CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3692

SPECIAL ISSUE PAPER

Green computing on graphics processing units

Frédéric Magoulès*,†, Abal-Kassim Cheik Ahamed and Atsushi Suzuki

CentraleSupélec, Grande Voie des Vignes, 92295 Châtenay-Malabry Cedex, France

SUMMARY

To answer the question 'How much energy is consumed for a numerical simulation running on Graphic Processing Unit?', an experimental protocol is here established. The current provided to a graphic processing unit (GPU) during computation is directly measured using amperometric clamps. Signal processing on the intensity of the current of the power supplied to a GPU, with noise reduction technique, gives precise timing of GPU states, which allow establishing an energy consumption model of the GPU. Energy consumption of each operation, memory copy, vector addition, and element wise product is precisely measured to tune and validate the energy consumption model. The accuracy of the proposed energy consumption model compared to measurements is finally illustrated on a conjugate gradient method for a problem discretized by a finite element method. Copyright © 2015 John Wiley & Sons, Ltd.

Received 3 May 2015; Revised 14 August 2015; Accepted 31 August 2015

KEY WORDS: Green computing; GPU; CUDA; Alinea; Cusp; linear algebra; sparse matrix–vector product; iterative methods; conjugate gradient; compressed sparse row format

1. INTRODUCTION

During the last decade, general-purpose computing on graphic processing unit (GPGPU) proved to be extremely powerful and efficient to accelerate graphics (video games, etc.) and numerical simulations (seismic imaging, etc.). GPGPU proved to be an alternative to classical approaches in parallel computing and showed a great promise for supercomputers toward exascale and petascale computing. The power of graphic processing unit (GPU) can be a real advantage when considering the treatment of an application with huge data size.

Recently, mobile devices such as personal computers, cell phones, and tablets are equipped with powerful GPU that can be harnessed for general purpose computations. Newly, Nvidia expanded its compute-unified device architecture (CUDA) parallel-processing architecture to mobile devices. Usually, embedded systems (aircraft, vehicle, drone, gravimeter, etc.) run with limited computer hardware resources, such as memory and performance. The recent idea proposed to help the performance of embedded processors is to couple the classical microprocessors with GPU devices in order to accelerate the computations. Having GPU available on embedded systems is interesting for a number of reasons, including real-time applications for solving large problems. However, these mobile devices are battery-dependent, that is, the battery capacity is dependent upon the equipment level in the device, therefore it is crucial to take into account the percentage of energy consumed by the GPU accelerator, which is energy-greedy. Beyond our interest in accelerating computations, we have been interested to reduce the energy consumption of embedded system.

*Correspondence to: Frédéric Magoulès, CentraleSupélec, Grande Voie des Vignes, 92295 Châtenay-Malabry Cedex, France.
†E-mail: frederic.magoules@hotmail.com

The main contribution of this work is to analyze and understand the behavior of algorithms when using GPU computing in terms of energy consumption. This analysis can be helpful in designing future GPU-accelerated energy-efficient embedded system. It can for instance be used in managing automatic switching between ordinary microprocessors, GPU, and hybrid microprocessors according to the needs of an algorithm, equipment level, and available resources, for example, the battery usage. The data used in this analysis are coming from gravity surveys using gravimeter. Recent gravimeters (ZLS dynamic gravity meter, absolute quantum gravimeter, micro-g LaCoste absolute gravimeters, etc.) include embedded processors connected to a host computer. In the *ZLS dynamic gravity meter*, the host computer stores all data on a hard disk and can simultaneously display data to the video monitoring. Real time is one of the most important constraints. GPU computing is an excellent candidate to achieve real-time rendering. GPU is a fast computational unit, which is approximately 10 times faster than a multicore CPU with similar energy consumption. The usage of hybrid systems with GPU is a promising way to construct energy-efficient embedded systems. Efforts to reduce energy consumption are based on hardware developments, for example, using special coolant for efficient heat exchange. Usage of GPU is *de facto* the future of embedded systems and mobile devices, but what about energy consumption for real applications?

For applied sciences and engineering problems, a system of partial differential equations (PDEs), ordinary differential equations, or algebraic differential equations is often used as a physical and mathematical model, which is then solved numerically. Nowadays, several physical parameters can be taken into account, coupling different physical models leading to more and more complex systems. To obtain numerical solution of PDE, discretization methods (finite difference/volume/element methods) are applied, and linear systems with very large sparse matrices are obtained. Several methodologies to solve this type of large linear system exist, for example, sparse direct solver, iterative solver, and combination of those. Krylov subspace methods are the most representative of iterative methods, where an approximate solution is updated repeatedly using products of sparse matrix and vectors. Several libraries are developed to provide fundamental components of the Krylov subspace methods [1–3]. One of the primary concerns for these libraries is the computational time needed to obtain the solution in a parallel environment, that is, 'How much time is necessary to solve the problem?' The second objective of these libraries is G Flop/s, because this indicator is useful to estimate performance on a new hardware by evaluation of the ratio between actual G Flop/s by the library and the peak G Flop/s by the hardware.

