Optimizing 3D Convolutions for Wavelet Transforms on CPUs with SSE Units and GPUs

Brice Videau, Vania Marangozova-Martin, Luigi Genovese, Thierry Deutsch

Optimizing 3D Convolutions for Wavelet Transform son CPUs with SSE Unitsand GPUs

Brice VIDEAU, Research Scientist, LIG, CNRS, France
Vania MARANGOZVA-MARTIN, Associate Professor, LIG, Grenoble University (UJF), France
Luigi GENOVESE, Research Scientist, L_Sim INAC SP2M, CEA, Grenoble
Thierry DEUTSCH, Senior Research Scientist, L_Sim INAC SP2M, CEA, Grenoble

RR-LIG-032
février 2013

http://rr.liglab.fr
ISSN 2105-0422

Optimizing 3D Convolutions for Wavelet Transforms on CPUs with SSE Units and GPUs

Brice Videau and Vania Marangozova-Martin
Nanosim (LIG), France
first.last@imag.fr

Luigi Genovese and Thierry Deutsch
CEA - Grenoble (L_Sim INAC SP2M), France
first.last@cea.fr

Abstract—**Nanosimulations present a big HPC challenge as they present increasing performance demands in heterogeneous execution environments. In this paper, we present our optimization methodology for BigDFT, a nanosimulation software using Density Functional Theory. We explore autotuning possibilities for BigDFT's 3D convolutions by studying optimization techniques for several architectures. Namely, we focus on processors with vector units and on GPU acceleration. We report on the portability and the performance gains of our approach (speedup x2 on CPU, x5 on GPU) and discuss the relation between algorithmic specifics, architecture and performance.**

Keywords-**3D Convolutions, Optimization, GPU, Vectorization, OpenCL, Code Tuning, Portability**

I. INTRODUCTION

Obtaining good application performance on a high performance computing (HPC) platform becomes a real challenge. Indeed, the HPC hardware landscape becomes very complex and each architecture brings its performance specifics. For instance, the behavior of the Nehalem processors is very different from that of NVIDIA GPUs. The first have a small number of general purpose high performance cores with vector units. The later have many specialized cores with no vector units. The first need coarse grain parallelism to be efficiently used while the later can manage much finer grain parallelism. CPUs have a cache hierarchy that can be transparently used while GPUs have a cache that needs to be explicitly managed and can prove inefficient if not used correctly. As a consequence, it is impossible to obtain satisfactory performances on different architectures with the same implementation and optimization of an algorithm.

In this paper we study the implementation, code restructuring and optimization of 3D convolutions. We have chosen convolutions as they are key components of BigDFT [1], a software for simulating the electronic properties of future nano materials. The optimization of the BigDFT code on HPC platforms and especially the definition of autotuning strategies are two major goals of the joint work between the Nanosim team of the LIG laboratory and the L_Sim team of CEA-Grenoble.

Our methodology for evaluating the performance and define tuning strategies for 3D convolutions is the following.

- *Choose a Target Architecture*
  We have worked on two target architectures, namely one containing CPUs with SSE and another based on GPUs. Both architectures have been invested in by the CEA - Grenoble who demanded efficient code.
- *Benchmark a Reference Implementation*
  For the CPU architecture, we have benchmarked the already existing algorithm in order to find performance issues. For the GPU architecture, we first implemented a basic version that we benchmarked before optimizing it according to the GPU specifics.
- *Identify Optimization Issues*
  The major optimization issue concerns data locality. Indeed, playing with the placement of data in the registers and in the cache can yield very different performance results. In the CPU case, our contribution is the definition of locality patterns to be used for memory placement, as well as for computation. In the GPU case, we contribute on the efficent usage of the cache.
- *Generate and Benchmark Different Optimizations*
  For each target architecture, we have explored different optimization strategies (in relation with the issues identifies in the previous point). For each strategy we have benchmarked and identified the best performing one.
- *Autotuning*
  For each optimization strategy we have analyzed the possibility to generalize its use and apply it in a different context.

The remainder of the paper is organized as follows. Setion II presents our use case, BigDFT, and details 3D convolutions. Section III explains how our optimization work is different from related projects. Section IV and Section V tackle optimizations on CPUs with vector units and on GPUs respectively. Finally, Section VI concludes and outlines our future works.

II. USE CASE: BIGDFT AND 3D CONVOLUTIONS

In this section we focus on the algorithm for 3D convolutions after presenting the general context of the BigDFT application.

A. The BigDFT Project

BigDFT [2] started as an EU FP6-STREP-NEST project with four partner laboratories: L_Sim - CEA Grenoble, Basel University - Switzerland, Louvain-la-Neuve University - Belgium and Kiel University - Germany. The goal of BigDFT is to develop a novel approach for electronic structure simulation

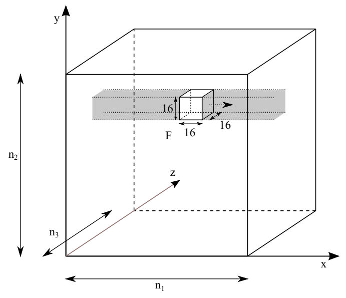

Fig. 1. 3D Convolution

based on the Daubechies wavelets formalism [1][3]. The code is HPC oriented i.e. it uses MPI, OpenMP and GPU technologies. So far, BigDFT is the sole electronic structure code based of systematic basis sets which can use hybrid supercomputers.

