CMST SI(1) 71-79 (2010) DOI:10.12921/cmst.2010.SI.01.71-79

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **Special Issue 2010,** 71-79

High Performance Computing on New Accelerated Hardware Architectures

Marek Błażewicz, Krzysztof Kurowski, Bogdan Ludwiczak, Krystyna Napierała

Poznań Supercomputing and Networking Center
Applications Department
ul. Noskowskiego 10, 61-704 Poznań, Poland,
e-mail: {marqs/krzysztof.kurowsk/bogdanl/krysia}@man.poznan.pl

(Received: 15 July 2010; revised: 29 October 2010; published online: 23 November 2010)

Abstract: This paper presents recent work that has been performed in the context of high performance computing and hybrid architectures at Poznan Supercomputing and Networking Center. Three algorithms: JPEG2000 – compression/decompression, computational fluid mechanics and motion tracking have been parallelized on various architectures and compared to reference sequential applications. The performance results, implementation issues and best practices are discussed as well.

Key words: high performance, hybrid computing, NVIDIA CUDA, GPGPU, hardware accelerators

I. INTRODUCTION

Nowadays, major hardware vendors redirect their roadmaps toward highly parallel and power-efficient devices. They focus on increasing the number of low power computing cores rather than increasing their complexity and clock frequency. This paradigm shift means that further substantial progress in performance require highly parallel software codes and tight application to hardware mapping. However, we should note that end users are interested in the results of programming, and not programming itself. Therefore, we have started various programming and support activities at Poznań Supercomputing and Networking Center (PSNC) in order to help end users with the software development on new accelerated hardware architectures to meet their demanding computing requirements. We envision that many-core, hybrid and accelerated computing will play a significant role in the future e-Infrastructure in Poland and worldwide. Thus we have decided to focus on hardware devices commonly available for end users, as their great computational power comes along with low price and power consumption. Two hardware architectures fulfilling these requirements were designated: nVidia GPU with CUDA programming environment standing for GPGPU architecture, and Cell B.E representing a more hybrid solution. We are working with leading IT vendors, such as nVidia, IBM or Intel. We are also involved in some R&D efforts in national-wide infrastructure projects, e.g. Pl-GRID or POWIEW, and we have managed to create many proof-of-concept applications scenarios demonstrating added values behind the accelerated hardware applied for High Performance Computing (HPC). This paper aims to present some example algorithms, representing completely different applications areas, in particular Computational Fluid Dynamics (CFD) and image processing – motion tracking and JPEG2000 compression, and the way the new accelerated hardware can be applied for efficient computing. Moreover, this paper shows the main accelerated hardware characteristics, their influence on parallel programming and experimental results of application benchmarks comparing traditional CPU-based with GPU and Cell-based approaches. Both advantages and disadvantages of new hardware architectures and programming environments are presented. Finally, the paper is concluded with a number of best practices, experiences and learned lessons that we want to share with end users porting or planning to run their own applications on accelerated hardware.

The paper is organized as follows. Sections II and III briefly describe the GPU architecture (NVIDIA GTX 280 graphic card) and Cell B.E (QS21) architecture, respec-

tively. Sections IV-6 present the selected proof-of-concept applications implemented in PSNC on GPUs and Cells: Computational Fluid Dynamics, Motion Tracking and JPEG2000 compression. These sections describe the general ideas of the problem, GPU implementation details and obtained experimental results. Section VII presents other users of PSNC's accelerated hardware infrastructure and their applications. Section VIII summarizes this work.

II. GPU ARCHITECTURE

The increasing programmability of graphics processing units (GPUs) allows to use these chips not only for specific graphics computations for which they were designed, but for general-purpose computing problems. This field is called GPGPU (General-Purpose computation on GPUs).

Programmable GPU is a highly parallel, multi-threaded, many-core processor with a very high computational power and memory bandwidth. The main difference between CPU and GPU is that graphic cards, specialized for graphics rendering (e.g. compute-intensive and highly parallel computation), have more transistors devoted to data processing than to data caching and flow control. As a result, while computational efficiency is much higher, memory transfers between GPU and CPU can be a bottleneck in some applications.