Nowadays, we need to point out another question for energy issue, that is, 'How much energy is consumed by the application?' For high-performance computing hardware, the appropriate answer gives a compromise between computational time and energy consumption. Numerical algorithms have already faced a similar question, that is, 'How much memory is necessary to solve the linear system?' This question is also very important, because memory limitation is one of the physical constraints for numerical simulation. If the memory is not enough, we have to give up the simulation or need to be satisfied with very slow out-of-core execution. Therefore, algorithms have been optimized in both directions to minimize computational time and memory usage. Unfortunately, there is no way to answer the new question on energy consumption only from the point of view of the software. To the best of our knowledge, there is no established way to measure real energy consumption for each application.

As a consequence, we have designed an experimental protocol to measure accurately the energy consumption of a GPU for fundamental and basic arithmetic operations and some heavy operations involved in numerical linear algebra with sparse matrices. This allows to build an accurate energy consumption prediction model of the GPU. This methodology could suggest a 'new vision of high performance computing' and answer some of the questions outlined in References [4–7] where the importance of energy consumption is highlighted when using GPUs.

The rest of the paper is organized as follows. Section 2 gives a short overview of GPU computing using CUDA environment. In Section 3, we describe the proposed experimental protocol, which allows measurement of the consumed energy by a GPU. In Section 4, measurement of energy consumption for basic operations on a GPU is reported, and a way of accurate evaluation and prediction of energy consumption is established. Section 5 presents energy consumption for a problem using two different linear algebra libraries for GPU computation. Finally, Section 6 concludes this paper.

DOI: 10.1002/cpe

2. GRAPHIC PROCESSING UNIT COMPUTING

Graphic processing units were originally used for graphics computing, for example, calculations of coordinates of vertices, edges, and surfaces, and lighting, coloring, and texturing with single floating point arithmetic. After the establishment of GPGPU usage, they have been equipped to manage double floating points arithmetic and have been widely used for numerical simulation. The important and most attractive property of GPU comes from its developing speed of hardware components. For example, Nvidia GTX275, used in our experiments for the measurement of energy consumption, has 200 cores running with 0.633 GHz, which achieves 126.6 G Flop/s in double floating points, and is equipped with 127 GB/s-bandwidth memory. Nvidia GTX275 was produced in 2009. More recent GPU, such as Nvidia K20, has 2496 cores running with 0.706 GHz, which achieves 1.17 T Flop/s in double floating points, and is equipped with 208 GB/s-bandwidth memory. We can see that the computational speed of GPU becomes about 10 times faster, and memory access speed becomes two times faster within 4 years whereas both GPUs consume about 220 W electricity. This tendency is continuing almost during a decade. Figure 1 also shows the same tendency of computational speeds in single floating points with other leading GPU by Advanced Micro Devices in comparison to classical CPU from Intel. The key idea on designing GPU architecture is based on simpler structure of arithmetical and logical unit (ALU) than CPU with low clock frequency, for example, one-fourth of CPU but very large number of ALUs, Figure 2. The other important factor of the popularity of GPU computation is the software environment available to develop computational codes.

2.1. *Overview of compute-unified device architecture*

Compute-unified device architecture [8] is a framework for GPGPU computing developed by Nvidia. The first version of CUDA appeared in 2007, and it is updated almost every year. CUDA

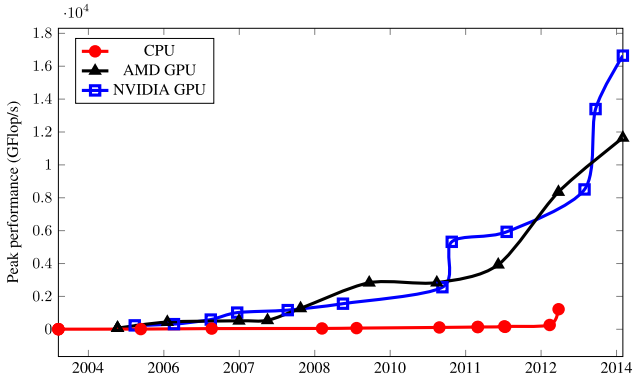


Figure 1. Evolution of the peak performance of graphic processing units (GPUs). AMD, Advanced Micro Devices.

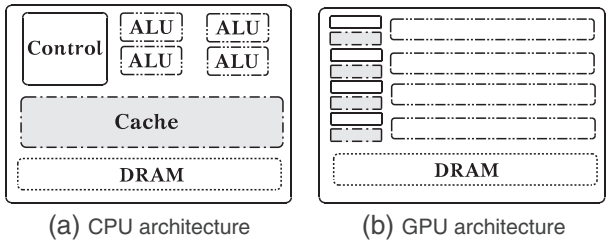


Figure 2. Difference of CPU architecture and graphic processing unit (GPU) architecture. (a) CPU architecture and (b) GPU architecture. ALU, arithmetical and logical unit; DRAM, dynamic random access memory.