In 2009, L. Genovese has received the French Bull-Fourier prize for having implemented and coordinated the realization of the hybrid version of BigDFT [4]. Thanks to its use of the wavelets formalism, as well as to the HPC structuring of the code, BigDFT is characterized by its high precision, efficiency and flexibility. Currently, it has been chosen as an official benchmark for the EU MontBlanc project [5].

B. Algorithmic Overview

The usage of the wavelets formalism allows BigDFT to define nanosimulations in terms of 3D convolution operations. These operations consist in applying three-dimensional (3D) cubic filters to three-dimensional cubic zones (cf. Figure 1).

The wavelet transform consists in applying the cubic filter in all three dimensions. The transform takes a three-dimensional array *in* of dimension $n_1 x n_2 x n_3$ as an input and is transformed into the output array Ψ_r given by (1). The values n_1, n_2 and n_3 give the dimensions of the BigDFT simulation domain. To calculate the output values, the transform uses the ω_k values composing the so called *magic filter*. The computation is done using an extension from $-L$ to U. The filter dimension $U + L + 1$ is equal to the order l of the Daubechie wavelet family. In our use case, l has been chosen by the physicists to be 16. When the indexes are outside bounds, the data is used in a circular manner by using the index modulo the size of the data line[1].

$$\Psi_r(i_1, i_2, i_3) = \sum_{j_1,j_2,j_3=-L}^{U} \omega_{j_1}\omega_{j_2}\omega_{j_3} in(i_1 + j_1, i_2 + j_2, i_3 + j_3) \ . \tag{1}$$

A direct implementation of this algorithm would contain six nested loops. The three outer loops scan through all elements of the input array. The filter application is shown for the computation of element $\Psi_r(i_1, i_2, i_3)$ (cf. Listing 1). It will incur $n_1 n_2 n_3 l^3$ reads, multiplications and additions, and $n_1 n_2 n_3$ writes.

[1]This is called periodic boundary conditions.

```
const int L = 7;
const int U = 8;
double W[L+U+1] = {W0, W1, ... , W15};
double temp = 0;
for(j1 = -L; j1 <= U; j1++) {
 for( j2 = -L; j2 <= U; j2++) {
  for( j3 = -L; j3 <= U; j3++) {
   temp += W[j1+L]*W[j2+L]*W[j3+L]*in[i1+j1][i2+j2][i3+j3];
  }
 }
}
psi[i1][i2][i3] = temp;
```

Listing 1. Basic Implementation of the BigDFT Magic Filter Operation

What is interesting about this 3D convolution is that it can be separated in three independent 1D convolutions (2).

$$\begin{array}{ll} 1) & f_1(i_1, i_2, i_3) = \sum_{j=-L}^{U} \omega_j s(i_1 + j, i_2, i_3) \\ 2) & f_2(i_1, i_2, i_3) = \sum_{j=-L}^{U} \omega_j f_1(i_1, i_2 + j, i_3) \\ 3) & \Psi_r(i_1, i_2, i_3) = \sum_{j=-L}^{U} \omega_j f_2(i_1, i_2, i_3 + j) \end{array} \tag{2}$$

This algorithm's implementation will generate only $3n_1n_2n_3l$ reads, multiplications and additions, and $3n_1n_2n_3$ writes. However, if we consider data to be allocated in column-major order, the first loop accesses elements with a stride of 1, whereas the second and third one accesses elements with a stride of n_1 and n_1n_2 respectively. In other terms, the first loop accesses elements sequentially, the second loop needs to access elements at distance of n_1 elements and the third loop accesses n_1n_2-distant elements. On most architectures, such memory jumps prove to be harmful for the execution performance.

The most convenient way to compute a three dimensional convolution of this kind is by combining 1D convolutions and transpositions [6]. The computation is given in (3).

$$\begin{array}{ll} 1) & F_1(i_2, i_3, i_1) = \sum_{j=-L}^{U} \omega_j s(i_1 + j, i_2, i_3) \\ 2) & F_2(i_3, i_1, i_2) = \sum_{j=-L}^{U} \omega_j F_1(i_2 + j, i_3, i_1) \\ 3) & \Psi_r(i_1, i_2, i_3) = \sum_{j=-L}^{U} \omega_j F_2(i_3 + j, i_1, i_2) \end{array} \tag{3}$$

In this case, only writes have a stride different than 1 (respectively n_2n_3, n_3n_1 and n_1n_2). It can be noted that by processing the convolution along the third dimension first, the reads would have a stride while the writes would be sequential.

III. Related Work

3D convolutions are optimized using the Discrete Fourier Transform or using 1D convolutions and transpositions, as presented in the previous section.

- *Discrete Fourier Transform (DFT)*
  In this approach, the 3D convolution is calculated by applying the Fourier Transform to each dimension [7]. It has been, for example, used in the work of Akira Nukada

et al. [8] for a software for the BlueGene machine. The problem with this solution is that the complexity of the algorithm is $C \ln(n)$ per element where C depends on the chosen radixes. For powers of 2, C is commonly accepted as being $5/\ln(2)$. So considering a 128 line length, the cost of the 1D convolution is about 50 FLOP per element. In our case, with the size of the filter we need to consider, applying the DFT approach would be too expensive.

