Performance dissection of a Molecular Dynamics code across CUDA and GPU generations

Matthew Wezowicz, Trilce Estrada, Sandeep Patel, and Michela Taufer
University of Delaware
Newark, Delaware, 19716
Emails: mwezz, estrada, sapatel, taufer@udel.edu

***Abstract*—The first release of CUDA was in 2007. Since then, we have experienced frequent new releases. CUDA reached its maximum performance with CUDA 4.0. Since its release, NVIDIA has started a re-design of the CUDA framework driven by software engineering principles, i.e., the search for a general, multi-layer framework whose compiler back-end is unified with OpenCL. At the same time, the GPU architecture has been moving from Fermi to Kepler by including faster memory speed larger numbers of stream processors, and redesigned pipelines. The two directions have orthogonal results in terms of performance. The software generality has resulted in the slow down of codes that were heavily optimized for older generations of CUDA, while the hardware improvements have resulted in steadily increasing performance. The performance dissection presented in this paper identifies sweet spots and trade-offs between software generality and hardware improvements for a diverse set of kernels in an open-source molecular dynamics code.**

I. Introduction

Programming tools and technologies have been continuously and steadily evolving in the GPU community. The CUDA programming language and NVIDIA technology have been playing a leading role from the beginning. A first version of CUDA was released in 2007 and reached its maximum performance with CUDA 4.0. Recently, NVIDIA has started a re-design of the CUDA framework driven by software engineering perspectives characterized by the search for a general, multi-layer compilation infrastructure which compiler back-end is unified with OpenCL. This can ultimately have a significant impact on both maintenance costs and cross-platform portability. The software engineering community applauded this direction. At the same time, the hardware technology has also evolved, i.e., from the Tesla to the Kepler architectures, driven by the search for higher performance, lower power consumption, more efficient instruction pipelines, and more accurate results.

Our work presented in this paper indicates that the new direction of CUDA comes at some performance loss for large scale simulations such as MD simulations. Supported by the rigorous performance dissection of an MD code and its optimized versions, this paper aims to raise the question: "Is the HPC community, that has been benefiting the most from the CUDA programming language and the GPU technology, ready to keep up with the challenges associated to the new CUDA and GPU directions?". We cannot forget that code developers have in the past heavily optimized their codes for older generations of CUDA such as CUDA 4.0 and GPUs such as the C2050. When running these codes on platforms with updated CUDA versions and, very soon, updated GPU architectures, scientists may have to cope with the associated performance loss. In this paper, rather than denying the problem and demonizing the emerging directions, we want to study the problem and help think of ways to reconcile performance with portability and maintainability.

To this end, we look at the performance from two different perspectives (i.e., the scientist and the computer scientist perspectives) and we dissect a diverse set of kernels from an MD code called FEN ZI (Yun Dong de FEN ZI = Moving MOLECULES) [1], [2] at three different levels for different code implementations, input data sizes, CUDA variants, and GPU architectures. First, for the different scenarios resulting for the possible combinations of codes, input data, CUDAs and GPUs, we look at the amount of science each MD simulation can perform in terms of nanosecond per day (ns/day). Second, we zoom into their executions, identify critical kernels, and present their performance from the point of view of their wall-clock times. Third, we dig into the hardware and look at the same critical kernels and any unusual behavior from their hardware resource point of view, e.g., registers, memory, I/O. The set of kernels in FEN ZI include diverse algorithmic components that can serve as a basic building blocks in other real applications. Our analysis is performed on both kernels that expand and contract their number of threads in the thread pool to accommodate larger or smaller inputs (i.e., number of molecular atoms) and kernels that expand and contract the thread load on a fixed-size thread pool to accommodate the larger or smaller inputs. We identify performance sweet spots and trade-offs that reconcile antagonistic software generality and hardware improvements. The paper contributions are: (1) to capture driving factors at both software and hardware levels that impact performance and (2) to translate this new knowledge into important lessons for the community of GPU users and code developers.

The rest of this paper is organized as follows: Section II lists relevant related work and outlines differences with our paper; Section III describes how CUDA has evolved in the

past five years to move towards a general framework; Section IV briefly compares the Fermi and Kepler architectures; Sections V presents our MD code, its kernels, and the scalable datasets; Section VI critically presents the three levels of our performances dissection; Section VII discusses the lessons learned; and Section VIII concludes the paper.

II. Related work

The performance comparison of codes across GPU platforms and between GPU and CPU platforms is an active topic in the high performance community. Stone and co-authors [3] provide an overview of benefits associated to the use of GPUs for MD simulations. Their work surveys the development of molecular modeling algorithms that leverage GPU computing. Contrary to Stone's and co-authors, we build a benchmark from multiple kernels of a single MD code and, rather than focusing on the algorithmic aspect of MD simulations, we meticulously cut the performance of our code at both hardware and software levels for different CUDA and GPU generations. In their article on moving towards a performance-portable solution for multi-platform GPU programming from CUDA to OpenCL, Dongarra and co-authors applied a similar performance approach as we do in this paper but for linear algebra applications. Rather than looking across GPU generations and variants of the same software framework, the authors looked across vendor's GPU for one single GPU generation [4]. Among the most relevant performance comparisons we can cite work of Lee and co-authors on optimization techniques for codes on CPU and GPU platforms and the correlation between performance and hardware features [5]. As with Lee and co-authors, we agree on the relevance of mapping hardware resources to performance. Contrary to them, we put our analysis in the context of the recent software directions that are driven by the need of maintenance and portability.

We look at how the hardware has evolved to the point that it compensates performance loss associated to the new software directions. Numerous web pages report about performance comparisons of two or more GPUs with well known benchmark suites on reduced datasets. In most cases the analysis does not include aspects such as the impact of software and hardware evolutions across diverse dataset sizes and kernels as it does in this paper. More in general, to the best of our knowledge, no work is currently available that looks at the evolution of the GPU programming languages and the GPU hardware in concert supported by a rigorous performance dissection.

Last but not least, our MD code (i.e., FEN ZI) that we use in this paper as our benchmark is only one of the several codes available for MD simulations. Among the most well-known and used we can cite ACMD [6], AMBER [7], NAMD [8], and GROMACS [9]. Some of these codes include the PME calculations on GPU, others split the execution of the MD steps between GPUs (for non-bonded, short-range electrostatic interactions) and CPUs (for bond interactions and PME calculations). Note that this paper's goal is not the comparison of these codes with FEN ZI (this was addressed elsewhere). These codes use very similar models and algorithms as FEN ZI to achieve the same scientific goals. Similarly to FEN ZI, all these codes were written and optimized for CUDA 4.0 on the Fermi GPU generation. Therefore it is expected that they are facing similar performance patterns as FEN ZI when using CUDA 5.0 and Kepler GPUs [10]. The lessons learned with FEN ZI in this paper are extensively applicable for these codes as well.

III. The CUDA generations

Figure 1 presents the key components of the multi-layer compilation framework in the four most recent CUDA releases (CUDA generations), i.e., CUDA 4.0, 4.1, 4.2, and 5.0. For the sake of completeness, we include the OpenCL multi-layer framework. The figure outlines significant similarities between CUDA 5.0 and OpenCL.

