the output pattern of the first operator should be the same as the input pattern of the second one.

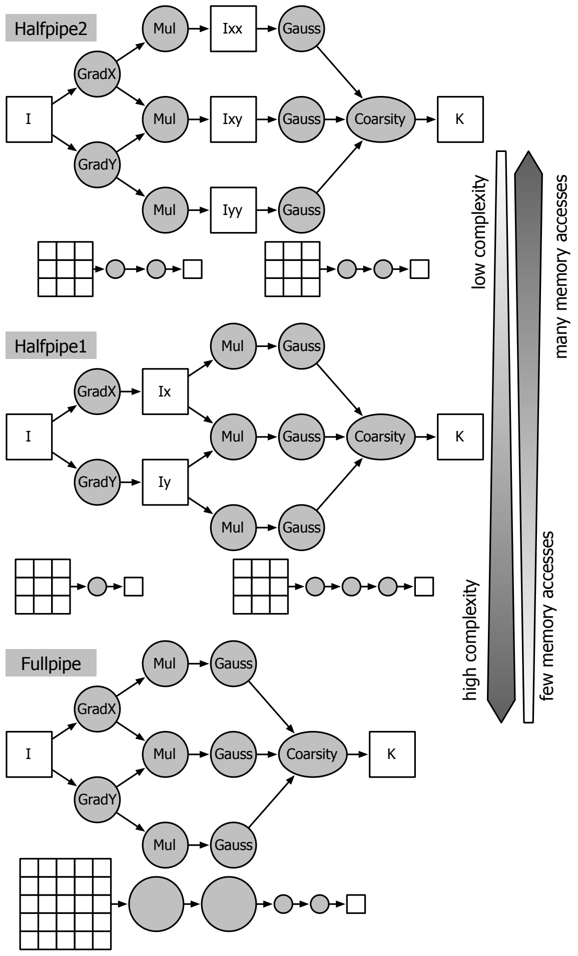

Figure 2. *Halfpipe* and *Fullpipe* transforms of Harris operator, from the lowest to the highest complexity

In its *Nopipe* version (Fig. 1), the Harris detector is composed of four computation stages: computation of the gradients, product of the gradients, smoothing of the products and computation of the point coarsity. This computation chain requires the access to eight intermediate arrays. It is therefore possible to *pipeline* the Sobel and Mul operators on the one hand, and the Gaussian and coarsity operators on the other hand, since their consumption and production patterns are compatible. The transformation is called *Halfpipe*$_2$ (Fig. 2, top) and reduces the number of memory accesses (Tab. 1).

It is possible to entirely pipeline the operators and suppress any intermediate memory access, resulting in the *Fullpipe* version (Fig. 2, bottom). In that case, it is necessary to adapt the consumption and production patterns: to produce (1×1) points in the output, it is necessary to consume (5×5) points and to perform the operations Sobel+Mul (3×3) times. A *reduction* along the columns is applied to optimize both the number of memory accesses and the number of computations.

Note that the *Fullpipe* version is much more complex than the other ones with up to 3 times more computations compared to the *Nopipe* version (Tab. 1). Indeed, without any intermediate image to store the common computations (which were performed 9 times by the Gaussian operators), these computations have to be reperformed.

The *Halfpipe*$_1$ version is designed to balance *Halfpipe*$_2$ and *Fullpipe*. Figure 2 shows that *Halfpipe*$_2$ has the lowest complexity, and the highest number of memory access. But once optimized (with *reduction*), *Halfpipe*$_2$ and *Halfpipe*$_1$ versions have similar complexities (Tab. 1) and memory accesses, while *Fullpipe* has a far lower number of memory accesses. If we consider the arithmetic intensity AI (ratio between computations and memory accesses) as a metric of performance, *Fullpipe* is particularly interesting from a parallelization perspective since its arithmetic intensity is higher than *Halfpipe* and *Nopipe* versions. *Fullpipe* version is *compute bound* while the two other versions are *memory bound* with different levels of memory stress.

version	MUL + ADD	LOAD + STORE	AI
without *reduction*			
Nopipe	5 + 44 = 49	48 + 9 = 57	0.9
Halfpipe$_2$	5 + 44 = 49	36 + 4 = 40	1.2
Halfpipe$_1$	29 + 44 = 73	27 + 3 = 30	2.4
Fullpipe	29 + 124 = 153	25 + 1 = 26	5.9
with *reduction*			
Nopipe+red	5 + 27 = 32	21 + 9 = 30	1.1
Halfpipe$_2$*+red*	5 + 27 = 32	12 + 4 = 16	2.0
Halfpipe$_1$*+red*	11 + 27 = 38	9 + 3 = 12	3.2
Fullpipe+red	29 + 82 = 111	5 + 1 = 6	18.5

Table 1. Algorithmic complexity, number of memory accesses and arithmetic intensity for Harris with HLT

2.2.1 Multithreading & cache overflow

For a multi-core SIMD processor, maximum performance is indeed reached when all cores are running. That means that evaluation of HLT for SIMD should be done when the code is multi-threaded. This corresponds to the configuration that maximizes the stress on the external memory bus. OpenMP is used for this purpose. The necessary code modification is very light. It mainly consists of *pragmas* that are simply put before the code sections to be parallelized. Considering a SPMD parallelization with a *sub-band* calculation, the modifications consist in parallelizing the outer loops, and privatizing variables to avoid serialization of accesses. The parallelization of a `for` loop only adds the directive (`#pragma omp parallel for`) to the sequential code.