- *1D Convolutions and Transpositions*
  Another approach for optimizing 3D convolutions is to consider is to 1D convolutions and transpositions. The possibilities are to consider 1D+2D or 1D+1D+1D. For the considered size of the filter (i.e. 16x16x16), methods working on 2D are not applicable as the data size is too big to be interesting from the performance point of view. If there are many references considering 2D convolution optimization using SSE units or GPUs [9][10], the projects focusing on 3D convolutions are rare. For works considering 3D convolutions with vectorization, we can cite Intel [11] but the filter is very small (3x3x3) and the data limited to 16bit data. Gaussian 3D filtering has also been ported to SSE [12], but in this case the filters are symmetric and the authors limit the use of buffers. As for 3D convolutions on GPU, [13] use a 2D+1D method but for very small filter size.
- *Autotuning*
  Auto-tuning is a difficult problem and as far as we know, nobody has tried to auto-tune the type of BigDFT convolutions . However, there are several libraries using auto-tuning for different problems. For linear algebra ATLAS [14] is a well known reference. For fast Fourier transforms FFTW [15] and SPIRAL [16] are the most well-known. However, these libraries are limited to CPU-based platforms.

IV. Optimizing 3D Convolutions by Vectorization

In this section we analyze the performances of several versions of the 3D convolution algorithm using various levels of optimization. Only the separated convolution in double precision is considered.

A. Preliminary Performance Evaluation

The test platform is a Lenovo D20 workstation using 1 Intel Xeon X5550 CPU (featuring 4 cores) and 8 GiB of RAM. Hyper-threading and turbo boost are deactivated and performance governor is set to *performance*, so the processor frequency is set to 2.67GHz. As the Nehalem architecture is capable of providing two multiplications and two additions per cycle in double precision, each core has a peak performance of 10.6 GFlop/s. The operating system is Ubuntu 10.10 using a backported 2.6.36 linux kernel. The compilers used are Intel Compiler Suite version 11.1 and Gnu Compiler Collection version 4.4.5. With the latter, the optimization option used is -O2, as -O3 proved harmful to performances. Performance counters are obtained using PAPI [17].

We have studied several versions of the 3D convolution algorithm and the corresponding results are given in Table I. The numbers indicate the mean for 10 runs for convolving an array of size 128x126x130 double precision numbers (16 Mio) The versions include `simple` and `simple t` which correspond to the straightforward implementations of convolutions without and with transposition. `unrolled` and `unrolled t` are two versions in which the nested loops for the `simple` and the `simple t` algorithms are unrolled with a degree of 8. Loop optimization techniques, such as loop unrolling, are discussed in the work of Wolf and Lam [18].

The counters we have considered include the total number of computation cycles (*TOT CYC*) and the number of executed instructions (*TOT INS*). We also present several cache-related counters, namely the ones reflecting the number of data cache accesses (*DCA*) and of data cache misses (*DCM*) for cache levels 1 and 2.

Gflop/s are computed using the formula $3 * 32 * n_{elem}/t$. 3 comes from the number of dimensions (3D convolutions). 32 comes from the length of the filter (16) and the number of operations per element (one addition and one multiplication). t is the computation time derived from the *TOT CYC* counter.

The first thing to note is that unrolling greatly reduces the number of cycles needed to compute the 3D convolution, by a factor 6.25 for non transposed algorithm and 8.17 for the transposed one. This difference can be mainly accounted for by the number of instructions needed to compute the 3D convolution that is reduced by a factor of 6 thanks to the loop unrolling.

The second thing that can be noted is that transposed versions of the algorithm present a better instruction throughput than their counterparts. This is especially true for the unrolled versions, where the transposed one present 28% more throughput than its counterpart. This can be explained by comparing the data cache miss results: non transposed versions incur twice as many cache miss in L1 than transposed versions. For L2 this ratio is of 3.5 for simple convolutions and 1.6 for unrolled convolutions.

Finally, even if the unrolled algorithms are much more efficient than the simple ones, their performance are still unsatisfactory. Indeed they achieve respectively 21% and 28% of the peak performances of the core they use.

The total number number of floating point operations needed to compute the whole 3D convolution is: $128 * 126 * 130 * 32 * 3 = 201M$. If we add to this result the number of data accesses (*L1 DCA*) in the case of the `unrolled t` version we obtain: $201M + 137M = 338M$ instructions. If we consider the ratio between these values, it indicates that there has been poor register reuse as memory operations take about $2/3$ of the arithmetic operations. If we compare this value to the one given in the table (TOT INS $386M$), this result leaves about $50M$ of operations that are not accounted for. These can be loop overheads, or copies in registers.