GTX 280 graphic card consists of 30 multiprocessors (MP) with texture filtering and addressing units, a texture cache, a set of registers, a cache for constants, and a parallel data cache. Each multiprocessor has eight stream processors (SP). A multiprocessor is responsible for creating, managing, and executing concurrent threads. Lightweight thread creation, zero-overhead thread scheduling and fast barrier synchronization efficiently support programs that can be decomposed into parallel subproblems according to SIMD architecture (single instruction – multiple data).

Threads are arranged in blocks, each block being executed on one multiprocessor. To manage hundreds of threads, a multiprocessor creates, manages, schedules and executes threads in groups of 32 parallel threads called warps. As soon as a block of threads finishes its calculations, another block is launched to a multiprocessor. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data- dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. When all paths complete, the threads converge back to the same execution path. As a consequence, to improve the performance a programmer should provide as few divergent instructions in a warp as possible.

Due to these characteristic features of GPUs, not all of the problems can be mapped efficiently to this architecture. The user will achieve the best results for problems which require massive computations performed locally, so that fast computational units could overlap relatively slow transfers to the global memory. For example, an image processing algorithm that performs the same operation on every pixel of an image can be very efficiently ported to GPU architecture – each thread can be assigned one pixel and perform the calculations in a fast and highly parallel manner. Other good examples of applications well suited to GPUs are: video and sound processing, cryptography, bioinformatics, genetics, chemical and physical simulations, weather forecasting and climate research. In this paper we present three sample applications which we efficiently ported to GPU architecture, to show the drawbacks and advantages of this architecture: Computational Fluid Dynamics, JPEG2000 compression and Motion Tracking algorithm.

III. CELL B.E. ARCHITECTURE

Cell Architecture grew from a challenge posed by Sony and Toshiba to provide power-efficient and cost-effective high-performance processing for a wide range of applications, including the most demanding consumer appliance: game consoles. Cell B.E. (CBEA) is an innovative solution based on the analysis of a broad range of workloads in areas such as cryptography, graphics transform and lighting, physics, fast-Fourier transforms (FFT), matrix operations, and scientific workloads.

Cell B.E. processor is a multi-core chip comprised of a 64-bit Power Architecture processor core and eight synergistic processor cores, capable of massive floating point processing, optimized for compute-intensive workloads and broadband rich media applications. A high-speed memory controller and high-bandwidth bus interface are also integrated on-chip. The breakthrough multi-core architecture and ultra high-speed communications capabilities deliver vastly improved, real-time response, in many cases 10 times the performance of the latest PC processors. The Cell BE architecture is OS neutral and supports multiple operating systems. In this paper we present our implementation of JPEG2000 compression standard realized on Cell B.E. and GPU to compare the two architectures.

IV. COMPUTATIONAL FLUID DYNAMICS

Computational Fluid Dynamics (CFD) is one of the branches of fluid mechanics which uses numerical methods and algorithms to solve and analyze fluid flows. CFD is used in various domains, such as oil and gas reservoir uncertainty analysis, aerodynamic body shapes optimization (e.g. planes, cars, ships, sport helmet, skis), natural phenomena analysis, numerical simulation for weather forecasting or realistic visualizations. The CFD problem is very complex and needs a lot of computational power to obtain the results in reasonable time. However, because in fluid dynamics the behavior of particles depends only on the behavior of *local, surrounding* particles, CFD is well suited for the GPU architecture.

IV.1. Algorithm

In this work the Navier-Stokes equations were implemented. They are derived from Newton's Second Law and describe the continuous incompressible fluid. The Navier-Stokes equations are [1]:

$$\frac{Du}{Dt} = -\nabla\varphi + v\Delta u + f. \tag{1}$$

$$\nabla \cdot u = 0, \tag{2}$$

where:

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + u \cdot \underline{\nabla}, \tag{3}$$

$$\nabla = \frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z}, \tag{4}$$

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{5}$$

and

v – is a coefficient of *kinematic viscosity* of the fluid,
u – is the velocity of the fluid,
f – represents various body forces (like gravity),
φ – is the pressure divided by the constant density of the fluid.

