Quantum Reactive Scattering Calculations on GPU

Leonardo Pacifici\textsuperscript{1}, Danilo Nalli\textsuperscript{2}, and Antonio Laganà\textsuperscript{1}

\textsuperscript{1} Department of Chemistry, University of Perugia, via Elce di Sotto, 8, 06123 Perugia, Italy
\textsuperscript{2} Department of Mathematics and Informatics, University of Perugia, via Vanvitelli, 1, 06123 Perugia, Italy

Abstract. An atom diatom time dependent reactive scattering code has been implemented on a platform made of a CPU and a GPU. The detailed analysis of the implemented code led to its restructuring to exploit the architectural features of graphic processors. Resulting gains of efficiency of the code when used for a prototype case study are compared.

1 Introduction

As part of our effort to build a virtual Research Community \cite{1} we have developed a Grid Empowered Molecular Simulator (GEMS) \cite{2} that can be used to simulate on the Grid \cite{3,4,5} \textit{ab initio} molecular and materials processes. To exploit the availability on the Grid of the innovative GPU (Graphic Processing Unit \cite{6,7}) architecture we have implemented some components of GEMS on a machine made by an Intel PC-QuadCore i7 with 6 GB of DRAM memory, equipped with a NVIDIA GTX 285 GPU, which communicates with the CPU via a PCI Express bus. Such machine is part of a larger cluster assembled as a science specialized platform made of many cores highly-parallel shared-memory units used by the virtual organization COMPCHEM \cite{8} of EGI \cite{9}. The NVIDIA architectural solution belongs to the SIMD (Single Instruction Multiple Data) \cite{10} class performing concurrent executions of the same instruction flow over large sets of data. From a functional point of view the NVIDIA architecture is based on \textit{Stream Processors} (SM), which are processors with a large number of elaboration units working in parallel.

The GPU utilized in our laboratory is a NVIDIA GTX 285 whose architecture is based on the NVIDIA SPA (Scalable Processor Array) and is equipped with a DRAM (Dynamic Random Access Memory) of 1 GB \cite{11}. The characteristic feature of this architecture is the use of a large amount (240) of SIMD (Single Instruction Multiple Data) computing cores connected to a shared small on-chip memory that allows a program to organize data as streams and express computations as \textit{kernels}. In particular, the SPA is made up of a certain number (depending on the the GPU model) of \textit{Texture Processor Clusters} (TPC). A single TPC contains a \textit{Geometry Controller} (GC), a \textit{Streaming Multiprocessor Controller} (SMC) and three \textit{Streaming Multiprocessor} (SM) executing computing instructions.
To enable GPUs to efficiently solve scientific problems, the NVIDIA CUDA [12] programming environment has been introduced allowing the use of programs written in the C (or C++) language and a better debugging. CUDA supports C/C++ high level programming language and is based on a thread-based execution model, shared memory mechanisms and synchronization mechanisms which can be managed by the user thanks to some extensions and additional constructs of the C programming language. These extensions provide the user with the possibility of partitioning the problem into decoupled subproblems which can be solved separately, thanks to cooperating threads (different concurrent execution threads for the same computation).

A CUDA application is made of sequential sections (generally executed by the host), and parallel sections (called kernels and executed by the device). In particular, a GPU kernel is written like a standard C function. C for CUDA, in fact, extends C by allowing the programmer to define C functions which are executed N times in parallel by N different CUDA threads, as opposed to the single execution in regular C functions. A kernel is organized as a grid of blocks, in which each block contains the same number of threads. These blocks are assigned sequentially to the Stream Processors in a coarse grained parallel fashion. At the same time the threads are dealt at a very fine grained parallel level. A thread belongs to a single block and is identified by a unique (among the kernel) index. Only threads of the same block can access the same shared memory. In particular, CUDA provides the possibility of labeling the block (using a two-dimensional index) and the threads (using a three-dimensional one). Therefore, while, as already mentioned, different kernels are executed sequentially, the threads and the blocks are executed concurrently. In particular, for a given computation the number of active threads depends on their organization inside the blocks as well as on the device available resources. Moreover, thanks to the CUDA primitives the GPU can access its global memory and the host can transfer data to/from it. However, while each SM can access its shared memory this is not possible for the host that can not manage this kind of memory: data allocated on the shared memory can be accessed only by a single block of threads.

This paper reports on the porting on a GPU of a time-dependent (TD) quantum reactive scattering program based on a wavepacket method that was recently ported on the Grid [5]. The program considered for that purpose is RWAVEPR [13], a Fortran code representing molecular systems as quantum wavepackets evolving in time under the effect of the system Hamiltonian. The code integrates the time-dependent Schrödinger equation for the generic atom diatom reaction

$$A + BC(v, j) \rightarrow AB(v', j') + C$$

having a reduced mass $\mu$ and the diatomic molecule in the $v$ and $j$ (for reactants) and $v'$ and $j'$ (for products) vibrational and rotational states, respectively. The related numerical procedure propagates the complex wavepacket in discrete time starting from an initial wavepacket $\Psi(R, r, \Theta)$ with $R$ being the atom diatom distance, $r$ the diatom internuclear distance and $\Theta$ the angle formed by the R