Vectorizing the code may allow a reduction by a factor of two of the number of arithmetic operations. It may also reduce the number of memory accesses by even bigger factor if

version	TOT CYC	TOT INS	INS/CYC	L1 DCA	L1 DCM	L2 DCM	GFlop/s
simple	1500M	2300M	1.53	120M	13.4M	4.68M	0.357
simple t	1480M	2470M	1.67	211M	7.54M	1.34M	0.361
unrolled	240M	401M	1.67	149M	4.72M	2.06M	2.23
unrolled t	181M	386M	2.13	137M	2.48M	1.30M	2.96
sse t	82.2M	175M	2.12	45.4M	3.26M	1.33M	6.52

TABLE I
COUNTER RESULTS FOR DIFFERENT VERSIONS OF 3D CONVOLUTIONS.

registers are reused efficiently. We focus on vector instructions used in the x86 architecture, namely the SSE instruction set [19] and its extensions SSE2, SSE3 and SSE4.

B. 3D Convolutions Vectorization Approach

In order to vectorize efficiently 3D convolutions, we have taken into account memory alignment, register pressure, register reuse and memory access patterns. We have worked by separating data (filters and input array) in 2-element vectors. As the algorithm needs only 32 arithmetic operations (1 multiplication and 1 addition for 16 elements) per step and per resulting element, reordering data in memory will prove too costly. Indeed, in our case the complexity is of $O(1)$ per element. Reordering methods are interesting on GEMM type operations with complexity of $O(n)$ per element. n being the length of the rows of the first matrix, the cost is easily masked when n is big enough.

We have adopted a solution with two filters as illustrated in Figure 2.

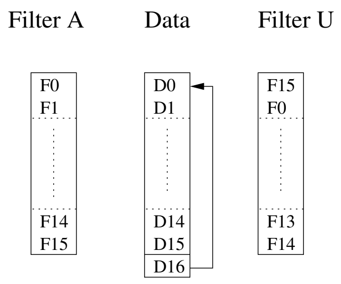

Fig. 2. Two-filter computation where data and filters are loaded into registers.

The first filter FA consists of values $(F_0, F_1, ..., F_{15})$. The second filter FU contains $(F_{15}, F_0, ..., F_{14})$. Both filters are aligned on 16 byte boundaries.

In our first approach, we assume that data is also aligned on 16 byte boundaries and thus allow the reuse of data registers as explained in the following.

Let us consider the computation of two successive data, R_0 and R_1, defined by (4) and (5). In the equation, D_0 is the one also shown in Figure 2 and is aligned on a 16 byte boundary. $\otimes$ is the vectorized multiplication operation which obtains a two-element result by respectively multiplying the first and the second elements of the two-element vector operands.

It can be noted that seven of the eight two-element data vectors can be reused directly. The first one needs to be updated with D_{16}. If we compute several values in parallel, filter vectors can also be reused.

$$R_0 = \sum_{i=0}^{15} D_i F_i = V_{0_0} + V_{0_1}$$
$$\text{where} \tag{4}$$
$$\vec{V_0} = \sum_{i=0}^{7} [D_{2i}, D_{2i+1}] \otimes [FA_{2i}, FA_{2i+1}]$$

$$R_1 = \sum_{i=0}^{15} D_{i+1} F_i = V_{1_0} + V_{1_1}$$
$$\text{where} \tag{5}$$
$$\begin{aligned}\vec{V_1} =& [D_{16}, D_1] \otimes [FU_0, FU_1] \\ &+ \sum_{i=1}^{7} [D_{2i}, D_{2i+1}] \otimes [FU_{2i}, FU_{2i+1}]\end{aligned}$$

The register reuse is limited by the number of available registers. Indeed, we need one register per result, half of the registers for data and two registers for the filters. Given this repartition, computing 8 values in parallel consumes 14 registers (8 results, 2 filters, 4 data). However, SSE provides only 16 registers! As a consequence, computing more than 8 values in parallel might create register spilling. On the contrary, using less registers should diminish register reuse and increase the ratio between memory operations and arithmetic operations.

C. Performance Study of Convolution Kernels

In this section we study the performance obtained through vectorization of different versions (we call kernels) of the convolution computation. First we consider variants without transposition and then variants with transposition (cf. Section II-B)

1) Kernels without Transposition: In order to experiment with the presented strategy, we have generated several patterns for computing data values (cf. Figure 3). The different patterns compute 2 to 12 values in one pass. Thus, if a transposition is to be added, several consecutive values can be written in memory. The vectorization is done using intrinsic operations, the compiler being responsible for register allocation and management.

The benchmarks use arrays of 32 columns of 5040 elements. The size is a multiple of the pattern dimension and fits in the L3 cache of the processor. The column size is much longer than what is used in BigDFT but our aim is to measure the asymptotic performances.

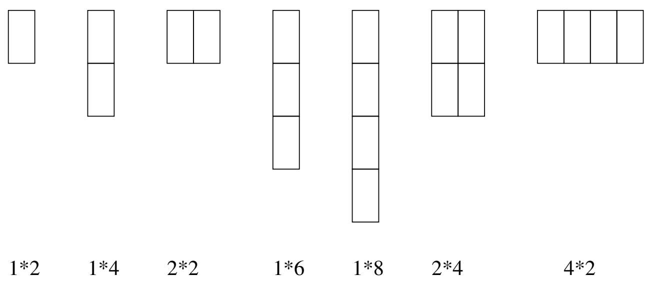
Fig. 3. Example of patterns used during experiments.