To perform the computations on CPU or GPU, the mathematical model had to be discretized. Fluid was divided into the Euler mesh. Each part of the mesh represents a small part of the fluid defined by velocity and pressure. Parameters defining the fluid (pressure and velocity) are iteratively computed for the whole mesh by simulating fluid position in the next period of time $(t + \partial t)$.

In the incompressible continuous model of fluid dynamics, to preserve the law of conservation of the mass the velocity divergence has to be equal to zero (Equation 2). However, in the discrete model, due to the inaccuracy of calculations, the velocity divergence becomes a non-zero value. In order to fulfill the requirements of the incompressible fluid, velocity divergence is minimized during the process of pressure calculation (see [2] for details). This process has low computational complexity, but it is the most time-consuming part of the algorithm, because it needs to be launched iteratively in order to gain the desired accuracy.

During the discrete simulation of fluid flow, the quantum of time ∂t has to be small enough to guarantee the stability of the mathematical model. Otherwise it results in indeterministic model behavior. The value of ∂t was determined during experimental analysis and it depended on mesh granulation..

The surface of the fluid is defined by the *MAC* [3] (*Marker And Cell*) method. Each particle is moved by the velocity of this part of the mesh in which the particle resided. Fluid properties are calculated only for those parts of the mesh which contain some particles. The particles do not take part in the fluid computation, they just mark the position of the fluid and especially the surface.

IV.2. GPU Implementation

Taking into consideration that all operations in the equations are performed in the close neighborhood for each cell in the Euler's grid, this algorithm fits the GPU architecture very well. In each iteration, every block calculates the positions of particles in its part of the mesh and exchanges borderline conditions with neighboring blocks after every iteration. In this way heavy computations can overlap occasional transfers to global memory. What is more, the resolution of the Euler grid may be very easily enlarged (by allocating new computational blocks), and thus the algorithm used in this work is very scalable – it can either perform calculations more accurately by reducing the grain of calculations, or perform simulations for larger scale.

IV.3. Results

Simulation was tested on 6 different sizes of mesh, and the results were averaged over 1000 time steps. The tests were performed on nVidia GPU GTX 280 and the speed-up was measured over the referential sequential CPU code run on Intel CPU Core 2 Quad Q9550. First of all, we observed that not all functions of CFD simulation benefited from

Table 1. Speed-up of different functions used in CFD model

Size	Total time	Particles advection	Calculating divergence	Calculating *Z* factor	Calculating pressure	Calculating velocity	Recalc. surface and border criteria
64	9.72	51.34	1.04	3.94	2.45	4.48	0.3
128	22.2	87.61	3.12	10.83	5.95	17.61	0.82
256	43.16	105.73	26.38	25.18	19.16	28.38	1.78
512	49.35	102.66	37.34	31.95	23.82	35.56	2.64
1024	66.74	131.6	59.37	49.36	32.33	45.85	3.9
1536	74.46	137.75	49.56	63.19	41.67	49.78	3.31

porting to GPU in the same way. The functions that benefited the most from parallelization were those with a great amount of homogeneous computations and a small number of conditional statements. Others could not be well parallelized onto a graphic device because of the architecture limitations. Table 1 shows speed-ups for all functions used in our CFD model. The worst speed-up was observed for *recalculating surface and border criteria* function which is responsible for tracking the fluid interface inside the computational grid and changing the status of grid cells. It consists of conditional statements and data transfers from one part of the grid to another. This type of operations is executed effectively on CPU, but slowly on GPU. That is why this function runs only 1 to 4 times faster than on CPU. In spite of slow speed-up (or sometimes even slowdown), in most cases it is still more profitable to execute this portion of the program on GPU, because copying data between GPU and CPU would be more time-consuming than performing it on GPU, even slower but without data transfers.

