Effective Parallelization of Non-Bonded Interactions Kernel for Virtual Screening on GPUs

Ginés D. Guerrero, Horacio Pérez-Sánchez, Wolfgang Wenzel, José M. Cecilia and José M. García

Abstract In this work we discuss the benefits of using massively parallel architectures for the optimization of Virtual Screening methods. We empirically demonstrate that GPUs are well suited architecture for the acceleration of non-bonded interaction kernels, obtaining up to a 260 times sustained speedup compared to its sequential counterpart version.

1 Introduction

The discovery of new drugs is a complicated process that can enormously profit, in the first stages, from the use of Virtual Screening (VS) methods. The limitations of VS predictions are directly related to a lack of computational resources, a major bottleneck that prevents the application from detailed, high-accuracy models to VS. However, the emergent massively parallel architectures, such as the Cell Broadband Engine (CBE) and the Graphics Processing Units (GPU), are continuously demonstrating great performances in a wide variety of applications and, particularly, in such simulation methods [5].

The CBE [6] is composed of several (6, 8, 16) very fast independent specialised processors called Synergistic Processing Elements (SPEs) mainly optimised for single-precision floating point operations and capable of vector processing reaching a theoretical peak performance of around 230 GFLOPS. The newest generations of

Ginés D. Guerrero · José M. Cecilia · José M. García
Grupo de Arquitectura y Computación Paralela, Dpto. de Ing. y Tecnología de Computadores
Facultad de Informática, Universidad de Murcia, Campus de Espinardo 30100, Murcia, Spain
e-mail: {gines.guerrero, chema, jmgarcia}@ditec.um.es

Horacio Pérez-Sánchez · Wolfgang Wenzel
Institute of Nanotechnology, Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany
e-mail: {horacio.sanchez, wolfgang.wenzel}@kit.edu

GPUs are massively parallel processors which can support several thousand concurrent threads. Current NVIDIA GPUs contain up to 512 scalar processing elements per chip and are programmed using C language extensions called CUDA (Compute Unified Device Architecture) [3]. In late 2009, some models reached a peak performance above 1000 GFLOPS, which is 4 to 5 times the peak performance of the CBE.

In this paper, we focus on the optimization of the calculation of non-bonded interactions (such as electrostatics, van der Waals forces), as this kernel is an important bottleneck to different VS methods [5]. This kernel is widely used and implemented in several VS methods, concretely the docking program *FlexScreen* [2]. Different authors have already worked on its implementation and optimization; on the CBE, Schiller et al. [7] attained a 30 times speedup while Pérez-Sánchez et al. [4] achieved a 150 times speedup. On GPUs, Stone et al. [8] reached speedups of around 100 times, while Harvey et al. [1] achieve a 200 times acceleration. We test our kernel in GPUs to exploits the paralelism of this application, getting up 260 times speedup compared to its sequential version.

The rest of the paper is organized as follows. Section 2 introduces the GPU architecture and CUDA programming model from NVIDIA. Section 3 presents our CUDA implementation for the electrostatic interactions kernel. The performance evaluation is discussed in the Section 4. Finally, Section 5 ends with some conclusions and ideas for future work.