Element computed	pattern column*line	GFlop/s
2	1*2	7.45
4	1*4	8.45
4	2*2	8.03
6	1*6	8.57
8	1*8	8.49
8	2*4	8.43
8	4*2	8.49
10	1*10	8.43
12	1*12	7.77

TABLE II
PERFORMANCE OF CONVOLUTION PATTERNS WITHOUT TRANSPOSITION

The mean results of 20 runs are presented in table II. What we see is that even small patterns present improved performances compared to the non vectorized versions (cf. Table I). However, these patterns are not 3D and cannot be used "as is" for a 3D convolution. To be able to use the patterns, we should also use a transposition.

Performances are maximal and stable for patterns of 6 to 10 elements. Analyzing the generated assembly code reveals that the pattern of 12 elements suffers from register spilling. The other patterns, with element from 2 to 10 do not generate register spilling i.e. the compiler generates computations that fit in the 16 available registers.

2) Kernels with Transposition: For a transposed version of a convolution kernel to be efficient, it has to write several elements successively in memory. Having this in mind, we have chosen two candidate kernels: the 2*4 pattern and the 4*2 pattern.

To calculate the transposition, we have simply used the available SSE operation for manipulating vectors. The operation computes this sum for 2 vectors and stores the results in a third one. Using column-adjacent values creates a vector that can be directly stored.

The results are presented in table IV-C2. Computing more columns simultaneously is faster and the 4*2 pattern presents better performances (7.14 GFlops/s versus 6.32 GFlops/s for the 2*4 pattern).

Unfortunately, the version with transposition is less performant than the version without transposition. Attempts to use buffers to reduce the transposition cost proved unsuccessful. The 4*2 pattern was elected to create the final convolution.

Element computed	pattern column*line	GFlop/s
8	2*4	6.32
8	4*2	7.14

TABLE III
PERFORMANCE OF CONVOLUTION PATTERNS WITH TRANSPOSITION

D. 3D Vectorized Convolution

In order to build the magic filter in periodic boundary conditions, the pattern had to be adapted to take into account boundary conditions. In periodic conditions, column length is always a multiple of 2 and column number a multiple of 4, so alignment is not a concern. The first 4 and last 4 instantiations of the pattern have to be modified to load data from the end and the start of the column respectively. The 1D algorithm is then used 3 times to create the 3D version.

The performances of this implementation are compared to the previous ones in Table I. The SSE version with transposition reaches 6.52 GFlop/s which is more than twice the performances of the unrolled and transposed version. We can see that the generated code is more efficient as the number of instructions has been reduced by the same factor. In fact, the instruction throughput is the same, and while L1 data cache miss augmented slightly, behavior in other cache levels stay similar.

We believe that the generic case will benefit from the same optimization approach. Indeed, the technique already addresses one aligned and one unaligned element. It would just have to be adapted with an offset and modified filters.

V. OPTIMIZING 3D CONVOLUTIONS ON GPU

GPU computing [20] in physics and chemistry is a strong trend. Indeed, GPUs offer impressive raw performances at the cost of refactoring part of the code. Indeed, some algorithms are not adapted to the GPU architecture and need a case by case treatment.

Most GPU programs are nowadays based on the NVIDIA's proprietary solution: CUDA [21]. Nonetheless, there is a significant effort in defining standard: OpenCL [22]. OpenCL drivers are provided by manufacturers and almost every device nowadays support OpenCL. Portability stays a major issue.

In our convolution optimization work, we had to implement from scratch 3D convolutions for both NVIDIA and AMD architectures. In the following we start by presenting the principles of the OpenCL model. We then discuss the algorithmic details for implementing convolutions on a GPU architecture before pointing out AMD specifics. We finish with performance evaluation of our impementations.

A. OpenCL GPU Architecture

OpenCL is an open standard defined by the *Khronos Group* that aims at cross-platform parallel computing. Among the several types of devices defined by the standard, the GPU one corresponds to modern Graphical Processing Units. These accelerators present a high performance-over-cost, as well as performance-over-power consumption ratio. GPUs have become increasingly available in computing clusters and are targets of choice for numerical experimenters.

The GPU device in OpenCL is made of several address spaces and a set of multiprocessors. OpenCL is aimed at data parallel tasks and describes the computation in terms of workgroups composed of work-items. When executing an OpenCL function (also called *kernel*), work-items execute the same code.

The difference between work-items from different work-groups is the visibility of address spaces. The four address spaces are *global*, *local*, *private* and *constant*. The *global* address space is usually larger and with a higher latency than the *local* address space which is private to a workgroup. The first corresponds to the on-board RAM while the later corresponds to a user managed cache. The *private* memory corresponds to the registers of a work-item.

B. Convolution Implementation on GPU

The first thing to optimize when using a GPU is memory access. Reading and writing to a *global* memory should be coalesced. The easiest way to achieve this goal while transposing is to use a padded square buffer in *local* memory [23]. Transposition can be done while reading input data or writing output data. In BigDFT, data is read coalesced and,as the GPU processes the last dimension first, is stored transposed in the buffer. As the filter is 16 elements long, each column of a 16*16 buffer needs 15 more elements to be convoluted. In order for threads to execute the same code, each work-item loads 2 elements in a 32*16 buffer padded to 33*16. Work-items then compute a filtered value in *private* memory. The calculated value is finally stored in the result buffer in *global* memory. Figure 4 presents an example of work items assignation for a 4*4 block processing a filter of length 5. Buffer is in column major order. Thread 0,i 1,i ... k,i are processed simultaneously.