The biggest speed-up is observed for *particles advection* function which moves the particles over the computational grid using the linearly interpolated in the particle's location speed. On GPU, the interpolation is computed by GPU's hardware in a structure called texture. The operation is very fast and the required data is cached, which makes it even more profitable. On CPU, the interpolation has to be implemented as a part of the software and thus is much slower. Moreover, *the particles advection* function performs very homogeneous computations and has fever conditional statements than other functions which represent a desired model of computations for the GPU architecture.

Figure 1 presents speed-up of total CFD simulation on GPU over a sequential code on CPU, depending on the mesh size. The GPU device is designed to compute massive amounts of data in parallel. When computations are launched

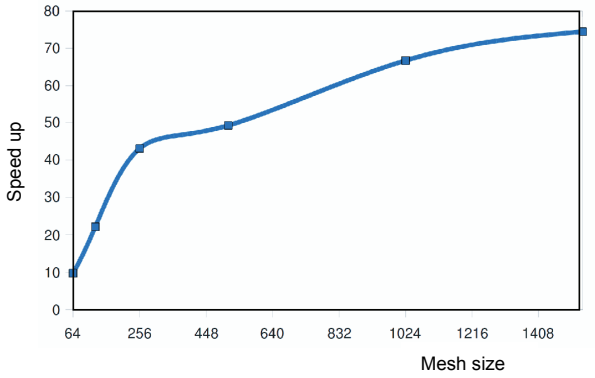


Fig. 1. Speed-up of GPU implementation depending on the mesh size

over a small (for this architecture) portion of data (domain size up to 256), the speed-up is not so significant because it does not take advantage of all available computational power. For domains smaller than 256, some computational units have no work assigned and stay idle. Larger domains use all available computational units, so the growth of speed-up is much slower. However, it still increases because the total time becomes less and less affected by a constant time of launching the code on external device. The irregularity of the second phase is caused by the fact that the computational blocks are not decomposed equally on multiprocessors at the beginning. This fact becomes less important with the growth of the computational domain and thus the deployed computational blocks. For the largest tested domain, the GPU version was up to 75 times faster than the sequential code.

V. MOTION TRACKING

Motion tracking is a process of identifying a moving object (or several ones) in time in a video stream from

a camera. The process of motion tracking can be divided into two challenging tasks:

1. How to choose the good targets (features) that will be easy to track.
   Good features to track are generally those parts of an image which have a high contrast. Therefore, neither plain regions nor edges are of interest (the former would be difficult to track in any direction, the latter can be tracked in only one dimension). The interesting points are called "corners" - points that are traceable in two dimensions.
2. How to associate target locations in consecutive video frames.
   This is especially difficult when the objects are moving fast relative to the frame rate, the image is blurry, the objects rotate in third dimension, there are rapid lighting changes or another object obscures the tracked object. The most common approach compares features between consecutive frames as it is considered to be robust to rotation and lighting changes.

Fig. 2. Feature tracking: Visualization of movement direction

The information about shifts of the corners between frames in a video stream can be used for various purposes. First, speeds of the objects in the scene can be tracked. Second, by analyzing the direction of movement, features belonging to the static landscape can be distinguished from the moving object (see Fig. 2 with our visualization of movement direction). Another information that can be retrieved from corner tracking is the depth of an image. If the video shows a static scenery and the camera constantly moves along, the depth of a picture can be estimated as the objects close to the camera move faster than the objects in a distance. Common applications include sports analysis, robotics, military tracking, industrial inspection and video surveillance.

V.1. Algorithm

To find interesting features in an image to track, the Harris corner detector is used. This approach is a popular point detector due to its strong invariance to rotation, scale, illumination variation and image noise. The Harris corner detector is based on the local auto-correlation function of a signal, where the local auto-correlation function measures local changes of the signal with patches shifted by a small amount in different directions. If changes of a signal are significant independently of the direction of a shift, it means a corner was found.