DOI: 10.1002/cpe

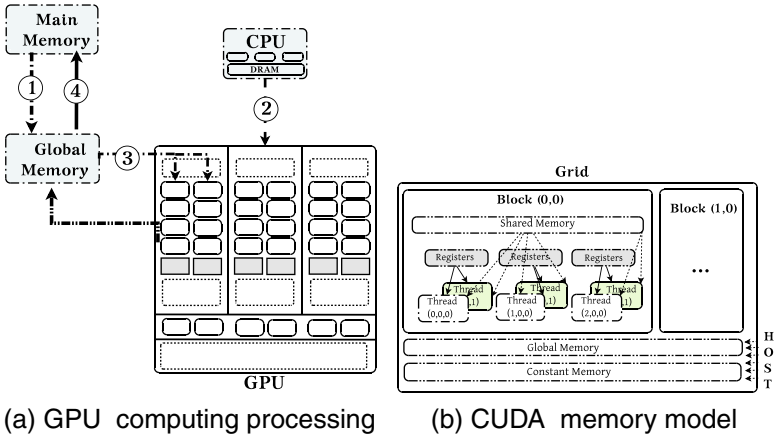


Figure 3. Compute-unified device architecture (CUDA) processing flow and memory model. (a) graphic processing unit (GPU) computing processing and (b) CUDA memory model. DRAM, dynamic random access memory.

defines both hardware and software environment with C/C++ based programming language to manage single instruction multiple data (SIMD)-type massively parallel execution by GPU. The CUDA software environment provides an abstract thread-computing model that could be common over different generations of GPU hardware. To be precise, CUDA parallel execution has more flexibility than SIMD parallel execution and is called as single instruction multiple threads. In the following sections, we will explain two key ingredients of the CUDA computation.

Following the development history of GPGPU as a graphics hardware, GPU with dedicated memory is packaged in a peripheral component interconnect (PCI) expansion board and connected to the host computer through PCI express bus [9, 10]. Here, GPU itself has no operating system, and general parallel executing flow is driven by CPU on the host computer. The flow of parallel computation using CUDA GPU is the following: Data are first copied from the main memory to the GPU memory, *global memory* (1). Then, the host (CPU) instructs the device (GPU) to carry out calculations (2). The *kernel* is then executed by all threads in parallel on the device (3). Finally, the results are copied back (from GPU memory) to the host (main memory) (4). Figure 3(a) illustrates this instruction flow.

2.2. *Memory model of compute-unified device architecture*

Graphic processing unit and CPU architectures have very different memory structure. This difference forces us to use a precise way of optimization of the code, but such optimization brings better performance in GPU.

As seen in Figure 2(a), CPU has a connection to the main memory through cache memory. When CPU accesses data, either of the two situations might happen, that is, to access the main memory or to access the cache memory. The cache memory can store the last accessed data. If data, which once have been read from the main memory, will be reused several times; the CPU needs to access only to such data located in the cache memory. However, we need to notice that the usage of cache memory is completely automatic and there is no way to control the behavior of the cache memory from user software.

Graphic processing unit system has a more complicated hierarchy of memory. GPU can access *register memory*, *shared memory*, *constant memory*, and *global memory*, that is shown in Figure 2(b). The *register memory* is seen as local memory of an ALU in GPU. This memory is accessed in the fastest speed in the GPU board with low latency. In the execution of the code, this latency will be completely hidden by execution of appropriate number of *threads*. The *shared memory* is the next faster memory that can be accessed from all ALUs in a group called a *block*. Then, the *shared memory* is used as a communication buffer between ALUs in a *block*. In CUDA programming user can declare usage of *shared memory* explicitly, which leads to big difference in the

DOI: 10.1002/cpe

code compared to CPU. The *constant memory* also provides faster access, but it is a only readable one. It is necessary to use host CPU to rewrite the data in the *constant memory*. The *global memory* is the main memory of the CUDA GPU, which can be accessed from all ALUs of GPU. The results of computation by ALUs are written in the *global memory* and sent to the host, in reality, which is performed by opposite way, that is, the host reads data through the PIC express bus. Both *constant memory* and *global memory* need to be allocated by the host computer. A comparison on speeds of access to memories can be summarized as follows:

$$\textit{register memory} < \textit{shared memory},$$
$$\textit{shared memory} < \textit{constant memory}, \text{and}$$
$$\textit{constant memory} \ll \textit{global memory}.$$

Because NVIDIA GPU uses GDDR5 memory with higher clock and wider memory bus, memory access speed of *global memory* is about four times faster than of the main memory of CPU, which consists of DDR3 memory. However, we need to be aware that *global memory* is shared by all ALUs of the GPU board and relative speed of the memory to an ALU is slow. Efficient memory access to *global memory* without idling of ALUs is the most important point in CUDA GPU computation for execution time and also for power consumption, because the GPU board consumes some amount of energy even though all ALUs are idling.