It has to be noted that this allocation allows memory accesses to be free from bank conflicts. This is true not only during the transposition, but also during the computation of the filtered values. Indeed, consecutive threads access consecutive elements in the buffer. As using *constant* memory for the filter values proved harmful to performances, the values were instead directly inserted into the code.

On more complex kernels found in BigDFT the order of the filtering operations also had an impact that could not be neglected. Indeed, once loaded into *local* memory, the order in which threads access the elements in the buffer has no real impact on performances (provided access do not produce bank conflicts). But trying to use identical filter coefficient in a grouped manner proved beneficial.

From the GPU parallelism point of view, there is a set of N independent convolutions to be computed. Each of the lines of n elements to be transformed is split in chunks of size N_e. Each multiprocessor of the graphic card computes a group of N_ℓ different chunks and parallelizes the calculation on its computing units. After the calculation of the convolution values, these $N_e N_\ell$ elements are copied in the corresponding part of the output array, which is transposed with relation to the input.

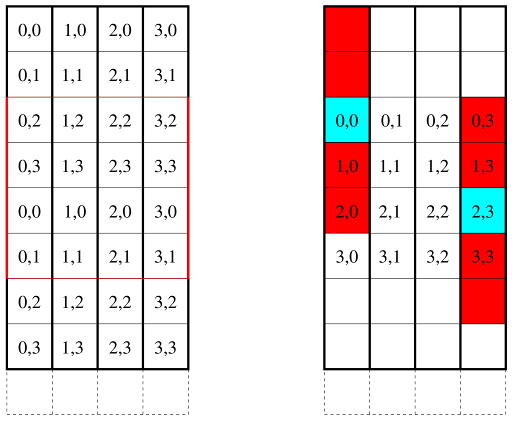

Fig. 4. Thread allocation: example of a 4*4 block and filter of length 5.

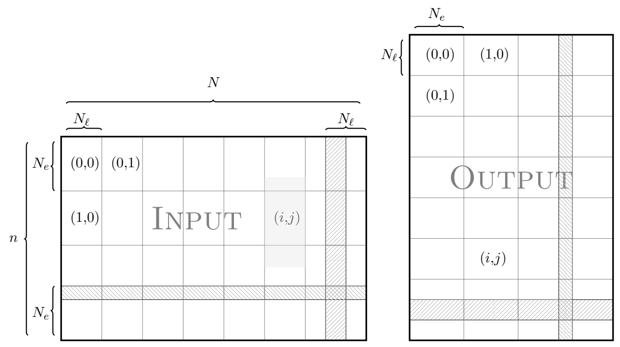

Fig. 5. Data Distribution for 1D Convolution+Transposition on the GPU. Input data (left panel) is ordered along the N-axis, while output data (right panel) is ordered in n-axis direction (cf. Section 3). When executing GPU convolution kernel, each block of the execution grid (i,j) is associated to a set of N_ℓ (N-axis) times N_e (n-axis) elements. The filled patterns in the figure indicate the overlap region, i.e. data which are associated to more than one block. Behind the (i,j) label, in light gray, is indicated the portion of data which should be copied to the shared memory to treat the data in the block, which contains also the buffers needed for computing the convolution.

The size of the data fed to each block is identical and chosen to prevent block dependencies. When N and n are not multiples of N_ℓ and N_e, some data treated by different blocks may overlap. This fact has no double-counting effect since the overlap is reproduced also in the output array. Figure 5 shows the data distribution on the grid of blocks during the transposition.

C. Convolution Specifics on the AMD Architecture

AMD offers OpenCL with double precision support for it's Radeon HD line of GPU. There are 2 main differences between NVIDIA and AMD architecture regarding OpenCL programming. First AMD architecture is vector-based whereas NVIDIA architecture is scalar. Secondly AMD global memory access are split between channel and banks. Accessing data between workgroups with large power of 2 strides can prove really harmful on AMD architecture.

The most recent AMD architecture (Radeon HD 69xx) is based on vector operations of length 2 in double precision. To account for this, at runtime, when an AMD GPU is detected the filter multiply is vectorized. Large powers of 2 problem size are not the rule in BigDFT but rather the exception. So the memory channel/bank problems have not been addressed in the current version of the kernels. Nonetheless if such a stride was detected at runtime it would be possible to use a staggered offset version of the kernel.

D. Performance Evaluation of 3D Convolution Kernels

GPU performance of the kernels are evaluated using 2 test computers. Each runs an Ubuntu 11.04 linux based system using an unmodified 2.6.38-11 kernel. Compiler used is Intel 11.1 Compiler Suite. Processor is a X5550 Xeon with 4 cores and 8GiB of RAM. The setups only differ in their GPU and graphic drivers, the first one uses an NVIDIA TESLA C2070 using 270.40 drivers, the second one a RADEON HD6970 using 11.6 drivers. The 3D convolution results directly of 3 applications of the 1D convolution. Problem dimension are 124 x 17160 for 1D problems and 124 x 132 x 130 for 3D problems.