Such parallelization may have a super-linear speedup. Let us introduce the measurement unit used for the analysis and the benchmarking: *cpp* (cycle per pixel). It is the number of clock cycles normalized by the number of pixels processed.

Figure 3 presents the *cpp* execution times of our benchmarks according to the size of the working set for different configurations of processors and caches. Let us first consider (Fig. 3a). On the horizontal axis, a point corresponds to the size of the working set, i.e. to the size of the data to be processed. As *cpp* is the number of clocks per pixel, a horizontal *cpp* curve means that the execution time, including computation and memory accesses, is exactly proportional to the data size. As a matter of fact, the actual *cpp* curve shows two plateaux and a transition area from one plateau to the other. This transition corresponds to the switch

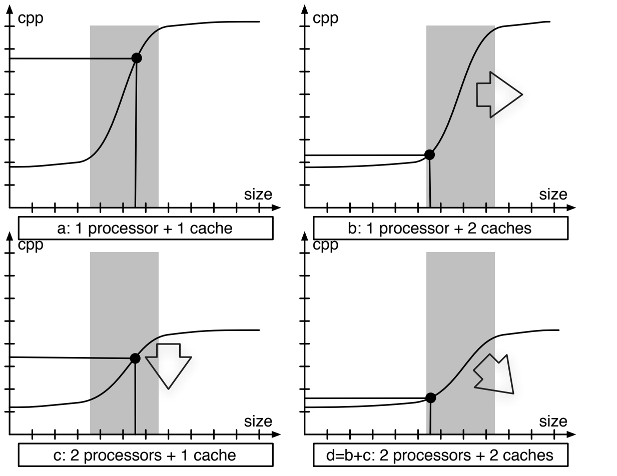

Figure 3. Expected transformations impact: evolution of the operating point according to the number & size of caches, cache overflow (in gray) and number of processors

between a state for which the working set can be held in the cache to another one for which the working set is larger than the cache size. We call this phenomenon a *cache overflow*. This phenomenon is widely detailed in [5].

Using more processors reduces execution time, even when the Amdhal law forbids a perfect speedup (Fig. 3c). Adding caches (or using a larger cache) would shift right the *cpp* curve and results are improved (Fig. 3b). For multi-core processors, the impact resulting from the combination of these two modifications is clearly visible (Fig. 3d). *Multithreading* with multi-core processors increases the available cache size and can be seen as a mean to postpone cache overflow. The performance decrease is more important for larger working sets. Note that the memory optimizations can lead to a super-linear speedup. For a given working set, it appears when the basic version (a) does not fit in the cache while the optimized one (d) does.

2.3 Data interleaving memory optimization

The data interleaving consists in replacing the accesses to several different arrays by accesses to one single array containing the same data [15]. This is the *SoA-AoS* transformation (Structure of Arrays *vs* Array of Structures). For the Harris detector, the arrays that are produced at the same computation stages are interleaved, namely I_x et I_y for the Sobel operator, I_{xx}, I_{xy} and I_{yy} for Mul and S_{xx}, S_{xy} et S_{yy} for the Gaussian operators. This optimization is particularly efficient when the number of arrays to manipulate is lower than the associativity of the memory caches, therefore avoiding systematic cache misses. For scalar computations, the interleaving is 1:1 and for SIMD the interleaving is 4:4 (that is 1 vector of 4 points for 1 vector of 4 points) to avoid adding SIMD instructions to deinterleave the data. For example we have a full SIMD register of I_x then a full SIMD register of I_y.

2.4 Circular buffers with modular addressing optimization

If array interleaving helps avoid systematic cache eviction, cache overflow for large data sets can not be avoided. The circular buffers with modular addressing optimization are designed to address this problem.

Circular buffers are widely used in embedded applications (typically signal processing on DSPs) since they reduce the memory required to store intermediate results and improve the performance

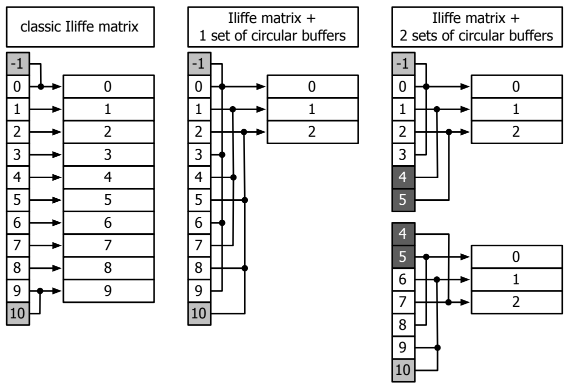
classic Iliffe matrix
Iliffe matrix + 1 set of circular buffers
Iliffe matrix + 2 sets of circular buffers
-1
0
1
2
3
4
5
6
7
8
9
10

Figure 4. Iliffe matrix with 0, 1 or 2 sets of three circular buffers for an image made of 6 rows and a border of 1, for a 3×3 convolution.

of the memory caches. The principle is to chain the different operators and store into the memory the intermediate data needed for the sequential execution of the operators.