To calculate the Harris measure of "cornerness" of a pixel, first I_x and I_y derivatives of an image under a window are calculated. I_x and I_y derivatives are calculated by convolving the image with two separable Sobel filters. Then the matrix that captures the intensity structure of the local neighborhood is computed and its eigenvalues are determined. By analyzing the eigenvalues λ_1 and λ_2, we can describe the characteristic of a point. Three situations can be distinguished:

1. If both λ_1 and λ_2 are small, it means that the analyzed point is a part of a plain region.
2. If one eigenvalue is high and the other low, an edge was found.
3. If both eigenvalues are high, it indicates a corner.

For more details on this algorithm, see [4].

Having calculated the cornerness values for each point in the image, we set a threshold over which we consider a point as a good feature to track. To avoid having features located very close to each other, we divide an image into small parts and choose one representative (best) corner for each region.

The standard approach to *feature tracking* is to measure the correlation between the feature in one image and a point shifted by some distance in the consecutive frame. By comparing the correlations in the search region, we choose the best match as the new position of the tracked feature. Among the strategies to measure the correspondence between two features under mask, the SSD (sum of squared differences) is considered to be of good precision, but very time-consuming and hence applicable only in non-real-time programs [5]. The cost of computations depends mostly on the range of the search region. Confining to a small search region saves search time and makes the search process easier, but runs the risk of the tracked feature leaving the search region entirely between frames. The other factor that influences the performance but also

the time of computations is the size of patch which we treat as a feature and correlate between the images. Again, a small patch saves time, but is less accurate.

V.2. GPU Implementation

As the calculations for pixels are mostly local, the Harris algorithm can be effectively parallelized by dividing the image into smaller regions and assigning each region to a different block of threads. Each block calculates the cornerness for each pixel in its region and sends the position of the best pixel (representative of the region) back to host. The image data is saved in a texture memory. Using a texture provides a fast and cached access to the data and is especially well suited for image processing. For example, the methods for accessing the texture provide safe access to outside-the-border points. On CPU, safe access to the points behind a picture border must be explicitly ensured by a programmer. By implementing a more efficient algorithm on GPU, larger search regions and larger patches can be chosen, making the algorithm more reliable.

In feature tracking, the most natural way to parallelize the algorithm is to let each block calculate the best shift for one feature. It can be risky if there are only few features to track (the number of block would be too small) – the best performance will be achieved if there are more than 30 features, which is usually the case.

V.3. Results

Performances of the two functions (*feature finding* and *feature tracking*) were analyzed independently. Results achieved on GPU were compared to the results of running the analogous serial code on one core on CPU. We observed the speed-up of 3,66 for *feature finding* in an HD image compared to CPU. The Harris corner detector requires high memory occupancy, which prevents the GPU version from efficient overlapping the memory transfers with computations, thus the results were not impressive. The experiments for *feature tracking* were run on two video streams, one producing around 40 corners in *feature finding* function, and the other producing around 70 corners. A comparison was drown to find out how the number of tracked features influences the performances on CPU and GPU. Here the differences were more significant, because memory requirements were lower and the calculations were more suited to GPU architecture. The data could be stored in texture memory, which enabled to use built-in management of outside-the-border pixels and interpolation of data. This simplified the code compared to CPU, where safe access to the points behind a picture border had to be explicitly ensured by a programmer.

The results showed that the time on CPU increases linearly with the number of features while the time on GPU dos not change. As a result, for 40 features the GPU version was 45 times faster than CPU, while for 70 features the speedup was of 58. The GPU algorithm is therefore more scalable and can be applied for real-time tracking in high-resolution video stream.

VI. JPEG2000

JPEG2000 is a new standard for picture encoding in digital movies. A movie with 4K (4096 × 2160) resolution demands a very fast real time encoding solution to distribute it, for eexample,. via live broadcasts. Although there are some hardware implementations that offer real time encoding, they are costly because specialized hardware is required. Current consumer-level architectures with software implementations can provide low-cost alternative to hardware solutions. Therefore, we wanted to use new hybrid computing architectures, GPGPU and CELL B.E., to implement low-cost software base alternative solutions and at the same time to compare these two architectures.