Two versions of the kernels have been tested one standard, and one where a work item computes 2 elements of the resulting array. Performance of the inverse operation is also considered, as well as the performance of the 3D convolutions. The 3D convolution results directly of 3 applications of the 1D convolution.

Results are presented in table IV. There is a slight performance advantage for AMD (10%) for these algorithm and those problem size. Using a 128 x 128 x 128 matrix halves performances on AMD while NVIDIA's performances are unaffected. Tis is due to the afore mentioned channel/bank problem on AMD hardware. This problem could be solved using staggered offsets. This approach does not need to embed the array in a larger one contrary to padding that would then need to be carried along the whole pipeline.

Similarly on NVIDIA hardware performance are decreasing for multiple of 7, but those dimension are forbidden in BigDFT for FFT performance reasons. Despite those specific problems performances are remarkably uniform regarding problem size, and variations in performances is mainly accountable on the overlapping part (and thus the waste of computing power) rather than on memory conflicts.

Most of the performance was gained on GPUs by optimizing memory accesses and by hiding computations between memory operations. Using constant memory for filter values proved less interesting than directly inserting the values inside the code using macro definitions. Computations are not completely hidden because vectorizing the computation loop on AMD hardware yields an increase of 10% in performance.

E. Performance Evaluation for BigDFT

In this section we present the impact of the OpenCL implementation and MPI configuration on the completion time of BigDFT. The system studied is a graphene sheet modeled with a 4 carbon atom supercell and 52 k-points. Only the core loop is timed, as initialization and finalization time are roughly equivalent between NVIDIA and ATI. Results are presented in Table V. It can be noted that due to the runtime OpenCL compilation the same binary is used on both machines.

The first important thing to note is that BigDFT is not scaling very well in MPI, due to bandwidth limitation. In our case GPUs bring not only their computing power, but also their impressive memory bandwidth. The second interesting trend is that NVIDIA GPU offer better (though comparable) performances than AMD on this problem. The situation is reversed compared to the unit tests presented in the previous section. The problem studied contains a dimension that is a power of 2: 32. So performances are seriously hindered on AMD's hardware in this case. As BigDFT is synchronous between MPI processes the hybrid case performances is limited by AMD node. Speedup is of 1.8 comparing 4 MPI + AMD and (4 MPI + NV) + (4 MPI + AMD). In this case, GPUs increase by a factor of about 10 the performance of the node considered.

For both architectures (GPU and CPU with small vector engines) the only way to increase the computation over memory access ratio and alleviate the memory bandwidth problem would be to merge different convolutions. This would yield longer filters but reduce the number of transposition needed. Strategies presented here can be adapted to fit those conditions, especially since later GPU OpenCL devices have more *local memory*. For the CPU our algorithm is oblivious to the filter length.

Architecture	Algorithm	Dim	Performances (GFlop/s)
NVIDIA	magic filter	1D	93
		3D	91
	magic filter block	1D	108
		3D	105
	magic filter inverse	1D	92
		3D	92
AMD	magic filter	1D	105
		3D	101
	magic filter block	1D	121
		3D	118
	magic filter inverse	1D	105
		3D	100

TABLE IV
PERFORMANCE RESULTS FOR DIFFERENT VERSIONS OF THE CONVOLUTIONS ON DIFFERENT GPU ARCHITECTURES.

MPI+NVIDIA/AMD	Execution Time (s)	Speedup
1 MPI	6020	1
4 MPI	1660	3.6
1 MPI + NVIDIA	300	20
4 MPI + NVIDIA	160	38
1 MPI + AMD	347	17
4 MPI + AMD	197	30
(4 MPI + NV) + (4 MPI + AMD)	109	55

TABLE V
PERFORMANCE RESULTS FOR SEVERAL CONFIGURATIONS OF BIGDFT, USING MPI + GPUS

VI. Towards Auto-Tuning of Convolutions: Conclusions and Future Work

In this paper we focused on optimizing the basic and most frequent operations, namely 3D convolutions, in the BigDFT nanosimulation application. This application simulates the properties of future electronic materials and because of its scalability properties has been elected official benchmark for the HPC EU project MontBlanc .[5]

We have considered optimizing 3D convolutions for two types of architectures: CPU architectures using SSE units, as well as GPU-based architectures. We have optimized BigDFT on three platforms, featuring respectively CPUs with SSE units, AMD GPUs and NVIDIA GPUs. For the first, we have explored vectorization techniques, while for the second we used intelligent cache utilization. For both types of work, the major principle has been *data locality*. In the case of CPUs, we have defined access patterns that define different locality groups used to optimize memory accesses, as well as computation. For GPUs, we have defined blocks of data for aggressive cache usage. All techniques are reusable in other contexts different from convolutions.

Concerning CPUs, we have used only 1 core and have not been limited by the CPU bandwidth. However, it would be interesting to consider using multiple cores, even if the performance gain would be limited. It could be beneficial to use cores for compute-intensive tasks or for network communication tasks [24]. The transposition performance of our algorithm are 75% of those of a memory copy, so going back to a straight algorithm that would require each dimension to be treated differently is not interesting.