Circular buffers (CB) can be transparently introduced into user code thanks to Iliffe pointer (offset addressing) [9] popularized by Numerical Recipes in C matrix [13] (Fig. 4, left). So, for a given point of coordinates (i, j), `T[i]` is pointing to the row $(i \bmod k)$ instead of (i) and j is the offset within the row (Fig. 4, center). Such a spatial locality optimization can also be used in a multithreaded context using OpenMP, but with some restrictions. First the number of threads should be known before the memory allocation, because the number of sets of circular buffers should be equal to p, the number of created threads. Hence, the number of threads cannot be dynamically changed (except if a buffer reallocation is performed). Second, the loop space should be split into p blocks of contiguous lines: OpenMP `parallel` macro should be tuned by specifying the thread policy with `schedule static`. Finally, p sets of k circular buffers should be allocated for an execution on p processors with a $k \times k$ convolution. Because of the overlapping of addressing due to the convolution kernel, the sets of circular buffers do not define a bijective application. On the right part of figure 4, the lines addressing the internal aprons (in dark gray) are duplicated while the external aprons (in light gray) are pointing to the first or last line of the block.

In the *Halfpipe* case (*Halfpipe*$_1$ and *Halfpipe*$_2$), rather than applying the Sobel and Mul operators to the entire image, this algorithm is first executed on the first two rows of the image (prolog). Once arrived at the third row, the first two operators have produced enough data so that the two subsequent operators can start their own work. Therefore, the two groups of two operators are chained together. If the *Halfpipe* transformation can be viewed as the pipeline of two operators, the use of circular buffers with *modular* addressing can be viewed as the *pipeline of two tiles of data* with cache-blocking. In the following, we call this optimization mod.

2.5 HLT adaptation to SIMD computations

Optimizing a convolution with SIMD instructions relies on the optimization of memory accesses and the management of unaligned vectors. For example, for the 1D horizontal Gaussian filter, $X(i-1) + 2X(i) + X(i+1)$ implies the access to the left unaligned and right unaligned data. The naive way is to perform an *unaligned load* using `_mm_loadu_ps` instruction when available (only on SSE, not on Altivec nor Neon). It is not efficient as 3 LOADs are required and cannot be combined with register rotation. The efficient way is to build unaligned vectors from aligned vectors (Alg. 4, lines 10, 11 and 12). For SIMD portability, Two macros named `vec_left` and `vec_right` encapsulate the instructions: `vec_sld` for Altivec, `vextq_f32` for Neon and `_mm_shuffle_ps` for SSE and SSE2 extension or `_mm_alignr_epi32` for SSSE3+ extension.

Algorithm 4: 1-pass implementation of the 3 × 3 binomial filter with one 2D-filter with Register Rotation

```
1  for $i = 1$ to $n-1$ do
2      $j \leftarrow 1$
3      $a_0 \leftarrow X(i-1, j-1), b_0 \leftarrow X(i-1, j)$
4      $a_1 \leftarrow X(i, j-1), b_1 \leftarrow X(i, j)$
5      $a_2 \leftarrow X(i+1, j-1), b_2 \leftarrow X(i+1, j)$
6      for $j = 1$ to $n-1$ do
7          $c_0 \leftarrow X(i-1, j+1)$
8          $c_1 \leftarrow X(i, j+1)$
9          $c_2 \leftarrow X(i+1, j+1)$
10         $a'_0 \leftarrow \texttt{vec\_left}(a_0, b_0), c'_0 \leftarrow \texttt{vec\_right}(b_0, c_0)$
11         $a'_1 \leftarrow \texttt{vec\_left}(a_1, b_1), c'_1 \leftarrow \texttt{vec\_right}(b_1, c_1)$
12         $a'_2 \leftarrow \texttt{vec\_left}(a_2, b_2), c'_2 \leftarrow \texttt{vec\_right}(b_2, c_2)$
13         $s \leftarrow 1a'_0 + 2b_0 + 1c'_0 + 2a'_1 + 4b_1 + 2c'_1 + 1a'_2 + 2b_2 + 1c'_2$
14         $Y(i, j) \leftarrow s/16$
15         $a_0 \leftarrow b_0, b_0 \leftarrow c_0$ // $Rot$ of 1st line
16         $a_1 \leftarrow b_1, b_1 \leftarrow c_1$ // $Rot$ of 2nd line
17         $a_2 \leftarrow b_2, b_2 \leftarrow c_2$ // $Rot$ of 3rd line
```

Such a technique is adaptable to the scalar column-wise reduction optimization (Alg. 3). Instead of using the *load-permute-add* sequence, we can swap the *permute* (4, lines 10, 11 and 12) and *add* steps: reductions are computed on aligned data for the vertical 1D filter, then the unaligned vectors are built (Alg. 5, lines 12 and 13). As the `vec_left` and `vec_right` instructions are applied to reduced registers, fewer are needed: from 6 in the version without *reduction* down to 2 in the version with *reduction*. That computation scheme makes the SIMD reduction possible and very efficient as it can be combined with register rotation or loop-unrolling.