2.3. Compute-unified device architecture kernel and gridification

In CUDA terminology [11], the word *kernel* is used to denote a subroutine that is executed in parallel by ALUs of GPU. A *thread* is the smallest unit of processing that can be scheduled by an operating system. CUDA *threads* are controlled by the dedicated hardware inside of GPU. They are grouped into 32 *threads* that are assigned to one actual ALU hardware components. The unit of 32 *threads* is called a *warp*. This is a fundamental size of *thread* execution and is understood as a size of vector length of a vector. Several *warps* are grouped into a *block* whose size is given by user as a multiplier of 32. Inside of a *block*, all *threads* can access a *shared memory* and can be synchronized by the hardware with almost negligible latency. This usage of *shared memory* and synchronization inside of a *block* play a key role in computation of the inner product of two vectors for scientific computation. A set of *blocks* is called a *grid*. The global index of the current *thread* is obtained from a pair of indices, index for the *block*, and local *thread* index inside of the *block*,

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

where `blockDim.x` sets as number of *threads* per *block* by user. CUDA has capability to assign global index of current threads in two-dimensional or three-dimensional-shaped grid, which is originally supposed to manage tasks for the bitmap display. Generalization of these assignments of thread index is understood as gridification. The literature [12–14] demonstrated that the gridification, that is, the grid features and organization, has a strong impact on the performances of the kernels.

3. EXPERIMENTAL PROTOCOL

In this section, we describe a physical setting to measure energy consumption of GPU by direct measurements of the current sent to GPU from the host computer, and a technique to understand state of GPU from raw experimental data on the intensity of the current to the GPU with noise reduction.

3.1. Setting of the experimental protocol

In order to get accurate measurements of the energy consumption of a GPU-parallelized code, we need to measure the energy consumption by the GPU alone without consumption by other components of the computer. Unfortunately, there is no dedicated sensor on GPU that can measure energy with status of ALUs of GPU. As a consequence, we have to plug a device, *amperometric clamp*, on power supply cables of the GPU. An amperometric clamp measures the intensity of magnetic

DOI: 10.1002/cpe

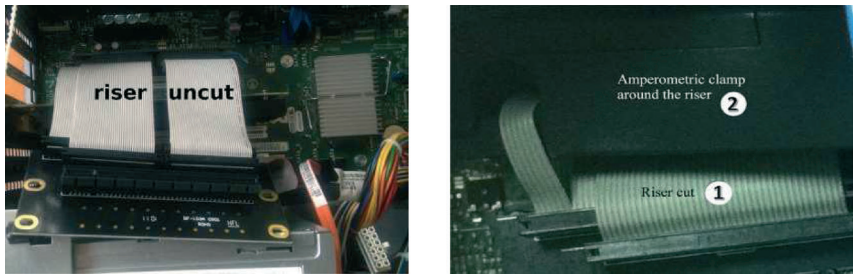
field created by the current circulation, and its output data are values of the current. This current is converted to voltage by an I-V conversion circuit for input of a digital oscilloscope to monitor time-dependent electric power. We note that we define positive or negative voltage according to the direction of the original current during the I-V conversion.

There is no space to clamp the amperometric device between the GPU card and the mother board, because GPU card is directly connected to the PCI express expansion slot. Then, we used an extension cable, called a *riser uncut* in Figure 4(a), which allows making space for the clamp. The power for the GPU is supplied through PCI express interface with two voltages, +3.3 and +12.0 V, and through supplemental cables with +12.0 V, directly from the power supply unit of the workstation. By specification of PCI express, each voltage is guaranteed as the constant value. The currencies on the PCI express slot for +3.3 and +12.0 V vary up to 3.0 and 5.5 A, respectively, which results in 75 W electric power at the maximum. The current of supplemental cables varies up to 18.75 A with 225 W electric power. We note that one GPU card can consume power up to 300 W. Figure 5 presents the experimental setting. For measurement of currencies for +3.2 and +12.0 V, we need to separate the +3.3 V cable from +12.0 V cable of riser uncut. A way of separation is shown in Figure 4(b), where (1) is the riser cut and (2) the amperometric clamp around the riser. Figure 6 shows the global wiring of the experiment, where (1) represents the GPU computer (workstation), (4) the first amperometric clamp, (2) the second amperometric clamp, (5) the USB plug for recording, and (6) the numerical oscilloscope.

In total, we used three clamps to measure each current of the power supplying cable.

By measuring currencies $I_{3.3V}$, $I_{12V\,riser}$, $I_{12V\,ext}$, we can get the energy consumption of the GPU card as

$$P = 3.3 \times I_{3.3V} + 12.0 \times (I_{12V\,riser} + I_{12V\,ext}).$$



(a) Riser uncut (b) Riser cut and connected to a clamp

Figure 4. Riser uncut and riser cut. (a) Riser uncut and (b) riser cut and connected to a clamp.

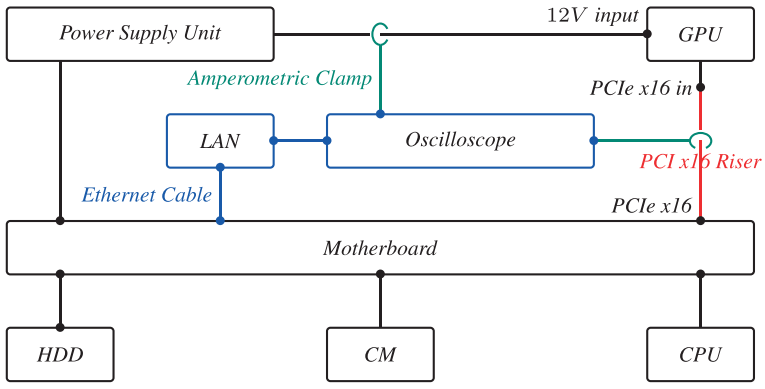


Figure 5. Schematic representation of the protocol. GPU, graphic processing unit; PCI, peripheral component interconnect; LAN, local area network; HDD, hard disk drive; CM, central memory.