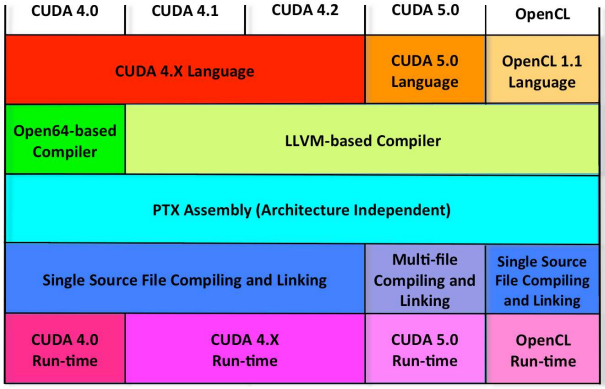


Figure 1. Multi-layer structures of CUDA and OpenCL frameworks.

A. CUDA language

The upper layer, the *CUDA language*, specifies the parallel section of the code in a version of C with parallel extensions. The main differences are between CUDA 4.x and CUDA 5.0, where CUDA 5.0 adds new features to the kernels. The most prominent addition is the new dynamic parallelism. Dynamic parallelism enables kernels to call other kernels from the device. Algorithms that required restructuring to eliminate recursion can now be written more easily and algorithms that needed global synchronization do not need the host to control it. OpenCL is currently closer to CUDA 4.x than CUDA 5.0 in terms of what can be written into the kernels.

B. Compiler layer

The *compiler layer* transforms CUDA kernels in .cu source files into CUDA Driver API function calls and C++ objects; it also transforms the first stage into PTX assembly and optimizes it. CUDA 4.0 has an Open64 based compiler back-end that has been modified by NVIDIA to generate PTX and optimize for GPU architecture (vectorized loads and stores, unlimited registers, no branch prediction) while CUDA 4.1, 4.2, and 5.0 have an LLVM based compiler back-end that does the same thing. NVIDIA's OpenCL implementation appears to

use the same LLVM based compiler for its run time compiling of kernels.

C. PTX assembly and kernel linking

The *PTX assembler* is the architecture independent assembly code. No significant change can be observed at this layer level. The single *source file compiling and linking layer* in CUDA 4.x was transformed into a multi-file compiling and linking in CUDA 5.0. This means that in CUDA 4.x the entire kernel needed to reside in a single source file, even if it used multiple functions. With CUDA 5.0, a kernel can be spread over multiple source files, each compiled separately and linked together to create the kernel. Now, it becomes very easy to reuse code, use third party closed-source functions in kernels, or allow an end-user to specify call backs that can be linked in. The latest OpenCL 1.2 specification allows for similar functionality but is not currently implemented by NVIDIA.

D. Runtime system

The *CUDA run-time system* supports the execution of CUDA programs on GPUs and provides math libraries. We observe three generations of run-time systems that are differentiated in terms of which CUDA version they support. The CUDA 4.0 run-time only supports programs compiled with the CUDA 4.0 compiler, the CUDA 4.x run-time supports programs compiled with CUDA 4.1 or 4.2, while the CUDA 5.0 run-time only supports programs compiled with CUDA 5.0. Using a different run-time causes programs to malfunction. OpenCL also has a run-time but it is part of the graphics driver and appears to be version independent as long as it is new enough to support the OpenCL specification being used.

IV. The GPU generations

We ran our tests on two different Tesla generations, i.e., Fermi and Kepler generations. For the Fermi generation, we consider both a consumer GPU card such as GeForce GTX480 and a computing GPU card such as Tesla C2050, running the 285.05.33 drivers. For the Kepler generation we consider the consumer GPU card GeForce GTX680. The detailed specifications of the three cards can be found in the vendor web page. We observed that across generations, key hardware features including memory speed and number of stream processors have been steadily growing. When comparing commercial GPUs such as the two GeForce cards, GTX480 has a clock speed of 700 MHz and a GDDR5 memory speed of 924 MHz. It also uses a 384-bit bus and features 480 stream processors, 60 texture address units, and 48 raster operations units. GTX680 features a GPU core clock speed of 1006 MHz and a 2048 MB of GDDR5 memory speed. It also uses a 256-bit bus and features 1536 stream processors, 128 texture address units, and 48 raster operations units. Tesla C2050 has a clock speed of 575 MHz and a GDDR3 memory speed of 1.5GHz. It also uses a 384-bit bus and features 448 stream processors.

When compared to the high end Tesla C2050/C2070, the gain in single precision speedup for GeForce cards (both GTX480 and GTX680) comes at the cost of key features in scientific simulations such as ECC memory for non-compromised accuracy and scalability. In terms of generations, both C2050 and GTX480 belong to the old Fermi architecture while the GTX680 card is based on the newer Kepler architecture including more stream processors and the redefinition of the instruction pipeline.

V. Benchmark and datasets

A. Benchmark: MD code

We use FEN ZI (Yun Dong de FEN ZI = Moving MOLECULES) as our benchmark for the analysis [1], [2]. FEN ZI enables GPU-based MD simulations in NVT, NVE, and NPT ensembles and energy minimization. With FEN ZI, MD simulations are entirely computed on a single GPU. The force field used is the CHARMM force field and long distance electrostatic interactions are computed with the Ewald summation method [11]. Both explicit or implicit solvent models are supported. Because of its advanced simulation features that have been tested with different relevant molecular systems, FEN ZI is a reliable tool for MD simulations on GPUs.

We use the FEN ZI code as our benchmark because it comprises a diverse set of kernels, each including diverse algorithmic components that can serve as basic building blocks in other real applications. Contrary to other similar studies, we do not use the SDK benchmarks because we believe that synthetic benchmarks would remove real world aspects while at the same time they would produce similar conclusions.

We classify the kernels of our code in two categories. The first category includes all those kernels whose number of threads grows with the number of atoms in the molecular system. These kernels allow us to study how kernels expand and contract their thread pool to accommodate larger or smaller inputs, i.e., number of molecular atoms. Kernels in this category include: `BondForce`, `NBBuild`, `NonBondForce`, and `CoordsUpdate`. The `BondForce` kernel computes the bond-, angle-, and dihedral interactions. This kernel accumulates the appropriate forces by iterating through a list of all atoms bonded to or involved in an angle with other atoms. The `NBBuild` kernel builds the list of all atom pairs whose distance is within a certain cutoff. The `NonBondForce` kernel computes the non-bonded interactions (i.e., Lennard-Jones and direct space electrostatic terms) by iterating through the list of all atom pairs whose distance is within the defined cutoff. The `CoordsUpdate` kernel updates the coordinates of each atom.