Algorithm 5: 1-pass implementation of the 3 × 3 binomial filter with two 1D-filter, with Register Rotation and *reduction*

```
1  for $i = 1$ to $n-1$ do
2      $a_0 \leftarrow X(i-1, j-1), b_0 \leftarrow X(i-1, j)$
3      $a_1 \leftarrow X(i, j-1), b_1 \leftarrow X(i, j)$
4      $a_2 \leftarrow X(i+1, j-1), b_2 \leftarrow X(i+1, j)$
5      $\mathbf{r_a} \leftarrow 1a_0 + 2a_1 + 1a_2$
6      $\mathbf{r_b} \leftarrow 1b_0 + 2b_1 + 1b_2$
7      for $j = 1$ to $n-1$ do
8          $c_0 \leftarrow X(i-1, j+1)$
9          $c_1 \leftarrow X(i, j+1)$
10         $c_2 \leftarrow X(i+1, j+1)$
11         $r_c \leftarrow 1c_0 + 2c_1 + 1c_2$
12         $r'_a \leftarrow \texttt{vec\_left}(r_a, r_b), r'_c \leftarrow \texttt{vec\_right}(r_b, r_c)$
13         $s \leftarrow 1r'_a + 2r_b + 1r'_c$
14         $Y(i, j) \leftarrow s/16$
15         $r_a \leftarrow r_b, r_b \leftarrow r_c$    // $Rot$ of reduced values
```

3. Benchmarking

3.1 Targeted SIMD processors

Three SIMD extensions are evaluated : SSE to SSE4 for Intel, Altivec (VMX) for IBM and Neon for ARM (Tab. 2). For each architecture the company's compiler is used : icc, xlc and armcc. To simplify the benchmark, and also allow a fair comparison with Neon, only the 128-bit SIMD extension is evaluated. Even with such a limitation, the hardware of 256-bit AVX Sandy/IvyBride processors can pair two identical 128-bit instructions into a 256-bit instruction. The second restriction is that FMA (*Fused Multiply-Add*) is not used, as this instruction is not present on all architectures. But the main point limiting the performance is the bandwidth, not the lack of FMA. In order to get a multi-architecture code, both kernel computations and load/store instructions are rewritten with macros. A header file holds the translation for each SIMD instruction set. This feature ensures that the same algorithm is executed on all architectures. We plan to rewrite the code with Boost.SIMD soon [6].

processor	nb cores	freq (GHz)	perf. GFlops	BW GB/s	AI ratio
Cortex A9 OMAP4	1 × 2	1.2	4.8	1.2	4.0
Cortex A15 Exynos5	1 × 2	1.7	13.6	5.8	2.3
PowerPC 970MP	2 × 2	2.5	40.0	5.4	7.4
Power6	2 × 2	4.0	64.0	15.1	4.2
Power7	4 × 8	3.8	486	265	1.8
Penryn X3370	2 × 4	3.0	96.0	15	6.4
Nehalem X5550	2 × 4	2.67	85.1	22	3.9
Westmere X5680	2 × 6	3.33	159.8	25	6.4
IvyBridge E5-2697v2	2 × 12	2.7	518.4	92	5.6
Xeon Phi	1 × 61	1.33	1298	170	7.6

Table 2. Main characteristics of the evaluated machines.

3.2 Halfpipe and Fullpipe impact with reduction

3.2.1 Impact of *reduction*

HLT	Penryn			Nehalem		
	without red	with red	gain	without red	without red	gain
before cache overflow (in 512 × 512)						
Nopipe	**3.76**	2.62	×1.4	**3.40**	3.36	×1.0
Halfpipe$_2$	2.96	2.04	×1.5	1.86	1.56	×1.2
Halfpipe$_1$	3.42	**1.50**	×2.3	1.76	**1.17**	×1.5
Fullpipe	5.75	4.10	×1.4	3.92	2.92	×1.3
total gain	$\times\mathbf{2.5}$			$\times\mathbf{2.9}$		
after cache overflow (in 2048 × 2048)						
Nopipe	**62.50**	62.50	×1.0	**10.50**	10.50	×1.0
Halfpipe$_2$	27.10	26.90	×1.0	4.47	4.43	×1.0
Halfpipe$_1$	19.10	19.10	×1.0	3.42	3.30	×1.4
Fullpipe	5.80	**4.20**	×1.4	3.99	**2.95**	×3.6
total gain	$\times\mathbf{14.9}$			$\times\mathbf{3.6}$		

Table 3. Impact of *reduction* for Penryn and Nehalem

Let us first compare the impact of HLT without and with *reduction* and focus on Penryn and Nehalem processors. Before cache overflow (Tab. 3), Penryn and Nehalem behavior and *cpp* are very similar: red provides a speedup in the range of $[\times 1.3 : \times 2.3]$ and the HLT total gains (*Nopipe* versus *Halpipe*$_1$ are also close: $\times 2.5$ and $\times 2.9$. After cache *overflow*, red is inefficient for *Halfpipe* versions. As *Fullpipe* version is memory bound, the *cpp* values remain unchanged and red oprimization still provides a speedup of $\times 1.4$. Because of the difference in memory bus (FSB for Penryn, QPI for Nehalem) the impact of cache *overflow* on *Nopipe* version is very different: the Penryn slowdown is 16.6 whereas Nehalem one is only 3.1. The total HLT gain increases a little for Nehalem : $\times\mathbf{3.6}$ and a lot for Penryn $\times\mathbf{14.9}$!