DOI: 10.1002/cpe

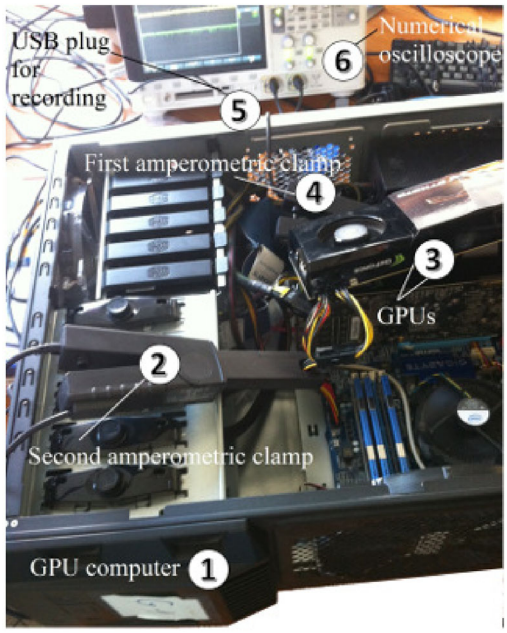
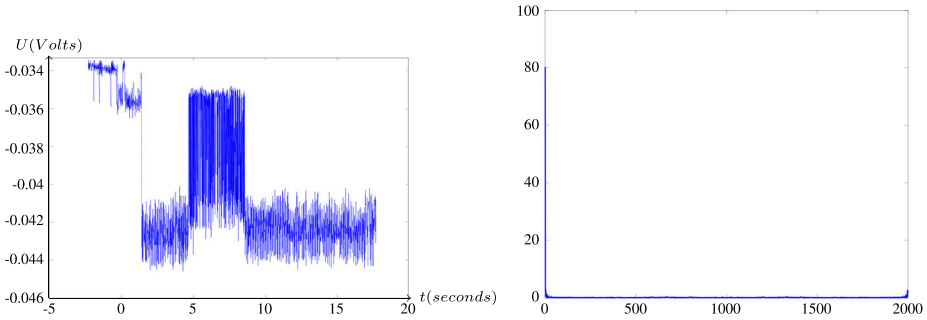


Figure 6. Experiment wiring: clamp on graphic processing unit (GPU) computer.



(a) Original signal measured by the oscilloscope (b) The frequency spectrum of the original signal of Fig. 7(a)

Figure 7. (a) Original signal measured by the oscilloscope and (b) corresponding frequency spectrum of the signal.

We have designed our own experimental protocol that measures precisely the energy consumption of the GPU, from raw data obtained by the amperometric clamps. The raw data are now processed by a digital oscilloscope, so that we can treat them as numerical data.

3.2. Signal processing

The intensity is categorized into two levels: (1) a level when the GPU is idle and (2) another when the GPU is busy. Unfortunately, as shown by Figure 7(a), the measured data are very noisy due to disturbances caused by the other components of the computer.

To retrieve these values, it is necessary to remove the noise. Then, we need to use a filtering technique. There are several filtering techniques to process digital signals. At first, we have examined three famous filters and chosen the best one from them for our purpose.

3.2.1. Ideal low-pass filter. The basic idea of the low-pass filter is based on the assumption that the noise oscillates much faster than the signal itself, which means the noise resides in higher frequen-