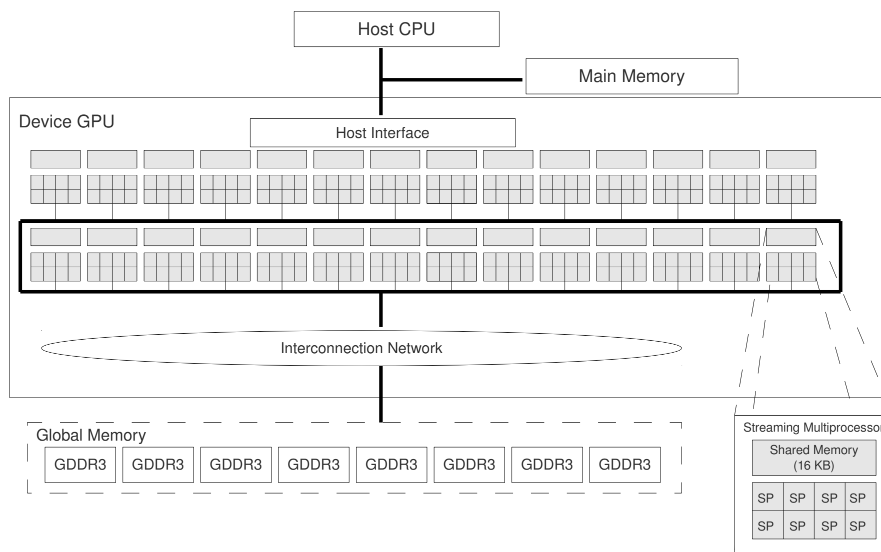

Fig. 1: Tesla C1060 GPU with 240 SPs.

2 GPU architecture and CUDA overview

In this section we introduce the main characteristics of the NVIDIA Tesla C1060 graphics card used in our experiments and the CUDA programming model.

The Tesla C1060 is based on scalable processor array which has 240 streaming processors (SPs) cores organized as 30 streaming multiprocessors (SMs) and 4GB off-chip GDDR3 memory called *device memory*. Each SM contains eight SPs, one double precision unit, a set of 16384 32-bit registers and a 16-Kbyte read/write on-chip *shared memory* that has a very low access latency (see figure 1).

The CUDA programming model allows write parallel programs for GPUs using some extensions of the C language. A CUDA program is divided into two main parts: the progam which run on the CPU (*host part*) and the program executed on the GPU (*device part*), which is called *kernel*. In a *kernel* there are two main levels of parallelism: CUDA threads, and CUDA thread blocks [3]. A block is a batch of threads which can cooperate together because they are assigned to the same multiprocessor. A grid is composed of several blocks which are equally distributed and scheduled among all multiprocessors, since there should be more blocks than multiprocessors (see figure 2). SMs create, manage, schedule and execute threads in groups of 32 threads, this set of threads is called *warp*. The *warp* is the scheduled unit, so the threads of the same block are scheduled in a given multiprocessor *warp* by *warp*. The programmer declares the number of blocks, the number of threads per block and their distribution to arrange parallelism given the program constraints (i.e., data and control dependencies).

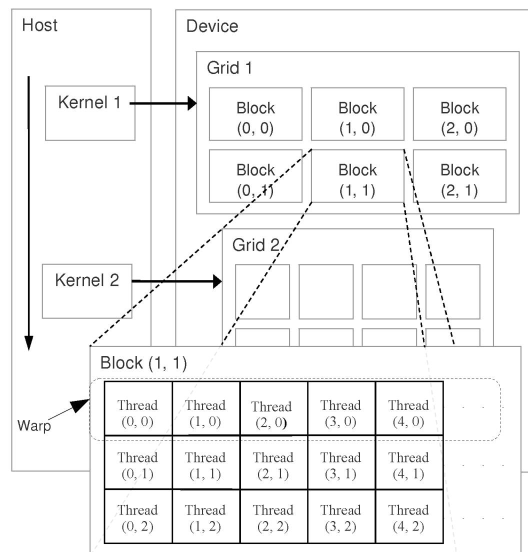

Fig. 2: CUDA programming model.

3 Our CUDA implementation

In order to exploit all the resources available on the GPU, and get the maximum benefit from CUDA, we focus first on finding ways to parallelise the sequential version of the electrostatic interaction kernel, which is show in the algorithm 1, where *rec* is the biggest molecule, *lig* the smallest molecule, *nrec* the number of atoms of *rec* and *nlig* the number of atoms of *lig*.

Algorithm 1 The sequential pseudocode.

```
1: for i = 0 to nrec do
2:     for j = 0 to nlig do
3:         calculus(rec[i], lig[j])
4:     end for
5: end for
```

Our best approach is that CUDA threads are in charge of calculating the interaction between atoms. However, the task developed by the CUDA thread blocks in this application can drastically affect the overall performance. To avoid communication overheads, each thread block should contain all the information related to the ligand or protein. Two alternatives come along to get this. The former is to identify each thread block with information about the biggest molecule; i.e. CUDA threads are overloaded, and there are few thread blocks running in parallel. The latter is exactly the opposite, to identify each thread as one atom of that molecule and then CUDA threads are light-weight, and there are many thread blocks ready for execution. The second alternative fits better in the GPU architecture idiosyncrasy.