The second category includes all those kernels that use mesh-based decomposition. These kernels allow us to study how kernels expand and contract their load on a fixed-size thread pool to accommodate larger or smaller inputs, i.e., number of molecular atoms. Kernels in this category include: `ChargeSpread`, `LatticeUpdate`, `BCMultiply`, and `CUFFTExec`. The `ChargeSpread` kernel performs

the charge spreading on a fixed-size mesh (or lattice). The `LatticeUpdate` kernel updates the mesh points as atoms (charges) move. The `BCMultiply` performs the multiplication of the charges stored at each mesh point by pre-computed structure constants. The `CUFFTExec` kernel performs the FFT computations, i.e., inverse and forward FFTs, and uses the CUDA FFT library.

We also consider two versions of FEN ZI. A first version that was initially implemented and optimized for CUDA 4.0 and is currently released (FEN ZI Release). The second version is called FEN ZI Development and is built upon FEN ZI Release but includes additional code optimizations by strictly following NVIDIA's guidelines. The main optimizations included in FEN ZI Development are:

- replacement of branching with masking
- replacement of slow operations on chars and shorts with INTs

As we will see in the next sections, these small changes that look minimal can have both positive and negative impacts on the overall simulation performance.

B. Datasets: membrane systems

For our performance study, we consider three different lipid bilayer membranes (DMPC) with the same structures but different dimensions (Figure 2), each one four times larger than the previous:

- Small (1x1): 46.8 Å X 46.8 Å X 76.0 Å, 17,004 atoms
- Medium (2x2): 93.6 Å X 93.6 Å X 76.0 Å, 68,484 atoms
- Large (4x4): 187.2 Å X 187.2 Å X 76.0 Å, 273,936 atoms

Note that, contrary to other datasets for which molecular systems or proteins with different atoms and bonds are used, our dataset allows us to rigorously study scalability because it considers three systems with exactly the same type of atoms and bonds but different sizes, i.e., 4 and 16 times larger respectively. Moreover the membranes are immersed into an explicit solvent that increases in number of water molecules in the same way as the membrane.

Molecular dynamics simulations of the three membranes are performed for 10,000 MD steps in constant temperature (we maintain the system temperature of 298K). The host

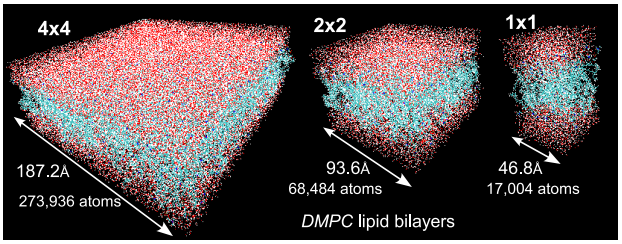


Figure 2. The 3 DMPC systems with their incremental sizes.

computer was a high-end workstation with dual quad-core Intel Xeon E5520 processors. The host Operating System was CentOS 6.2.

VI. Performance dissection

A. Level 1: amount of science

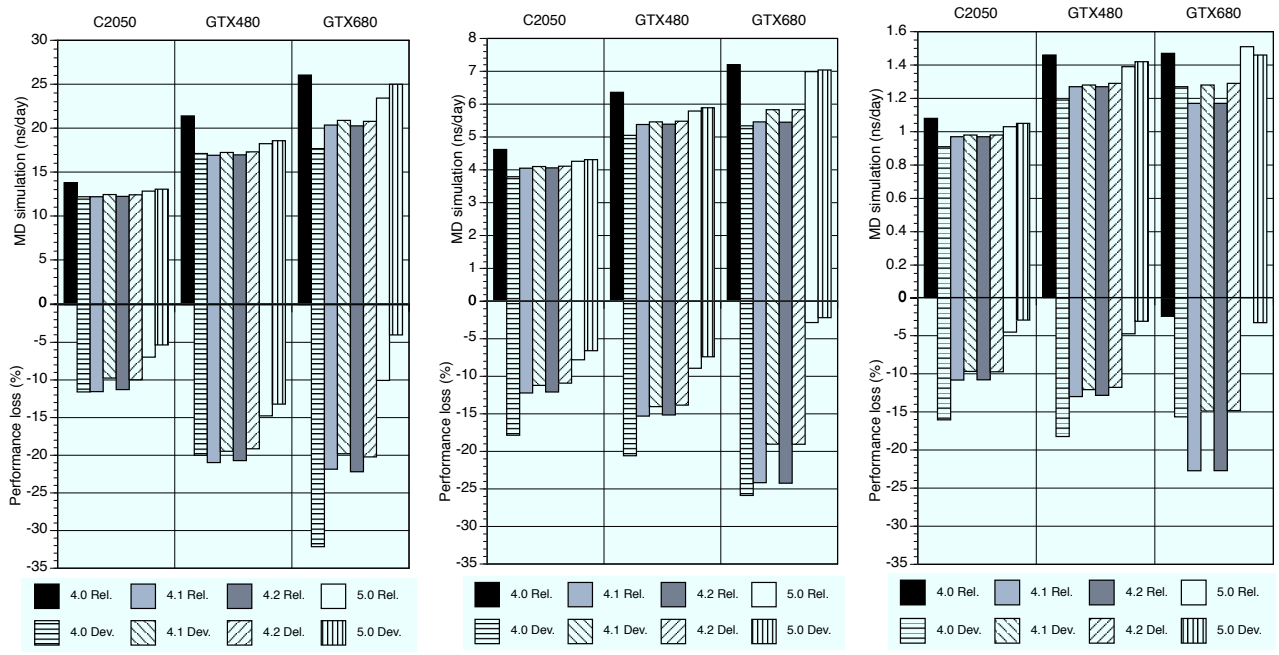
At the highest level of our performance dissection, we look at the overall science in nanoseconds of MD simulation per day that we can perform with FEN ZI by comparing this metric (i.e., ns/day) for the two FEN ZI implementations (i.e., Release and Development) and the three DMPC membrane systems (i.e., $1x1$, $2x2$, and $4x4$) when compiled with the four CUDA variants (i.e., 4.0, 4.1, 4.2, 5.0) and executed on the three GPUs (i.e., C2050, GTX480, and GTX680). We repeated each run 20 times and compute the average ns/day (the deviation was not significant).

Figure 3 shows the MD simulation times in ns/day (top sub-figure) and performance loss in percentage with respect to the best performance (bottom sub-figure) for the different scenarios considered in the paper. More specifically, the top sub-figure in Figure 3(a) shows the average ns/day of MD simulation achieved on a C2050, GTX480, and GTX680 for the two different FEN ZI implementations (i.e., Release and Development) when simulating the small membrane. The bottom sub-figure shows the performance loss in percentage for the same small membrane compared with the best performance. Figure 3(b) shows the ns/day and the associated performance loss for the medium membrane. Figure 3(c) shows the ns/day and the associated performance loss for the large membrane.