The most important and computationally costly part of the JPEG2000 image compression standard is a Discrete Wavelet Transform (DWT), a signal processing technique for extracting information based on sub-coding. It can represent data by a set of coarse and detailed values in different scales. DWT is frequently used in many practical applications such as audio analysis, image compression and video encoding. In image compression DWT is used to decompose data into the horizontal and vertical characteristics. It is a one-dimensional transform in nature, but applying it in the horizontal and vertical directions forms a two-dimensional transform, which results in four smaller images. The DWT process can be repeated a number of times and it is then called a dyadic decomposition (see Fig. 3). The Cohen-Daubechies-Feauveau wavelet is one of the most commonly used set of discrete wavelet transforms in image compression. There are two versions of CDF wavelets: reversible integer-to-integer (CDF 53) and non-reversible real-to-real (CDF 97) wavelet transforms. The reversible transform uses only rational filter coefficients during compression and no data is lost due to rounding. It is called lossless compression. The non-reversible transform called lossy compression uses non-rational filter

coefficients. Both of these transforms are implemented in the JPEG2000 image compression standard [8], which has better performance compared to the JPEG standard [7].

VI.1. Algorithm

DWT can be realized by the lifting-based wavelet transform proposed by Sweldens [8]. Lifting-based filtering is done by using four lifting steps, which update alternately odd or even sample values. Figure 4 shows a basic scheme of the DWT algorithm. First, in the horizontal transform a source image data rows using a lifting procedure are decomposed into a set of even samples and a set of odd samples. Then, samples are exposed to the deinterleaving procedure and the image data is transposed to represent rows as columns. The whole process is repeated to create a 2-dimensionally transformed image data.