One important point is that the best transform changes with the data set size. Before cache overflow the best transform is the

Halfpipe$_1$ and after, it is the *Fullpipe* version. This result is different for IBM and ARM processors where *Halfpipe* is always the best optimization (see next paragraph).

3.2.2 Impact of vectorization

Concerning the vectorization, C99 source code is used with the `restrict` qualifier to enforce the fact that there is no pointer aliasing. For 16, 32 or 64 byte alignments, `_mm_malloc` is used for Intel, `posix_memalign` for ARM. For IBM, the alignment of classical `malloc` is the size of Altivec registers (when activated). Under these conditions, the relative vectorization speedup goes from 0% (cannot vectorize due to data dependency) for *red* versions up to 90% (quite perfect vectorization) for the version without *reduction*. In fact, column-wise *reduction* is too restrictive for the compiler. Regarding the vectorization capability of compilers, it appears that only HTL without red optimization is vectorized with a performance very close to the hand-coded SIMD version. If data do not fit in the cache, there is no need to apply red optimization nor to SIMDize the code. Otherwise, code SIMDization combined with HLT with reduction is efficient. So the "ninja gap" [14] between the basic version (vectorized and parallelized by the compiler - we assume efficient compiler parallelization) and the best handcrafted version is equal to the total HLT gain here. The impact of data interleaving is not detailed in the paper. The speedup is about 20% when there is no *reduction* and falls to a few percents when *reduction* is active. The impact of manual *loop-unrolling* compared to unrolling by the compiler is not evaluated either. Compiler unrolling option is set for all compilers and architectures.

3.2.3 Impact of *Halfpipe* and *Fullpipe* transforms

The three processor families, have the same kind behavior. First, all *Halfpipe* versions are memory-bound: a cache overflow occurs when the bandwidth is too much stressed. Secondly, *Halfpipe*$_1$+red is always faster than *Halfpipe*$_2$+red: even for ARM and Power processors that have a smaller AI, it is more efficient to reduce the number of memory accesses than the algorithmic complexity.

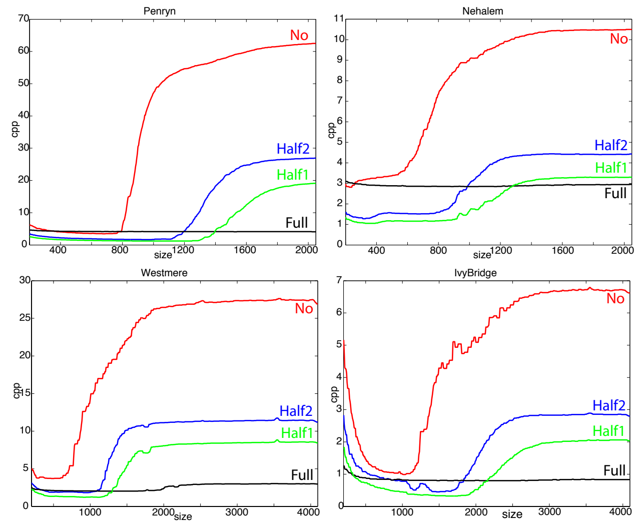

Figure 5. Penryn, Nehalem, Westmere and IvyBridge *cpp*

The four generations of Intel processors have very close behavior (Fig. 5). The only differences are the scale (related to core number and bandwidth of the processor) of the graph and the data set size when a cache *overflow* occurs. After overflow, *Fullpipe*+red is faster than *Halfpipe*$_1$+red.

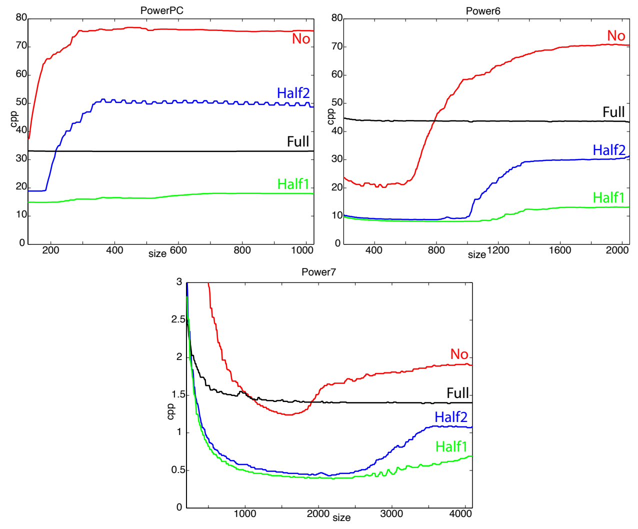

Figure 6. PowerPC, Power6 and Power7 *cpp*

For IBM processors (Fig. 6), depending on AI, *Fullpipe*+red is faster (for PowerPC) than *Halfpipe*$_2$+red, but never outperforms *Halfpipe*$_1$+red.