From Figures 3(a), 3(b), and 3(c) we see that, for our comparison of different CUDA variants for a given membrane on a given GPU architecture, CUDA 4.0 Release always serves as reference except for the FEN ZI simulation of the large membrane on GTX680. For the latter, CUDA 5.0 Release serves as the reference. In other words, CUDA 4.0 Release always outperforms the other CUDA variants across platforms and membranes except for large membranes on the newer Kepler architecture GTX680. We also observe that:

- Optimizations in FEN ZI Development slightly increase the performance for all CUDA variants except for CUDA 4.0. For the latter, the performance significantly slows down.
- On GTX GPUs, the performance gap is sensitive to the membrane size. The gap between CUDA 4.0 Release and CUDA 5.0 Release/Development shrinks as the membrane grows to the point that on the the GTX680 and for the large membrane, CUDA 5.0 Release outperforms the other conformations. Performance on C2050 seems less sensitive to the membrane size.
- FEN ZI shows linear scalability in performance across GPU architectures as the size of the membrane increases from small to medium and large.

One could argue that a loss in performance in the range of 10% is a marginal loss and thus tolerable in some applications. This is not the case for MD simulations when studying molecular phenomena like a peptide penetrating into a membrane while folding. These types of studies require the molecular dynamics simulation of medium to large membranes computed on the scale of hundreds of nanoseconds to microseconds (i.e.,



(a) Small membrane (b) Medium membrane (c) Large membrane

Figure 3. MD simulation times in ns/day and performance loss in percentage compared with best FEN ZI execution for different CUDA variants and GPU architectures.

several hundred days of simulations). A loss in performance of e.g., 10% can mean a loss in time of weeks or even months of simulations.

B. Level 2: times for critical kernels

At a lower level of dissection, we profile key application kernels. Figure 4 shows the impact in percentage over the total running time and the time variability for each one of the eight key kernels in FEN ZI. For each kernel we collect the time for 10,000 MD steps for the 72 scenarios (i.e., 2 FEN ZI implementations $\times$ 3 membrane sizes $\times$ 4 CUDA version $\times$ 3 GPU types). The box-and-whisker diagram in the figure shows the lower and upper quartiles at the top and bottom boundaries of the box for the kernels; the median is the band inside the box; the mean is the black square; the whiskers extend to the most extreme data points or outliers; and outliers are plotted individually as + symbols. From the eight key kernels in FEN ZI, we identify three of them that are critical for our analysis, and thus, worth of further investigation. Note their heavy impact on the overall running time and their time variability across the 72 scenarios. The kernels we selected are `NBBuild`, `NonBondForce`, and `ChargeSpread`. Note that we select representatives from both kernel categories: `NBBuild` and `NonBondForce` expand and contract their number of threads in the thread pool while `ChargeSpread` expands and contracts the thread loads.

`NBBuild` has a mean running time of 12% with respect to the other kernels and a standard deviation of 3%. `NonBondForce` is the most expensive kernel across runs, accounting for 37% of the time in average, a standard deviation of 9%, and a difference of 35% between its lower and upper quartile. This represents a very large variability across FEN ZI runs. Finally we chose `ChargeSpread` because it has a slightly skewed distribution and it is responsible for 10% in average of the kernels' running time, with a standard deviation of 4%. Note that the computation in `LatticeUpdate` is very similar to `ChargeSpread` and therefore we will not dissect this kernel.

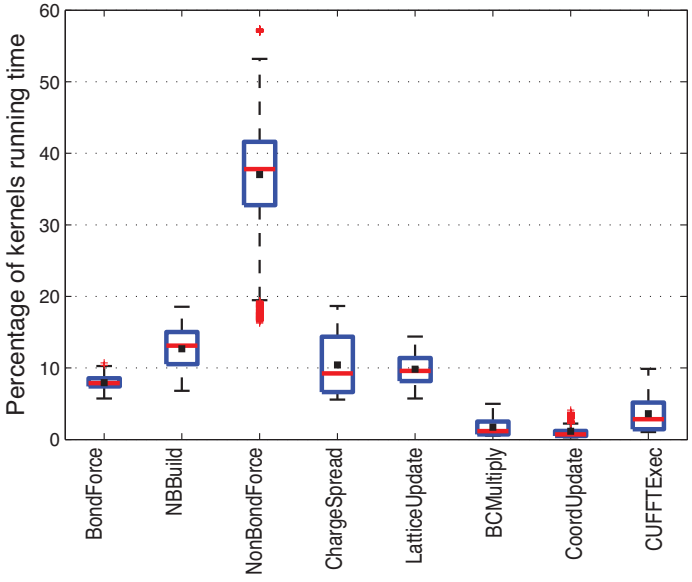


Figure 4. Percentage over the total running time and the time variability for each one of the eight key kernels in FEN ZI

In order to systematically dissect the overall behavior of

FEN ZI, we compare performance of the three selected critical kernels per architecture, CUDA version, and data size. In Figure 5 we show this comparison. Recall that the small dataset represents a DMPC of dimension $1x1$, the medium dataset is a DMPC of $2x2$ dimension (4 times larger than the small dataset), and the large dataset is a DMPC of $4x4$ dimension (16 times larger than the small dataset). For this analysis we normalize the running time of each kernel with respect to the small membrane; thus, we divided the run time of FEN ZI kernels by 4 and by 16 for the medium and large datasets respectively. For each of the three selected kernels we measure the average time across 20 runs.

For each row, the sub-plots to the left correspond to the C2050 platform, GTX480 and GTX680 are the middle and right sub-plots respectively. Every CUDA version is differentiated by the symbol used: version 4.0 is a black square, version 4.1 is a blue triangle facing upwards, version 4.2 is a green triangle facing downwards, and version 5.0 is a red circle. We also differentiate the Release code (R) and Development code (D) by using filled and empty symbols respectively. The different membrane sizes are depicted using small, medium, and large marks. We show the trends followed by runs of the same CUDA version and different sizes by joining them with a line (solid line for the Release code and dotted line for the Development code).

Figure 5(a) shows the performance comparison for the `NBBuild` kernel to build the nonbond list. It is based on a cell-based neighbor list method. The total number of threads in the kernel is equal to the number of atoms. The domain is divided into equal cells of a give cutoff size and each thread (atom) searches for neighboring atoms only in its current cell and 26 adjacent cells. Across all cases, the medium membrane performs better than both small and large membrane.

Figure 5(b) shows the performance comparison of the `NonBondForce` kernel. Note that for better visualization, the scale of C2050 is different than the scale of GTX480 and GTX680. As with the `NBBuild`, the total number of threads in the kernel is equal to the number of atoms. The key feature of `NonBondForce` is a nested loop that computes for each atom (associated to a thread) the nonbond interactions with all the other atoms within a certain cut-off (all atoms listed in the atom's nonbond list). This kernel shows the highest difference between Release and Development codes for version 4.0 across architectures; this difference not only occurs in terms of magnitude as in C2050, but also in the relative impact of data size shown by Release and Development for GTX480 and GTX680, and for release between C2050 and both GTXs. Another interesting behavior of this kernel is the reverse trend that occurs in GTX680; while C2050 and GTX480 show better performance as the size of the datasets increase, the contrary is occurs for GTX680. Moreover, the performance gap between the different datasets is larger for GTX680 than for C2050 and GTX480. Finally, it is also interesting to note that, across architectures, FEN ZI runs using CUDA versions 4.1 and 4.2 perform consistently better than the newest version 5.0.