Concerning GPU architectures, both have presented similar behavior. A major problem has been the bandwidth usage so we should evaluate the trade-off between the block size and the consumed bandwidth. Anyway, if we consider performance on our target hardware, memory copies limit us to about 170 GFlops. Any further improvement is limited to a 70% increase in performance, and that would require efficient algorithms without overlaps and perfect recovering of computation time per memory accesses.

A pending challenge, raised by the experiments on vectorized convolutions, is the remaining parameter space to explore in order to find the best convolution version. The different configurations are meaningful not only from the computational point of view but also from a physicist point of view. Indeed, filters of different sizes have different properties in terms of convergence and accuracy. For now the filter size is fixed, but we could imagine changing this parameter in order to improve convergence speed and balance this gain with the computational performance impact.

Our future work will be to build a convolution generator that is able to produce a library of parametrized convolutions to be used for BigDFT's performance evaluation. This tool would allow a broader coverage of SSE versions as those are tedious to produce *by hand*, error-prone and difficult to maintain. The generator could also produce OpenCL kernels and allow better performance studies. These performance evaluations, studies and experimentations are a necessary step in understanding the behavior of HPC applications and in being able, one day, to produce a prediction performance model depending on the target architecture.

References

[1] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, A. Bergman *et al.*, "Daubechies wavelets as a basis set for density functional pseudopotential calculations," *The Journal of chemical physics*, vol. 129, p. 014109, 2008.

[2] "The bigdft scientific application," 2012. [Online]. Available: http://inac.cea.fr/LSim/BigDFT/

[3] H. Nussbaumer, "Fast fourier transform and convolution algorithms," *Berlin and New York, Springer-Verlag.*, vol. 2, 1982.

[4] L. Genovese, M. Ospici, T. Deutsch, J. Méhaut, A. Neelov, and S. Goedecker, "Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures," *The Journal of chemical physics*, vol. 131, p. 034103, 2009.

[5] "The mont-blanc project," 2012, http://www.montblanc-project.eu.

[6] S. Goedecker, "Rotating a three-dimensional array in an optimal position for vector processing: case study for a three-dimensional fast fourier transform," *Computer Physics Communications*, vol. 76, no. 3, pp. 294 – 300, 1993.

[7] S. Goedecker, M. Boulet, and T. Deutsch, "An efficient 3-dim fft for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes," *Computer Physics Communications*, vol. 154, no. 2, pp. 105–110, 2003.

[8] A. Nukada, Y. Hourai, A. Nishida, and Y. Akiyama, "High performance 3d convolution for protein docking on ibm blue gene," in *Parallel and Distributed Processing and Applications*, ser. Lecture Notes in Computer Science, I. Stojmenovic, R. Thulasiram, L. Yang, W. Jia, M. Guo, and R. de Mello, Eds. Springer Berlin / Heidelberg, 2007, vol. 4742, pp. 958–969.

[9] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU," in *Information Visualization, 2006. IV 2006. Tenth International Conference on*, july 2006, pp. 609 –614.

[10] V. Podlozhnyuk, "Image convolution with cuda," *NVIDIA Corporation white paper, June*, vol. 2097, no. 3, 2007.

[11] Z. Danovich, "16bit 3D Convolution: SSE4+OpenMP implementation on Penryn CPU." [Online]. Available: http://software.intel.com/en-us/articles/16bit-3d-convolution-sse4openmp-implementation-on-penryn-cpu/

[12] A. Vaško and M. Šrámek, "Optimizing Gaussian Filtering of Volumetric Data Using SSE," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 1, pp. 100–116, 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1620

[13] M. Hopf and T. Ertl, "Accelerating 3d convolution using graphics hardware (case study)," in *Proceedings of the conference on Visualization '99: celebrating ten years*, ser. VIS '99. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 471–474. [Online]. Available: http://dl.acm.org/citation.cfm?id=319351.319457

[14] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999, Electronic Proceedings.

[15] M. Frigo, "A fast fourier transform compiler," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, ser. PLDI '99. New York, NY, USA: ACM, 1999, pp. 169–180.

[16] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "Spl: A language and compiler for dsp algorithms," in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 298–308.

[17] P. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proc. Dept. of Defense HPCMP Users Group Conference*. Citeseer, 1999, pp. 7–10.

[18] M. Wolf and M. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 452–471, 1991.

[19] S. Thakkur and T. Huff, "Internet Streaming SIMD Extensions," *Computer*, vol. 32, no. 12, pp. 26 –34, dec 1999.

[20] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879 –899, may 2008.

[21] NVIDIA, "NVIDIA Compute Unified Device Architecture," 2011. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[22] Khronos OpenCL consortium, "OpenCL: Open Computing Language," http://www.khronos.org/opencl/.

[23] C. NVIDIA, "Opencl best practices guide," 2010. [Online]. Available: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[24] S. R. Alam, G. Fourestey, B. Videau, L. Genovese, S. Goedecker, and N. Dugan, "Overlapping computations with communications and i/o explicitly using openmp based heterogeneous threading models," in *IWOMP*, ser. Lecture Notes in Computer Science, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds., vol. 7312. Springer, 2012, pp. 267–270.

LIG