DOI: 10.1002/cpe

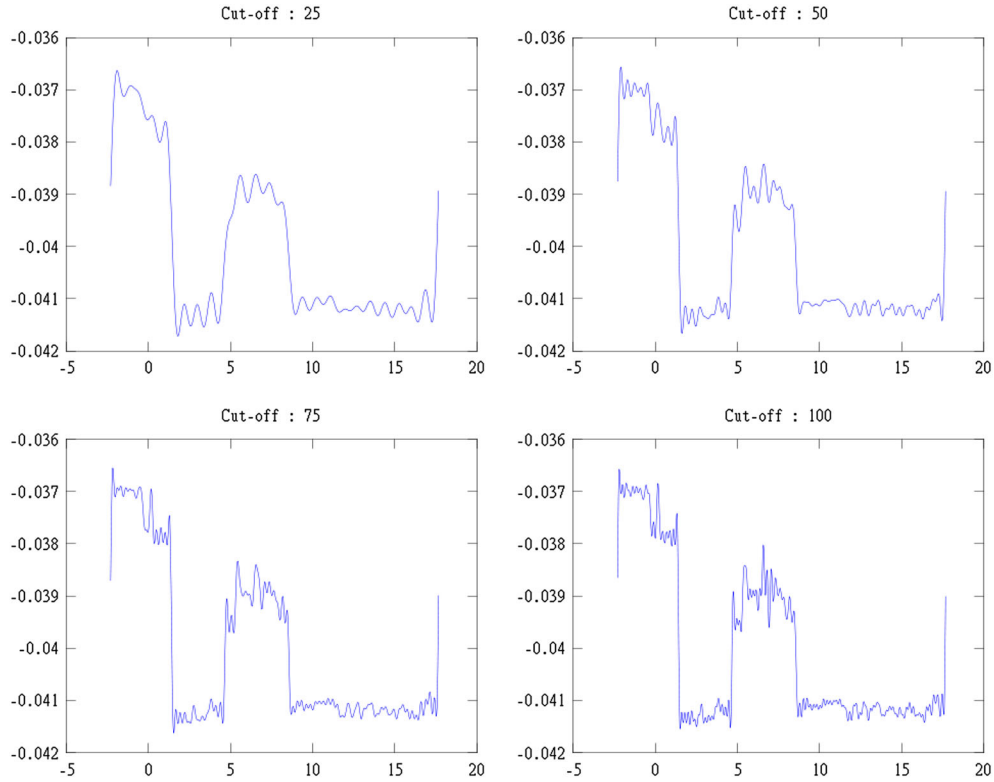


Figure 8. Low-pass filters with different cutoff frequencies.

cies. Then, we apply *fast Fourier transform* (FFT) to the signal and get the frequency spectrum. Figure 7(a) shows the original signal, and Figure 7(b) shows the frequency spectrum, respectively. Most amount of the noise is located in the high spectrum region; then, we cut these frequencies. By applying *inverse* FFT, we get a smoother signal by reducing noise, but it is little perturbed as a result of filtering. There is a tuning parameter to define which frequency will be cut. We showed results of processed signal with different cutoff frequencies in Figure 8.

3.2.2. Simple averaging filter. The other fundamental filtering technique is *averaging filter*, which use the fact that the noise causes fluctuations and they will be removed by replacing the value on each point by the average of several neighboring points. *Simple averaging filter* uses a uniform weight on the points. The computation is realized by a convolution product. There is a parameter to define the number of averaging points. We show results of proceeded signal with different number of the points in Figure 9.

3.2.3. Weighted averaging filter. This filter is an extension of simple averaging filter by introducing weights for averaging. The weight with a Gaussian distribution results in a combination of a simple averaging filter in the temporal domain and of a low-pass filter in the frequency domain. The standard deviation σ defines the distribution of a Gaussian function. We show results of proceeded signal with different σ in Figure 10.

Figure 11 summarizes the results of the three different filters. We see that *simple averaging filter* cannot remove noise completely without losing information. The low-pass filter and the Gaussian filter produce almost the same results, but the Gaussian is better in terms of accuracy and by preserving of intensity of the signal. Therefore, we have chosen the Gaussian filter to eliminate noise from the signal measured by the digital oscilloscope in our experimental protocol.

DOI: 10.1002/cpe

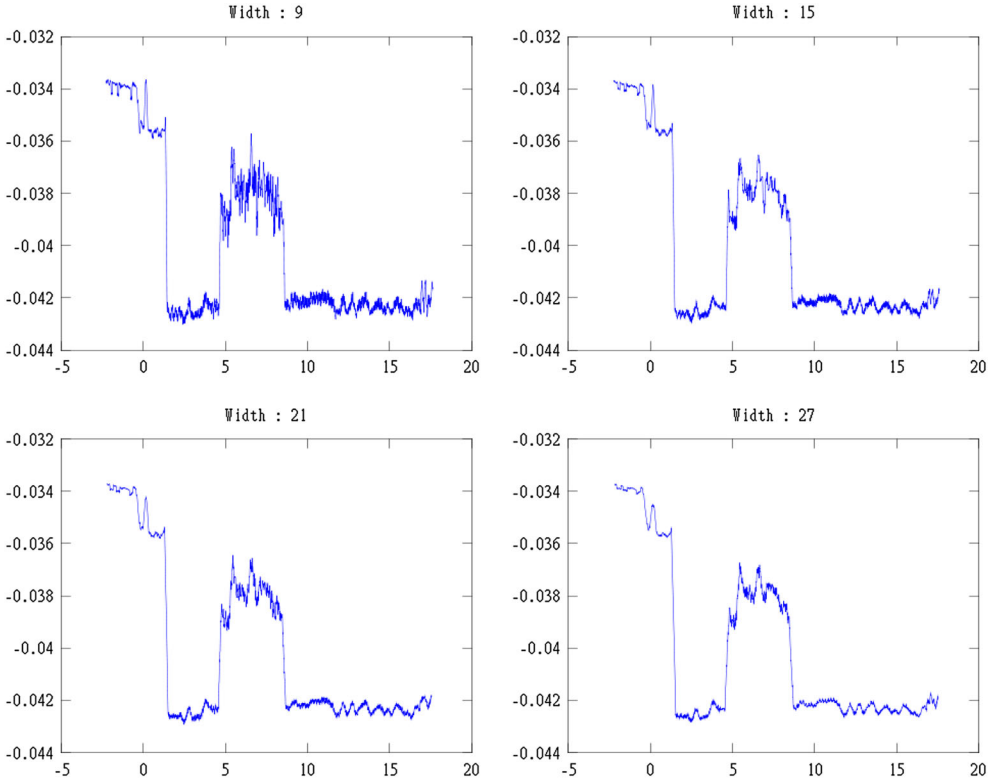


Figure 9. Simple averaging filters.