Figure 5(c) shows the performance comparison of the `ChargeSpread` kernel. Charge spreading on GPU can be parallelized easily by the grid points instead of the atoms. Each thread works on a single or a set of grid points. The key feature of the `ChargeSpread` kernel is that its total number of threads is function of $fftx * ffty * fftz$, where $fftx$, $ffty$ and $fftz$ are the FFT dimensions. Note that the FFT dimensions should grow with the size of membrane; they are $64 * 64 * 64$ for the small and medium system while they are $64 * 64 * 128$ for the large system - in other words the mesh is two times larger. The figure outlines a wide performance gap between the small membrane and the medium and large membranes. This gap is even larger for C2050 but decreases for CUDA versions 4.0 and 5.0 in the GTX architectures.

We can observe how:

- For `NBBuild`, the medium membranes always perform better than small and large membranes across GPUs and code variants.
- For `NonBondForce`, the CUDA 4.0 Release always perform significantly better than the CUDA 4.0 Development.
- For `NonBondForce` on the Fermi-based GPUs, the small membranes always perform worse than medium and large across GPUs and code variants. This is not the case for the Kepler-based GPUs for which it is the opposite: the small membranes perform better than the medium, and the latter perform better than the large.
- For `ChargeSpread` there is a wide performance gap between the small membrane and the other two membranes (medium and large).

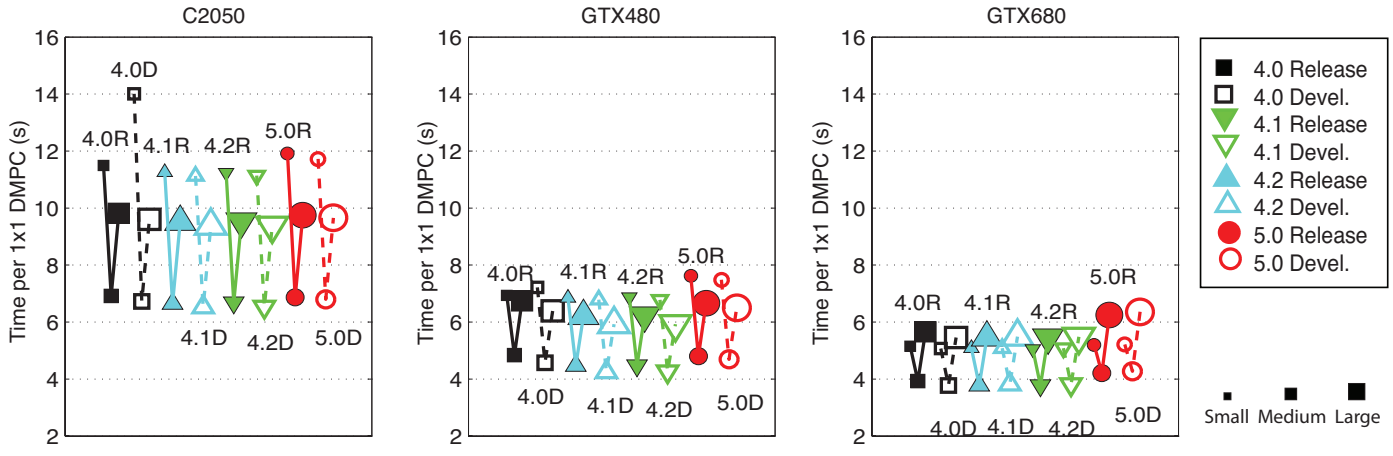
C. Level 3: resources for critical kernels

At the lowest level of our dissection, we profile resources using the NVIDIA Visual Profiler for the three critical kernels, i.e., `NBBuild`, `NonBondForce`, and `ChargeSpread`. We present the key metrics for two of the three GPUs, i.e., the Fermi-based GTX480 and Kepler-based GTX680. Key metrics we outline here allow us to identify whether a kernel is compute or memory bound. As a rule-of-thumb, a kernel is compute bound if the ratio of *instruction/byte* is two to three times larger than the ideal instruction/byte ratio. The latter is device dependent and kernel independent. For the Fermi-based GTX480, the ideal ratio of peak hardware GFlops/s to GB/s is 8.1 while for the GTX680 it is 16.1.

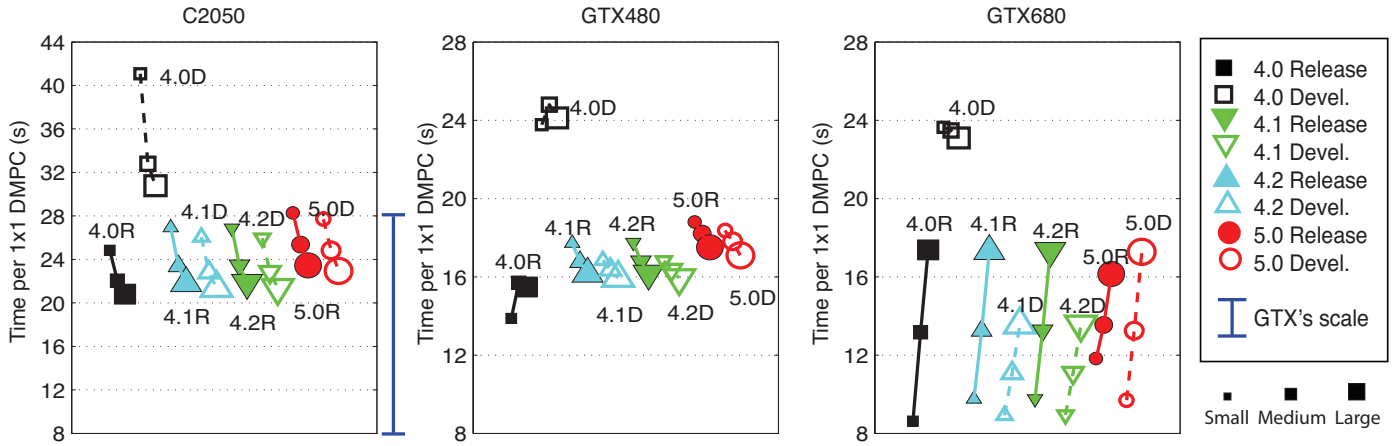
For both GPUs, we capture several metrics including execution time (ms), total instructions issued (GInst), warp execution efficiency (%), achieved occupancy (%), DRAM read (GB/s) and DRAM write (GB/s). We compute the total instruction throughput or (GInst/sec) as follows:

$$instructions\_per\_cycle/(gpu\_time * clock\_frequency)$$

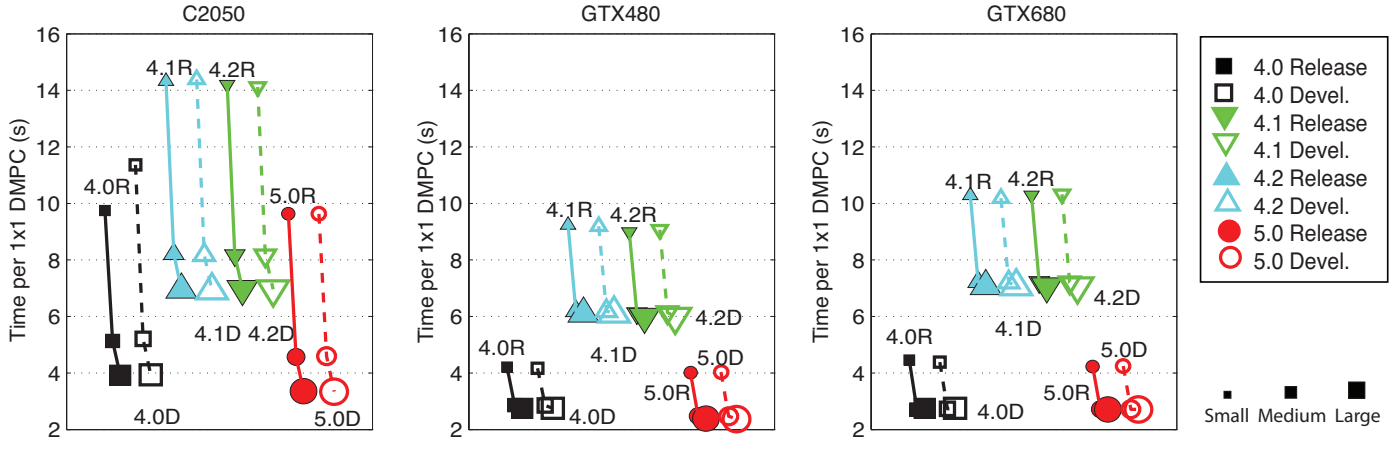
The total global memory throughput (GB/sec) can be easily computed as the sum of DRAM reads and DRAM writes. For the instruction:byte ratio, we assume that the instructions are 32-bit floating point operations. Note that, although the throughput for the other instructions may vary, our MD code's



(a) **NBBuild** kernel



(b) **NonBondForce** kernel



(c) **ChargeSpread** kernel

Figure 5. Time comparisons of individual critical kernels per membrane size, CUDA and GPU generations.

operations are for the large majority floating point operations. Thus, we compute the ratio of instructions per byte as follows:

$$\frac{32 * instructions\_issued}{128B * (global\_store\_transaction + l1\_global\_load\_miss)}$$

Table I shows the considered key metrics for GTX480, the three kernels, and the small and medium membranes both for FEN ZI Release and Development variants and CUDA variants. The goal for these measurements is to further study the impact of code optimizations by contrasting the Release vs Development code versions.

Table II shows the considered key metrics for GTX680, the three kernels, and the small, medium, and large membranes with both FEN ZI Release and Development and a subset of CUDA variants. The goal for these measurements is to test the effects of membrane size on scalability; we also want to analyze the performance exception of FEN ZI Release with CUDA 5.0 over FEN ZI Release with CUDA 4.0 and FEN ZI Development with CUDA 5.0. This latter phenomena inverses the performance pattern observed in the other scenarios.

When considered in concert, Tables I and II also provide us with the understanding of how Kepler's hardware features impact performance differently than Fermi's features. Both tables contain the following metrics for the three kernels: Execution time *'Exec. time (ms)'*, instruction throughput *'Inst. TPH (TInst/sec)'*, Giga-instructions issued *'Inst. issued (GInst)'*, percentage of branch efficiency *'Branch effic. (%)'*, percentage of achieved occupancy *'Achieved occup. (%)'*, DRAM read throughput *'DRAM read (GB/s)'*, DRAM write throughput *'DRAM write (GB/s)'*, and instructions per byte *'Inst/byte'*. The notation for the columns is the concatenation of CUDA version (**4.0**, **4.2**, **5.0**), code variant (**R** Release, **D** Development, and data size (**S** small, **M** medium, **L** large). For easier readability we underlined minimum values and highlighted with bold fonts maximum values.

From the two tables, we observe that, on the GTX480 GPU, kernels like `NBBuild` and `NonBondForce` are compute bound (i.e., higher instruction:bond ratio than the theoretical ratio) while `ChargeSpread` tends towards becoming memory bound (i.e., lower or equal instruction:bond ratio than the theoretical ratio). The GTX680 GPU transforms a compute bound kernel such as `NBBuild` into a memory bound kernel; `ChargeSpread` is clearly memory bound and `NonBondForce` remains a compute bound kernel. We also observe that:

- Factors damaging performance of CUDA 4.0 Development with respect to the other variants in GTX480 are as follows: (1) there is a decrease in instruction throughput of 700 GInst/sec; (2) there is an evident compiler slowdown that manifests in additional 500 Mega instructions issued; and (3) the total memory throughput of CUDA 4.0 Development is 1GB/sec smaller than the 4.0 Release and the other CUDA variants.
- Achieved occupancy and instruction issued are consistently better for CUDA 4.0 Release in the `NBBuild` kernel and for the GTX480 architecture.
- Memory throughput of CUDA 4.0 Release is always better than memory throughput of the other variants for `NonBondForce` kernel and for the GTX480 architecture.
- Branch efficiency of CUDA 4.2 and 5.0 drops significantly with respect to CUDA 4.0 in the `ChargeSpread` kernel running on GTX480.
- Instruction throughput of GTX680 increases up to one order of magnitude with respect to GTX480 across all the three kernels.
- Memory throughput of `NonBondForce` in GTX680 increases substantially with respect to GTX480. However, the proportion of instructions per byte decreases with respect to its ideal instructions per byte ratio compared to GTX480. This indicates a memory bottleneck in GTX680, perhaps due to its reduced memory bus.
- Instruction throughput of CUDA 5.0 Development decreases from 100 to 300 GInst/s with respect to CUDA 5.0 Release in the three kernels running on GTX680 and the large membrane for kernels `NBBuild` and `NonBondForce`. Particularly, in the `NonBondForce` kernel, instruction throughput of CUDA 5.0 Development goes from being the largest in small and medium datasets, to the smallest in the large dataset.

These observations together with the observations collected at Level 1 and Level 2 of our performance dissection drive and support our discussion and conclusions in the next section.

VII. Discussion

It is legitimate to wonder whether we are indeed getting closer to a general framework for CUDA that shares common layers with OpenCL. In some ways the answer is yes since we are now using similar languages, similar programming models, and a unified compiler back-end (i.e., the LLVM-based compiler). The motivation behind pursuing this generality are easier portability and maintenance on a long term. We have to keep in mind that generality always comes at a price. If applications are heavily optimized for CUDA 4.0 and the Fermi GPUs then the price is performance. This is also what we observe in our performance dissection. With the LLVM-based compiler, the same operations take more instructions. This can be observed with the increase of instructions issued and the slow-down of compute bound kernels such as `NonBondForce`. The latter kernel is also the dominant kernel, counting for $\approx 50\%$ of the GPU execution. Thus we ultimately observe how the science achieved with CUDA 4.0 (the CUDA variant that is most different from OpenCL in its structure in Figure 1) outperforms the other CUDA variants independently from the GPU generation used. In this search for a general framework, some other features are still different, i.e., a unified run-time is missing and different libraries are used. We observe that the run-time system does not seem to significantly impact the execution time of out kernels. Another difference across CUDA variants is their file compiling and linking methods, i.e., CUDA 4.0-4.2 rely on a single source file compiling and linking while CUDA 5.0 relies on a multi-file compiling and linking. We do not observe any performance change due to this factor.