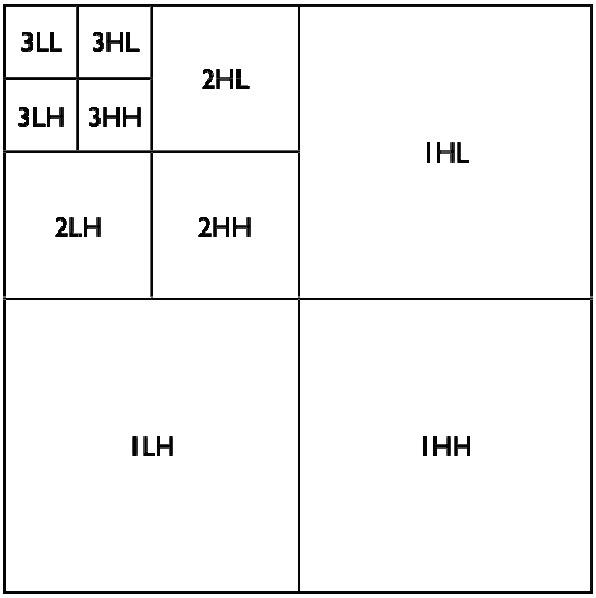


Fig. 3. Dyadic decompression used in JPEG2000

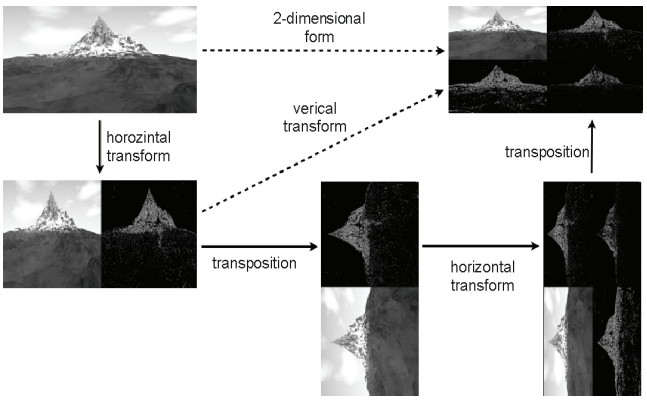


Fig. 4. DWT97 algorithm scheme

VI.2. Cell B.E. implementation

The sequential algorithm given in [8] can be parallelized without any major changes by using the many – loops approach and computing it in two steps – horizontally (on all rows) and then vertically (on all columns or transposed rows). Columns are processed after rows, so there is a need to synchronize computations. However this approach does not use the Cell architecture efficiently, therefore we developed a parallel algorithm called *tiled DWT* which decomposes the problem differently. While the base DWT works on an image as a whole, the tiled version splits images into rectangles of the same size and invokes DWT on them independently. Our experiments showed that using tiled DWT improved the speed almost twice.

VI.3. GPU implementation

For GPU implementation, the problem was also decomposed as in Cell's *tiled DWT*. An image is divided into multiple tiles, and the DWT algorithm is applied on each tile. Each thread block processed one tile in the image. Compared to the non-tiled version, it resulted in a reduced number of kernel invocations and calls to the global memory which optimized the amount of calculations for a single thread to overlap memory transactions. It should be stressed that although the concept of parallelization of the algorithm for both GPU and Cell B.E. is similar, the optimization details are completely different.

VI.4. Results

The most important difference between Cell B.E. and the GPU is that Cell relies on heavy persistent threads, whereas GPU paradigm is to use very light-weighted threads. This has a huge impact on programming style for those architectures, resulting in the development of separate approaches. In our opinion the biggest drawback of GPU computing is a relatively high cost of memory transfers, which is not a problem in case of Cell B.E. thanks to the Element Interconnect Bus that allows the memory transfer and computation to overlap. On the other hand, the biggest drawback of Cell B.E. seems to be the PPE which uses a noticeable amount of time to initialize the threads compared to negligible time of thread creation in GPUs. The difference is presented in

Table 2. For the JPEG2000 application, heavy thread creation in Cell B.E. turned out to have a smaller impact on the overall performance than the long time of memory copy in GPU. As a result, Cell B.E. implementation achieved better speedup (see Table 3), although both implementations were very good compared to the sequential CPU version. However, in both cases in order to achieve satisfying results, a considerable number of optimization techniques had to be applied.

Table 2. Cell QS21 and GPU comparison

Device	Initialization	Memory copy	Computations
Cell B.E. : QS21	10,8 ms	altogether 0.7ms (using double buffering)	
GPU: GTX 280	0.15 ms	16.9 ms	2.29 ms

Table 3. Cell QS21 and GPU comparison: total speedup

Device	Time	Speedup
x86	1500 ms	1.0x
Cell B.E. : QS21	11.5 ms	130.43x
GPU: GTX 280	19.5 ms	76.92x

External applications and other works

Apart from the topics described in this paper, we are involved in many other application areas which are expected to gain performance increase or computation quality by porting them to the new architectures. We cooperate closely with several other application teams and support them with our expertise in porting the existing applications to various new architectures, or provide them with access to experimental testbeds that consist of the most sophisticated GPU solutions, IBM PowerXCell based servers or modern FPGA boards.

Let us mention some of the most interesting application areas we are active in:

- efficient implementation of DNA alignment algorithms with backtracking routine on multiple GPUs,
- general-purpose quantum-chemical calculations, including Hartree-Fock methods in Gaussian basis sets, DFT in Gaussian basis sets, DFT in Slater-type basis sets and others,
- medical image processing and analysis,
- complex protein structure simulation in biochemistry and biotechnology.

VII. CONCLUSIONS

The use of new accelerated hardware (such as GPUs and Cell B.E.) for high performance computing is in line with the current trend of "green computing" – increasing the computational power by using a lot of low power computing cores instead of using singular cores of high complexity. These architectures require tight application to hardware mapping. In this paper we have presented the characteristics of each architecture and several proof-of-concept applications from different domains, to demonstrate how to efficiently exploit the computational capabilities of this hardware.