Figure 3 shows this design. Each atom from the biggest molecule is represented by a single thread. Then, every CUDA thread goes through all the atoms of the smallest molecule.

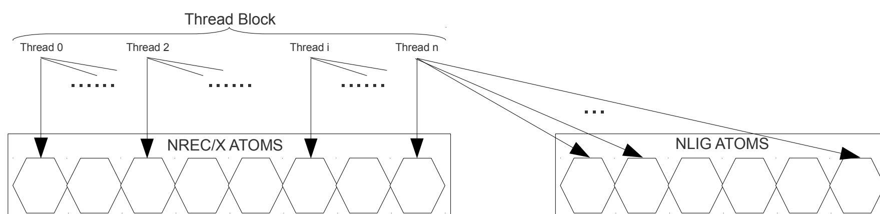

Fig. 3: GPU design for X thread blocks (with $X = 1$) with n threads layout.

Algorithm 2 outlines the GPU pseudocode we have implemented. Notice that, before and after the kernel call, it is needed to move the data between the CPU RAM and the GPU memory.

Algorithm 2 The GPU pseudocode.

```
1: CopyDataFromCPUtoGPU(rec)
2: CopyDataFromCPUtoGPU(lig)
3: numBlocks := nrec/numThreads
4: Kernel(numBlocks, numThreads)
5: CopyDataFromGPUtoCPU(result)
```

The kernel implementation is straightforward from figure 3. Each thread simply do the electrostatic interaction calculations with its corresponding atom of the *rec* molecule and all the *lig* molecule atoms.

CUDA Kernels

Kernel 1 Basic implementation

```
1: for all Blocks do
2:     for i = 0 to nlig do
3:         calculus(myAtomRec, lig[i])
4:     end for
5: end for
```

Kernel 2 Tiles implementation

```
1: for all Blocks do
2:     numIt = nlig/numThreads
3:     for i = 0 to numIt do
4:         copyBlockDataToSharedMemory(lig)
5:         calculusBlock(myAtomRec, ligBlock)
6:     end for
7: end for
```

We have derived two different implementations: the basic one (kernel 1), and the advanced one (kernel 2), where a blocking (or tiling) technique is applied to increase the performance of the application, grouping atoms of the *lig* molecule in blocks and taking them to the *shared memory*, taking advantage in this way of the very low access latency to the *shared memory*.

4 Performance evaluation

The performance of our sequential and GPU implementations are evaluated in a quad-core Intel Xeon E5530 (Nehalem with 8 MB L2 cache), which acts as a host machine for our NVIDIA Tesla C1060 GPU. We compare it with a Cell implementation [4] in a IBM BladeCenter QS21 with 16 SPE.

Figure 4 shows the execution times for all our implementations (both GPU and Cell) taking into account the data movement between the RAM memory and the corresponding device memory. All the calculations are done using simple precision floating point, due the smaller number of double precision units of the Tesla C1060. The benchmarks are executed by varying the number of atoms of the smallest molecule and also the number of atoms of the biggest molecule for studying both: a protein-protein and ligand-protein interactions. In this figure the performance of the Cell implementation, GPU basic implementation (GPU V1) and GPU tiles implementation (GPU V2) enhances along with the value of *nrec*, defeating the se-