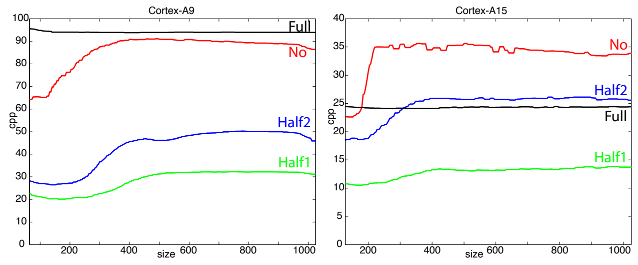

Figure 7. Cortex-A9 and Cortex-A15 *cpp*

ARM Cortex-A9 and A15 processors (Fig. 7) have also a similar behavior. But the scales are very different: the Cortex-A15 has around a twice smaller *cpp* than Cortex-A9. It has a globally improved architecture compared to A9. Moreover A15 can execute one SIMD Neon instruction every cycle instead of one instruction every two cycles for A9.

3.3 Impact of circular buffers and *modular* addressing

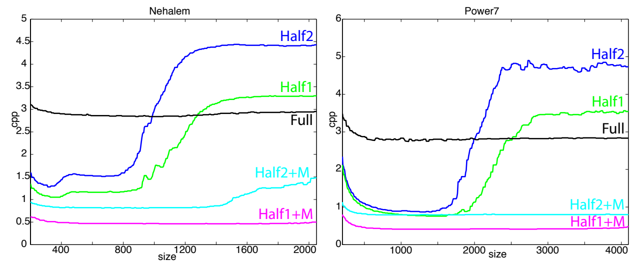

Figure 8. HLT with red versus HLT with red+mod

To address cache overflows and to prolongate HLT performance after overflow, mod optimization has been implemented. If a cache overflow should occur, mod will only postpones it to bigger data set, as only $3p \times n \times$ sizeof(`float`) bytes should stay in the cache instead of $n \times n \times$ sizeof(`float`) for each image (3 being the size of the convolution, p the number of threads and n the image size). That makes the *Halfpipe*$_1$+*red*+*mod* alway the best version as it requires only two arrays (I_x and I_y) to stay in the cache, instead of three (I_{xx}, I_{xy} and I_{yy}) for *Halfpipe*$_2$+*red*+*mod* (Fig. 8).

When there is no cache overflow or before it happens, the *mod* optimization provides an extra speedup on *red* version close to $\times 2$. After cache *overflow*, the *mod* speedup is in the range $[\times 1.8 : \times 6.3]$ (Tab. 4). The higher AI the higher speedup: the max is reach for Westmere that has the same bandwidth than Nehalem, but 50% more cores. As no compiler can vectorize scalar *red* + *mod* versions, it is mandatory to write them manually, but it's worth it. Finally, compared to the *Nopipe* SIMDized+OpenMPized version, the total gain of HLT is in the range $[\times 6.1 : \times 89.3]$. Other said, without HLT, the available power after overflow is only 16.4% downto 1.1% of the peak power available before.

	cpp			HLT speedup		
processor	*No*	*red*	*mod*	*red*	*mod*	tot
Cortex A9	86.40	31.24	9.41	×2.8	×3.3	×9.2
Cortex A15	34.08	13.88	**5.55**	×2.5	×2.5	×6.1
PowerPC	75.7	18.0	10.20	×4.2	×1.8	×7.4
Power6	70.8	13.2	4.39	×5.4	×3.0	×16.1
Power7	1.62	0.40	**0.21**	×4.1	×1.9	×7.7
Penryn	62.5	4.10	0.70	×**15.2**	×5.9	×**89.3**
Nehalem	10.5	2.95	0.50	×3.6	×5.9	×21.0
Westmere	26.6	2.65	0.42	×10.0	×**6.3**	×63.3
IvyBridge	5.30	0.65	**0.15**	×8.2	×4.3	×35.3
Xeon Phi	0.99	**0.18**	-	×5.5	-	-

Table 4. *cpp*, *red* and *mod* speedups after cache *overflow*

3.4 A first evaluation of Xeon Phi

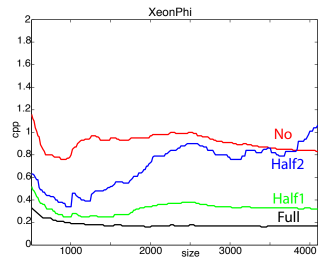

Figure 9. Xeon Phi: *Halfpipe* and *Fullpipe* *red* versions

	No	*Half*$_2$+*red*	*Half*$_1$+*red*	*Full*+*red*	gain
cpp	0.99	0.81	0.35	**0.18**	×5.5
GFlops	65.8	52.5	144.4	820.2	
BW (GB/s)	306.3	105.1	182.4	177.3	