The experienced performance loss cannot be completely recovered without further optimizations as it was for most cases of FEN ZI Development on CUDA 4.1-5.0. Note that our optimizations only partially mitigated the performance loss. More importantly, the aspect that comes in support of the loss in performance is the hardware itself and the evolutions of GPUs. With the higher number of stream processors and a different instruction pipeline, the new Kepler GPUs compensate some of the performance losses especially if combined

Table I
PERFORMANCE EVALUATION OF THE KERNEL NBBUILD FOR DIFFERENT MEMBRANES AND GPUS.

NBBuild	**4.0RS**	**4.0DS**	**4.2RS**	**4.2DS**	**5.0RS**	**5.0DS**	**4.0RM**	**4.0DM**	**4.2RM**	**4.2DM**	**5.0RM**	**5.0DM**
Exec. time (ms)	8.59	8.85	**9.59**	9.24	9.55	9.46	**35.35**	33.38	34.98	34.07	34.77	33.95
Inst. TPH (TInst/sec)	0.38	0.35	0.39	0.39	**0.39**	0.38	0.35	0.38	0.40	0.40	0.40	**0.40**
Inst. issued (GInst)	0.11	0.11	0.12	0.12	**0.12**	0.12	0.42	0.42	0.46	0.45	**0.46**	0.45
Branch effic. (%)	**87.74**	87.72	85.93	85.81	85.92	85.92	87.42	**87.82**	85.81	85.80	85.80	85.80
Achieved occup. (%)	**0.50**	0.44	0.43	0.43	0.43	0.44	**0.48**	0.48	0.48	0.48	0.48	0.48
Memory TPH (GB/s)												
- DRAM read (GB/s)	0.04	**0.04**	0.03	0.04	0.04	0.04	0.38	**0.42**	0.39	0.40	0.39	0.41
- DRAM write (GB/s)	**20.88**	19.64	19.47	18.50	18.47	0.00	21.41	**22.61**	21.68	22.24	21.65	22.24
Inst./byte	22.06	21.81	**24.09**	23.44	23.79	23.42	21.32	21.21	23.73	23.04	**23.77**	23.01
NonBondForce	**4.0RS**	**4.0DS**	**4.2RS**	**4.2DS**	**5.0RS**	**5.0DS**	**4.0RM**	**4.0DM**	**4.2RM**	**4.2DM**	**5.0RM**	**5.0DM**
Exec. time (ms)	1.39	**2.39**	1.87	1.82	1.89	1.83	6.58	**10.39**	7.61	7.44	7.61	7.43
Inst. TPH (TInst/sec)	0.57	0.50	0.59	0.59	0.59	**0.59**	0.60	0.54	0.62	**0.62**	0.62	0.62
Inst. issued (GInst)	0.02	**0.04**	0.03	0.03	0.03	0.03	0.12	**0.18**	0.15	0.14	0.15	0.14
Branch effic. (%)	97.67	**98.96**	95.33	90.74	95.33	90.75	91.51	**96.69**	95.44	90.97	95.44	90.97
Achieved occup. (%)	0.48	0.38	0.51	**0.51**	0.51	0.51	0.48	0.41	0.48	0.48	0.48	**0.48**
Memory TPH (GB/s)												
- DRAM read (GB/s)	**18.88**	11.01	13.92	14.44	13.97	14.50	**25.59**	15.19	21.94	22.66	21.93	22.66
- DRAM write (GB/s)	**0.78**	0.44	0.41	0.59	0.41	0.59	**0.70**	0.43	0.40	0.60	0.40	0.59
Inst./byte	28.61	**42.46**	40.32	39.93	39.94	39.99	33.71	**47.70**	39.97	39.20	39.93	39.26
ChargeSpread	**4.0RS**	**4.0DS**	**4.2RS**	**4.2DS**	**5.0RS**	**5.0DS**	**4.0RM**	**4.0DM**	**4.2RM**	**4.2DM**	**5.0RM**	**5.0DM**
Exec. time (ms)	**0.41**	0.41	0.38	0.39	0.39	0.39	**1.13**	1.13	0.97	0.97	0.97	0.97
Inst. TPH (TInst/sec)	0.48	**0.48**	0.45	0.45	0.45	0.45	0.47	0.47	0.49	**0.49**	0.49	0.49
Inst. issued (GInst)	**0.01**	0.01	0.01	0.01	0.01	0.01	**0.02**	0.02	0.02	0.02	0.02	0.02
Branch effic. (%)	93.59	**93.59**	61.35	61.25	61.54	61.44	**96.28**	96.28	76.05	76.08	76.07	76.04
Achieved occup. (%)	**0.84**	0.84	0.60	0.60	0.60	0.60	0.62	0.62	0.62	0.62	0.62	**0.62**
Memory TPH (GB/s)												
- DRAM read (GB/s)	24.98	24.97	26.35	**26.46**	26.40	26.40	24.02	24.01	**27.87**	27.85	27.86	27.87
- DRAM write (GB/s)	8.59	8.56	8.98	9.04	8.95	**9.06**	3.97	3.97	4.59	4.60	4.58	**4.60**
Inst./byte	**14.41**	14.40	13.36	13.40	13.39	13.36	17.78	**17.82**	16.37	16.37	16.35	16.47