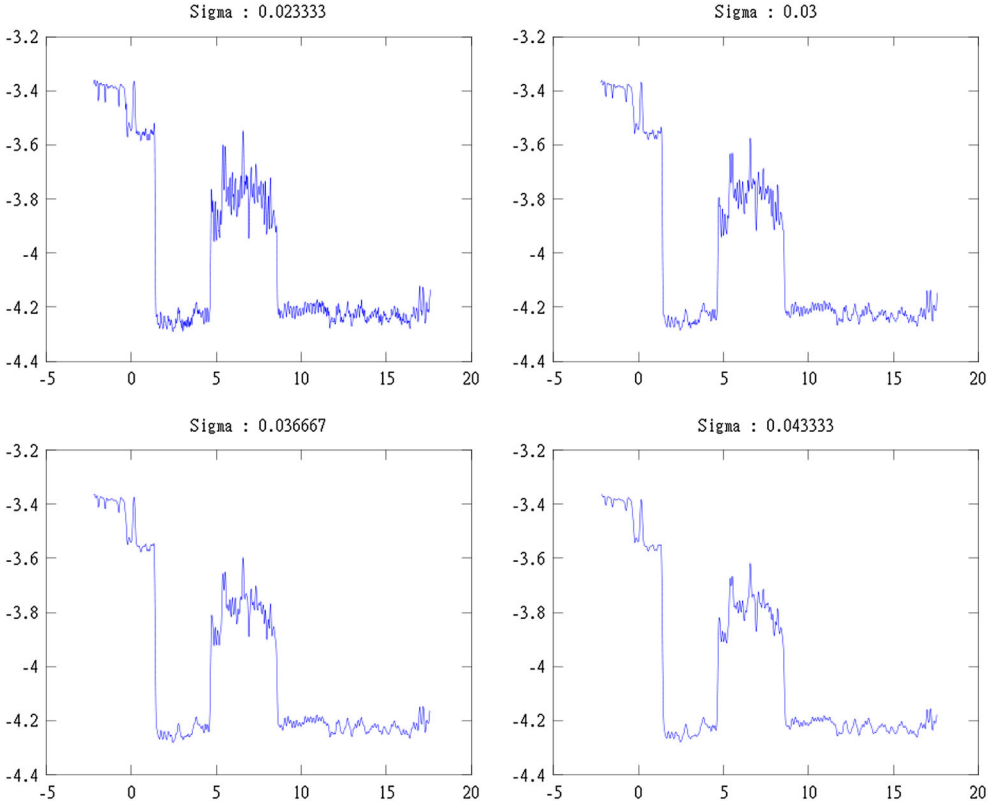


Figure 10. Gaussian filters.

DOI: 10.1002/cpe

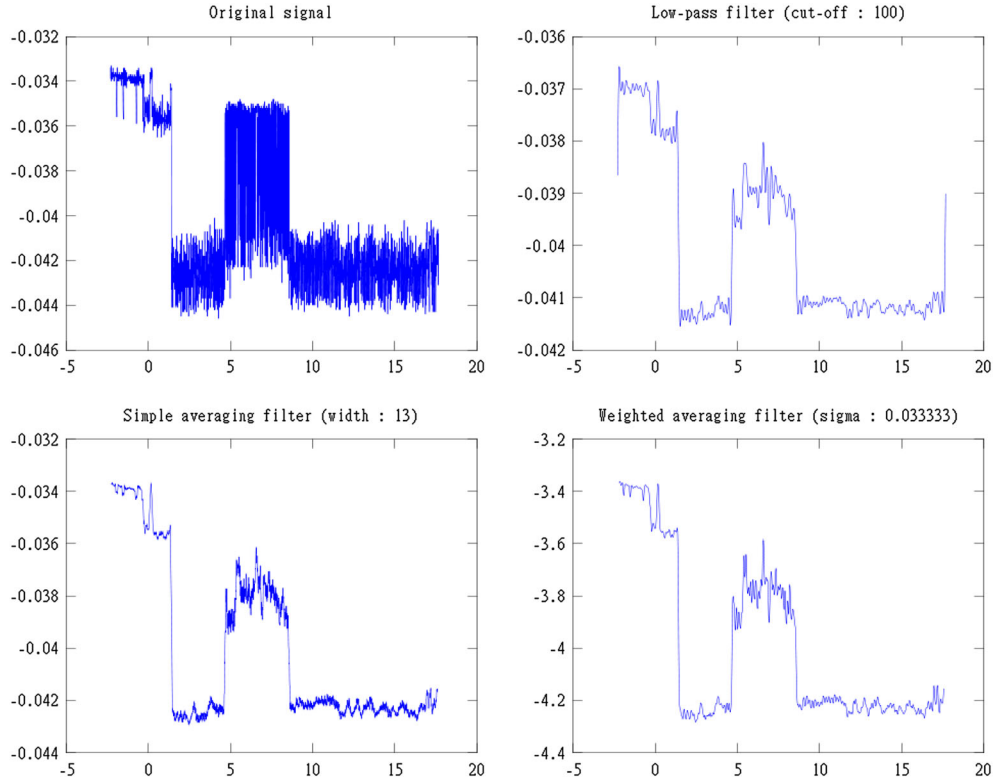


Figure 11. Summary of filters.