Table 5. Xeon Phi performance for HLT+*red* optimizations

The codes have been easily ported on the Xeon Phi: Compilation tools and computing model are same as classic processors. As our codes use macro to handle 3 SIMD dialects, the only modification was to replace `_mm_` intrinsics prefix by `_mm512_`, and set the SIMD cardinal to 16 instead of 4. FMA is not used in order to be consistent with other processors. Because its arithmetic intensity is very high, *Halfpipe*$_1$+*red* is faster than *Halfpipe*$_2$+*red*. For the same reason, *Fullpipe*+*red* is faster to *Halfpipe*$_1$+*red* and is actually the fastest version. The *mod* transform is currently inefficient. More investigations are required with Vtune to understand the problem and fix it. Anyway compared to *one* IvyBridge Xeon, the Phi has a better *cpp* and – due to CPU frequency difference – an execution time that is 25% longer. When the bandwidth (Tab. 5) exceeds stream triad performance, it is not an error but the evidence that some data still remain in the caches.

3.5 Conclusion for SIMD processors

As previously said, the first step to accelerate a code is to optimize when data fit in the cache. This is the goal of HLT with *red* optimization. While the second step is to sustain the performance after cache *overflow*: that is the goal of the additional *mod* optimization. Table 4 provides *cpp* for HLT+*red* optmization before *overflow* and HLT+*red*+*mod* after *overflow*. In fact *mod* only provides an additional speedup of $\times 2$, but it prevents performance deceleration. Without it, the computing performance delivered for big images (2048×2048) by the evaluated parallel machines is only 16.3 % down to 1.1 % of the performance delivered for small images (512×512). HLT are mandatory.

It appears that Intel IvyBridge and IBM Power7 have very close *cpp* performance, despite the architecture differences: 2×12 cores versus 4×8, 256-bit AVX versus 128-bit Altivec. Power7 is faster with *red* optimization and IvyBridge is faster for *mod* optimization. From an execution point of view they have the same execution time 0.23 ms. We can also notice the impact of HLT transforms: the higher AI the high impact. Even if latest processors have a better memory architecture – that will decrease the impact of HLT compared to previous processor – it is still very important: $\times 7.7$ for Power7 and $\times 35.3$ for IvyBridge. For Cortex, impact should increase with latest quad-core processor.

3.6 Multi-core SIMD GPP versus GPU

execution time (ms) of SIMD multi-core and many-core GPP					
	Nopipe	*H+R*	*F+R*	*H+R+M*	gain
Cortex-A15	84.08	34.2	60.3	13.69	×6.1
IvyBridge	8.23	1.01	1.26	**0.23**	×35.3
Xeon Phi	3.12	1.10	**0.57**	-	×5.5
Power 7	**1.79**	**0.44**	1.56	**0.23**	×7.7
execution time (ms) of GPU					
	Nopipe	*H*	*F*	*F*	
	Global	*Tex*	*Tex*	*Shared*	gain
GTX 580	6.52	2.24	1.40	1.16	×5.6
GTX Titan (est.)	*2.29*	*0.79*	*0.49*	***0.41***	×5.6
K40 (est.)	*2.41*	*0.83*	*0.52*	*0.43*	×5.6

Table 6. Global comparison of State-Of-The Art GPP and GPU for 2048×2048 images (*H* and *F* stand for *Halpipe* and *Fullpipe* transforms, *R* and *M* for *red* and *mod* optimizations).

In order to make a *fair* comparison, we also applied these HLT to Nvidia GPU with CUDA 4. There are three kinds of memory: *Global* memory that is shared by all threads of all SM processors, *texture* memory that provides hardware bilinear interpolation and *shared* memory that is private to a SM processor. Texture usage reduces both the complexity and the number of memory accesses of the convolution. For example `X[i-1]+2*X[i]+X[i+1]` is equal to `0.5*(X[i-0.5]+X[i+0.5])`. 2D-convolutions only require 4 LOADs instead of 9, and 3 ADDs instead of 8 ADDs and 5 MULs. For *shared* memory, the Nvidia example "convolution2D" is specialized to a 3×3 convolution wit loop unwinding. As GPU are very sensitive to tile size, an exhaustive search of the best tile has been done for all benchmarked GPU (GTX 285, GTX 480, GTX 580, Quadro 4000). Usually 16×8 is a nice size for many algorithms, but the best one provide a boost up to 50 %.

Profiling helps to explain the *Shared* performance: the occupancy is 25 % and there are 41 registers per thread. The reason is that the GPU uses a lot of Shared memory. Two other metrics are interesting: the ipc (instruction per cycle) equals 1.85 (for a max of 2) and the L1 hit rate is 99.6 %. That clearly means that we are very close to peak performance, with a quite optimal pipeline feed. Note that the other tested configurations have a better occupancy, but are slower. So, we are confident that this configuration is globally optimal, and that "Fermi has better performance at lower occupancy" [16] (if there is enough *computations*). We can see (Tab. 6) that HLT transforms are also efficient for GPU: $\times 5.6$. Moreover, the *Fullpipe* versions (with texture memory and interpolation, or with shared memory) are faster than *Halfpipe* (version that requires two kernels instead of one for the *Fullpipe*). That focuses on the importance of avoiding communication and synchronization on GPU (*Halfpipe* requires two `__syncthreads` instead of one for *Fullpipe*).