Table II
PERFORMANCE EVALUATION OF THE THREE KERNELS GIVEN THEIR DIFFERENT SIZE IN GTX680

NBBuild	**4.0RS**	**4.2RS**	**5.0RS**	**5.0DS**	**4.0RM**	**4.2RM**	**5.0RM**	**5.0DM**	**4.0RL**	**4.2RL**	**5.0RL**	**5.0DL**
Exec. time (ms)	6.62	6.63	6.59	**6.72**	32.27	32.33	32.11	**32.94**	125.12	124.97	124.73	**126.22**
Inst. TPH (TInst/sec)	**3.13**	3.09	3.12	3.00	**2.40**	**2.40**	2.40	2.29	**2.34**	2.34	2.34	2.24
Inst. issued (GInst)	**0.09**	0.09	0.09	0.09	0.32	0.32	**0.32**	0.32	**1.25**	1.24	1.24	1.22
Branch effic. (%)	85.98	**86.12**	85.98	86.00	85.91	85.90	85.89	**85.93**	84.96	85.13	85.13	**85.14**
Achieved occup. (%)	0.54	0.54	**0.54**	0.54	**0.61**	0.61	**0.61**	0.61	0.62	0.62	0.62	0.62
Memory TPH (GB/s)												
- DRAM read (GB/s)	0.82	**0.85**	0.78	0.80	**1.07**	1.02	1.06	1.04	1.41	**1.41**	1.40	1.39
- DRAM write (GB/s)	27.69	27.34	27.49	**27.75**	**23.81**	23.78	23.60	22.62	**24.23**	24.21	24.13	24.07
Inst./byte	**18.67**	18.57	18.57	18.33	**16.84**	16.81	16.81	16.58	16.36	**16.46**	16.45	16.18
NonBondForce	**4.0RS**	**4.2RS**	**5.0RS**	**5.0DS**	**4.0RM**	**4.2RM**	**5.0RM**	**5.0DM**	**4.0RL**	**4.2RL**	**5.0RL**	**5.0DL**
Exec. time (ms)	**1.25**	1.25	1.25	1.01	5.77	**5.77**	5.60	5.51	26.43	26.42	26.39	**28.20**
Inst. TPH (TInst/sec)	5.03	5.03	5.03	**6.01**	4.57	4.58	4.58	**4.60**	4.06	**4.09**	4.04	3.71
Inst. issued (GInst)	0.02	**0.02**	0.02	0.02	0.11	**0.11**	0.11	0.10	0.43	0.43	**0.43**	0.42
Branch effic. (%)	95.32	**95.32**	95.32	90.14	**95.43**	**95.43**	**95.43**	90.32	95.20	**95.21**	**95.21**	89.76
Achieved occup. (%)	0.53	0.53	0.53	**0.68**	0.58	0.58	0.58	**0.71**	0.60	0.60	0.60	**0.72**
Memory TPH (GB/s)												
- DRAM read (GB/s)	22.28	22.37	22.34	**27.70**	46.42	46.55	46.44	**61.65**	49.88	49.92	49.86	**56.24**
- DRAM write (GB/s)	0.64	0.64	0.64	**0.88**	0.54	0.54	0.54	**0.62**	0.46	0.46	0.46	**0.50**
Inst./byte	28.63	**28.63**	28.63	27.72	28.63	**28.63**	28.63	27.77	**28.62**	28.61	28.62	27.79
ChargeSpread	**4.0RS**	**4.2RS**	**5.0RS**	**5.0DS**	**4.0RM**	**4.2RM**	**5.0RM**	**5.0DM**	**4.0RL**	**4.2RL**	**5.0RL**	**5.0DL**
Exec. time (ms)	0.42	0.42	0.42	**0.42**	**1.10**	1.10	1.05	1.07	4.24	4.24	**4.24**	4.19
Inst. TPH (TInst/sec)	2.42	2.42	**2.43**	2.42	**2.42**	2.42	2.42	2.42	2.52	**2.56**	2.47	2.50
Inst. issued (GInst)	0.01	0.01	**0.01**	0.01	0.02	0.02	0.02	**0.02**	**0.07**	0.07	0.07	0.07
Branch effic. (%)	58.92	**59.21**	58.99	58.99	75.57	75.59	**75.60**	75.58	74.01	74.11	74.14	**74.14**
Achieved occup. (%)	0.74	**0.74**	0.73	0.73	0.78	0.78	**0.78**	0.78	0.79	0.79	**0.79**	0.79
Memory TPH (GB/s)												
- DRAM read (GB/s)	24.43	**24.57**	24.44	24.55	24.61	**24.69**	24.62	24.67	32.74	**32.76**	32.49	32.72
- DRAM write (GB/s)	7.75	**7.77**	7.76	7.76	3.06	**3.09**	3.08	3.09	3.18	3.18	3.18	**3.19**
Inst./byte	16.28	16.29	**16.31**	16.29	17.90	17.88	17.85	**17.94**	**17.75**	17.67	17.63	17.68

with CUDA 5.0. So for example, the large number of stream processing units in the GTX680 increases by a factor of 10 the overall instruction throughput. Again this impacts most the compute-demanding kernels such as `NonBondForce` and ultimately the science delivered by GPUs. The different instruction pipeline (two independent instructions per warp can be dispatched each cycle [10]) has also a positive impact on the instruction throughput and this is notable in our study.

VIII. Conclusions

In this paper we document the trade-offs between software generality and hardware improvements for a diverse set of kernels in an open-source molecular dynamics code. Our performance dissection allows us to identify sweet spots where the loss in performance due to portability/maintenance is compensated by the hardware evolution. This is for large molecular systems on Kepler GPUs and CUDA 5.0. In this case, the faster hardware architecture is able to compensate the penalty associated to the software generality and ultimately can catch up in performance to become the fastest simulation.

We also document that trade-off between portability/maintenance and performance is a tough choice. Going back to the question that we raised at the beginning of this paper "Is the HPC community, that has been benefiting the most from the CUDA programming language and the GPU technology, ready to keep up with the challenges associated to the new CUDA and GPU directions?", we think the answer is yes as long as the hardware is able to support the scientific community by compensating the performance loss due to abstractions and generalizations. Not long ago, Sutter wrote in one of his articles how the "free lunch will soon be over" when referring to the Moore's law and the evolution of single processors [12]. We wonder whether we are starting a new season of free lunches, this time for multi-core programming in general and GPU programming in particular.

IX. Acknowledgments*

This work was supported by the U.S. Army, #ARO 54723-CS, NSF #941318, and by the NVIDIA University Professor Partnership Program.

References

[1] N. Ganesan, B. Bauer, T. Lucas, S. Patel and M. Taufer. Dynamic and Electrostatic Properties of Fully Hydrated Bilayers From Molecular Dynamics Simulations Accelerated with Graphics Processing Units. *J. Comp. Chem.* 32(14): 2958-2973, 2011.

[2] N. Ganesan, B. Bauer, S. Patel and M. Taufer. FENZI: GPU Enabled Molecular Dynamics Simulation of Large Membrane Regions Based on CHARMM Force Field and PME. In *Proc. of the Tenth IEEE Workshop on Hi-Performance Computational Biology (HiCOMB)*, May 2011.

[3] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-Accelerated Molecular Modeling Coming Of Age. *J. Molecular Graphics and Modelling* 29(2): 116-125, 2010.

[4] P. Du and R. Weber and P. Luszczek and S. Tomov and G. Peterson and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing* 38(8): 391-407, 2012.

[5] 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. *SIGARCH Comput. Archit. News* 38(3): 451-460, 2010.

[6] M. Harvey, G. Giupponi and G. De Fabritiis, ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *J. Chem. Theory and Comput.* 5: 1632-1639, 2009.

[7] S. Le Grand, A. W. Goetz, D. Xu, D. Poole and R. C. Walker. GPU Acceleration of AMBER PMEMD Calculations *URL: http://ambermd.org/gpus/*, 2010.

[8] J. E. Stone, *et al*.. Accelerating Molecular Modeling Applications with Graphics Processors. *J. of Comp. Chemistry* 28(16): 26182640, 2007.

[9] D. van der Spoel, *et al*.. GROMACS: Fast, Flexible, and Free. *J. Comput. Chem.* 26: 1701-1718, 2005.

[10] NVIDIAs Next Generation CUDATM Compute Architecture: Kepler TM GK110. *http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf*, 2012.

[11] B. Brooks, *et al*.. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comp. Chem.* 4: 187-217, 1983.

[12] H. Sutter. The Free Lunch Is Over. *Dr. Dobb's Journal*, 30(3), March 2005.