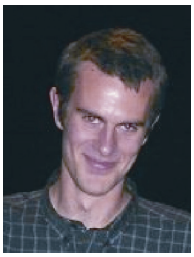
Using three case studies (CFD, Motion Tracking and JPEG2000) we have showed a number of important guidelines which should be considered while porting new applications to accelerated hardware. The most important learned lessons are:

- the proportion of computations to memory operations should allow to overlap costly data transfers with massively parallel computations;
- applications with structured accesses to memory and local computations are naturally suited for these architectures;
- calculations are efficient only for the datasets large enough to make use of all computational cores; otherwise the overheads, such as threads launching time, may degrade the performance;
- although codes will generally be more complex than for standard CPU, proper use of different memory structures can even simplify the code and programming effort;
- accelerated hardware architectures have different characteristics (concerning e.g. thread allocation and memory structures); it should be taken into account when choosing the most appropriate architecture for a given application.

All things considered, the use of the new accelerated hardware requires a considerable programming effort. However, if applied properly, it can satisfy very demanding computational requirements. The computations are done at lower cost, solutions are more scalable and/or more precise, and can reach impressive speed-ups, enabling for example an advanced real-time video processing or physical simulations. Considering fast development of new hardware architectures, dedicated languages and R&D communities arising around them to help end users with their applications, we envision that many-core, hybrid solutions using new accelerated hardware are an important point in the development of e-Infrastructure.

References

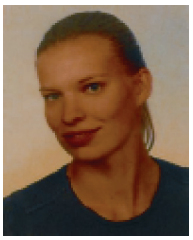
[1] A.J. Chorin, J.E. Marsden, A Mathematical Introduction to Fluid Dynamics (2000).
[2] F.H. Harlow, J.E. Welch, Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface (1965).
[3] M. Matyka, Computer Simulations in Physics (2002).
[4] K.G. Derpan, The Harris Corner Detector (2004).
[5] J.P. Lewis, Fast Normalized Cross-correlation, Industrial Light & Magic (1995)
[6] Nov. 2000. ISO/IEC 15444-1: Information technology, JPEG 2000 image coding system – Part 1: Core coding system, 2000.
[7] ISO/IEC 10918-1: Information technology, Digital compression and coding of continuous still images: Requirements and guidelines, 1994.
[8] W. Sweldens, *The lifting scheme: a new philosophy in biorthogonal wavelet constructions, in: proceedings of the SPIE.* Wavelet Applications in Signal and Image Processing III, 2569, 68-79S (1995).



MAREK BŁAŻEWICZ has graduated Computer Science at Poznań University of Technology. He works in Applications Department at Poznań Supercomputing and Networking Center. His research is focused on new hardware architectures and designing efficient and computationally intensive applications for them.

KRZYSZTOF KUROWSKI holds the PhD degree in Computer Science and he is leading now Applications Department at Poznań Supercomputing and Networking Center, Poland. He was involved in many EU-funded R&D projects in the areas of Information Technology and Grids over the last few years, including GridLab, inteliGrid, HPC-Europa, or QosCosGrid. He was a research visitor at University of Queensland, University of Wisconsin, University of Southern California, and CCT Louisiana University. His research activities are focused on the modeling of advanced applications, scheduling and resource management in HPC and networked environments. Results of his research efforts have been successfully presented at many international conferences and workshops.

BOGDAN LUDWICZAK graduated from Poznań University of Technology in Computer Science. He is a member of Applications Department at Poznań Supercomputing and Networking Center, Poland and now he is a leader of New Architectures Lab. He was involved in many EU-funded R&D projects in the areas of Information Technology and Grids over the last few years, including GridLab, ACGT or OMII-Europe. His research activities are focused on the modelling of advanced applications for modern hybrid computer architecture and advanced visualisations.



KRYSTYNA NAPIERAŁA is a PhD student at Poznań University of Technology in the Institute of Computing Science, Laboratory of Intelligent Decision Support Systems. Her PhD research concerns learning strategies from imbalanced data. She also works in Applications Departament at Poznań Supercomputing and Networking Center on GPU accelerated applications.