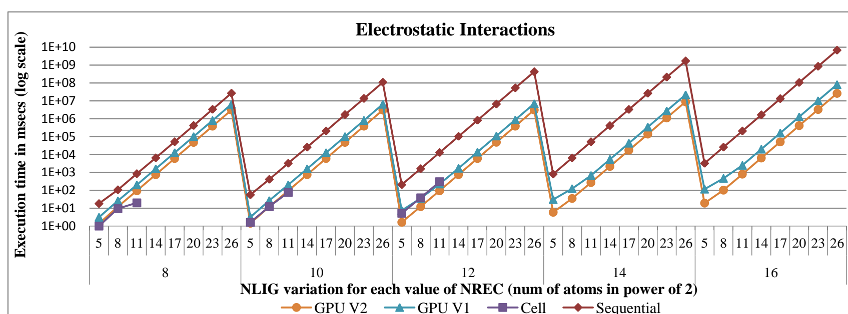

Fig. 4: Results obtained for different molecular size ratios. The execution time for the calculation of the electrostatic potential, in single precision, executed 128 times in a loop for different conformations of the molecule.

quential code by a wide margin (up to a speed factor of 260x). Notice that, the speedup factor between GPU and CPU increases faster when the value of *nrec* is higher. It is because the number of thread blocks running in parallel is also higher, and then the GPU resources are fully used. Similarly, for larger values of *nlig*, the speedup factor between GPU and CPU increases also because there are more threads running at the same time. However, it remains flat for a configuration greater than 256 threads per block.

Cell processor is not able to execute some of the biggest benchmarks due to its hardware constraints, mainly related to the 256K SPE Local Storage. However, it performs similarly compared to the GPUs for the smallest benchmarks in which the GPU is not fully used.

5 Conclusions and future work

In this paper we have introduced the kernel implementation for the calculation of non-bonded interactions applied to electrostatic interactions for different emergent parallel architectures. The results obtained for GPU are indeed promising, given the obtained speedup values up to 260x, compared to its sequential version. Cell processor gives similar results to GPUs only in some cases, where the molecules are small and the saturation situation for the GPU is not reached, but for higher workloads GPUs attain speedup values 7 times higher than the Cell processor. This way we can work with bigger molecules and thus perform more realistic calculations.

Given the adequacy of GPUs for the optimization of such calculations, our next step will be the implementation on the new GPU architectures such as the NVIDIA Fermi, which provides higher double precision floating point performance, and thus increasing the accurancy of the calculations. This parallel version of the kernel will be adapted and integrated into the docking program *FlexScreen*.

Acknowledgements This research was supported by a Marie Curie Intra European Fellowship within the 7th European Community Framework Programme (FP7 IEF INSILICODRUGDISCOVER), by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, and also by the Spanish MEC and European Commission FEDER under grants CSD2006-00046 and TIN2009-14475-C04.

References

1. Harvey, M. J., De Fabritiis, G.: An Implementation of the Smooth Particle Mesh Ewald Method on GPU Hardware. J. Chem. Theory. Comput. **5**, 2371–2377 (2009).
2. Kokh, D., Wenzel, W.: Flexible side chain models improve enrichment rates in in silico screening. J. Med. Chem. **51**, 5919-5931 (2008).
3. NVIDIA. CUDA Programming Guide 3.2. (2010).
4. Pérez-Sánchez, H. E., Wenzel, W.: Implementation of an effective non-bonded interactions kernel for biomolecular simulations on the cell processor. In: Gesellschaft fuer Informatik, Jahrestagung 2009 (Lecture Notes in Informatics). **154**, pp. 721-729 (2009).
5. Pérez-Sánchez, H. E., Wenzel, W.: Optimization methods for virtual screening on novel computational architectures. Curr. Comput. Aided. Drug. Des. **7**, 1–17 (2011).
6. Pham, D., Aipperspach, T., Boerstler, D., Bolliger, M., Chaudhry, R., Cox, D., Harvey, P., Hofstee, H., Johns, C.: Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. IEEE J. Solid-State Circuits. **41**, 179–196 (2006).
7. Schiller, A., Sutmann, G., Yang, L.: A Fast Wavelet Based Implementation to Calculate Coulomb Potentials on the Cell/B.E. In: Proceedings of the 2008 10th IEEE ICHPCC, IEEE Computer Society: pp. 162-168 (2008).
8. Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., Schulten, K.: Accelerating molecular modeling applications with graphics processors. J. Comput. Chem. **28**, 2618–2640 (2007).