To perform a State-of-the Art comparison (Tab. 6), GTX Titan and K40 performance have been estimated according to the clock frequency and cores numbers. We do not take into account PCI transfer duration for Xeon Phi nor for GPU. If we compare the performance without any optimization, (*Nopipe* column), the GPUs are faster than the GPPs. But if we compare the most optimized version, GPPs match GPUs performance thanks to mod optimization. Only Xeon Phi can comes close to GPUs without the mod optimization.

4. Main conclusion and future works

In this paper, we have presented algorithmic High Level Transforms for SIMD General Purpose Processors applied to low-level computer vision algorithms. HLT have a major impact on performance, and make the difference. They can be applied to any code using 2D stencils or convolution and so, can scale across a wide range of codes

The combination of operator fusion/pipelining with algorithmic *reduction* and circular buffers with *modular* addressing provide huge speedups: from $\times 6.1$ for dual Cortex-A15 up to $\times 89.3$ for Penryn (that has a small bandwidth). For State-of-the-Art processors with a more important bandwidth like IvyBridge and Power7, the respective speedups are $\times 35.3$ and $\times 7.7$. Xeon Phi is easy to program thanks to SIMD intrinsics and OpenMP and match other processors. These benchmark optimizations have clarified several important points. First of all, SIMDization is really effective on the SIMD multi-core machines and is the only way to match GPU performance. Secondly, considering the fusion of operators, the *Halfpipe* and *Fullpipe* transforms provide consequent additional gains. Since these transformations remain out of reach even for the best current compilers [12], the manual coding is fully justified. Third, the algorithmic *reduction* and the use of *modular* addressing is beyond the scope of compilers as such transforms modify the code semantic and should be also handed-coded.

Depending on the user skills and the required level of performance, there are finally two choices. On one hand, the *cost-effective* implementation is to combine the compiler auto-vectorization with openMP. It may be sufficient for some applications with soft real-time constraints – especially if data fit in the caches. On another hand, the *most-effective* version is to combine SIMDization with *reduction* and *modular* addressing. The circular buffers enforce spatial and temporal locality that are mandatory to sustain performance for processors that have a cache overflow. It is a significant modification, but it is a worthwhile modification as it ensures a high and *quite* constant level of performance, without cache overflow for any realistic image size.

In future works, we will finalize the on-going benchmarks on Intel Xeon Phi and Nvidia Kepler GPUs. The general-purpose code will be specialized to take into account the specificities of processors (larger SIMD, FMA, for GPP, CUDA 5 and communications for GPU) in order to provide more accurate results. More important, we also plan to implement the algorithmic *reduction* into a DSL (Domain Specific Language) above Boost.SIMD for signal/image/stencil operators. Concerning *circular buffers* and *modular addressing*, the memory layout modification should be also embedded into a tool that capture such abstraction to combine it with the semantic of image and multidimensional stencil operators.

References

[1] R. Allen and K. Kennedy, editors. *Optimizing compilers for modern architectures: a dependence-based approach*, chapter 8,9,11. Morgan Kaufmann, 2002.

[2] N. Arora, A. Shringarpure, and R. W. Vuduc. Direct n-body kernels for multicore platforms. In *International Conference on Parallel Processing*, pages 379–387. IEEE, 2009.

[3] R. Bordawekar, U. Bondhugula, and R. Rao. Can cpus match gpus on performance with productivity? technical report rc25033, IBM, 2010.

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Super Computing*, pages 1–12. ACM/IEEE, 2008.

[5] U. Drepper. what every programmer should know about memory. technical report, Red Hat, 2007.

[6] P. Estérie, M. Gaunard, J. Falcou, J.-T. Lapresté, and B. Rozoy. Boost. simd: generic programming for portable simdization. In *International Conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012.

[7] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28:13–27, 2008.

[8] C. Harris and M. Stephens. A combined corner and edge detector. In *4th ALVEY Vision Conference*, page 0. Editions Hermes, Paris, 1988.

[9] J. Iliffe. The use of the genie system in numerical calculation. *Annual Review in Automatic Programming*, 2:1–28, 1961.

[10] E. Lee. Multidimensional streams rooted in dataflow. In *IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 1993.

[11] 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 *International Symposium on Computer Architecture*, pages 451–460. ACM, 2010.

[12] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, 2008.

[13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C book set: Numerical Recipes in C: The Art of Scientific Computing*, chapter 1, pages 20–23. Cambridge, 1992.

[14] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications ? In *International Symposium on Computer Architecture*, pages 440–451. ACM, 2012.

[15] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 322–329. ACM, 1998.

[16] V. Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 2010.

[17] V. Volkov and J. Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, May 2008.