3.3. Acquisition process

To evaluate our results and validate our experimental protocol, we studied qualitatively the response of the GPU for the elementary operations such as memory allocation, addition (called as double precision addition of vectors (DAXPY)) and product of two vectors (element-wise product), and communication (memory copy) between the GPU and the CPU. Because our target application is numerical solution of real engineering problems, we deal with double precision arithmetic, which has been the standard in numerical simulations for three decades.

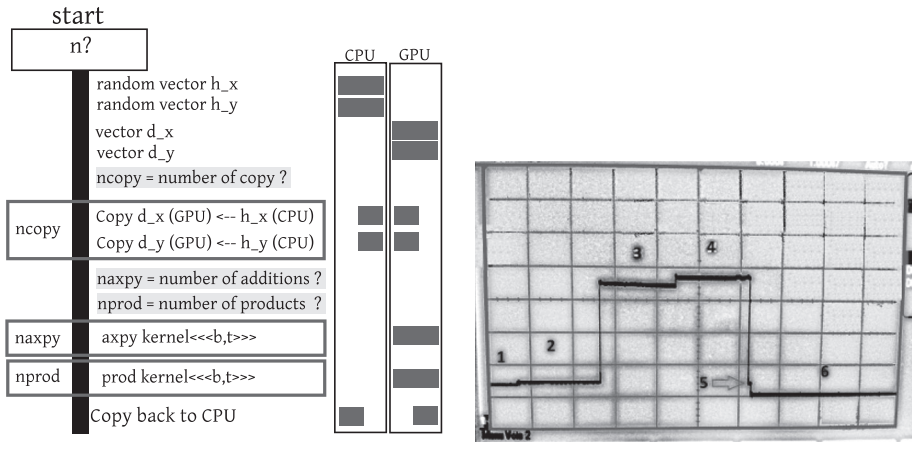
3.3.1. Procedure. A graph obtained by the digital oscilloscope allows us to follow the execution steps of the code, preparation in CPU, and launching GPU kernel corresponding to each type of basic operations. There are the following steps, which are also illustrated in Figure 12(a):

- launch the program by the given n, which define, the size of the vectors,
- generate random vectors in the main memory of the CPU,
- allocate working vectors in global memory in the GPU,
- enter a parameter: n_copy, the number of copy operation will be performed,
- copy data from CPU to GPU,
- enter parameters: n_axpy, the number of DAXPY, and n_prod, the number of products to be performed, and number of threads per block,
- execute the kernel of DAXPY n_axpy times and save the total elapsed time,
- execute the kernel product n_prod times and save the total elapsed time, and
- return the results of all computations from GPU to CPU.

The result on the screen of the oscilloscope shows clearly six different phases corresponding to the following states:

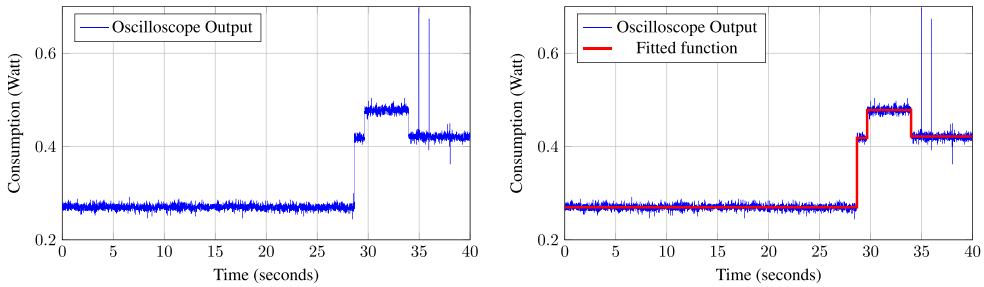
1 GPU is allocating the memory without touching the values.
2 GPU is copying data from CPU to the memory.

DOI: 10.1002/cpe



(a) Acquisition process (b) Screen of the oscilloscope

Figure 12. (a) Acquisition process and (b) screen of the oscilloscope. GPU, graphic processing unit.



(a) Energy consumption for the different phases of the execution

(b) Approximated function of the signal

Figure 13. (a) Energy consumption for the different phases of the execution and (b) approximated function of the signal.

3 GPU is preforming DAXPY.
4 GPU is performing the product.
5 GPU is copying back data to CPU and deallocating the memory.
6 GPU is back to standby.

This is verified by Figure 12(b) with human eye, but to get precise timing of changing phases, we need to proceed the further step.

3.4. Detection of the phases

Figure 13(a) shows an example of a measurement of the energy consumption of the execution obtained from the oscilloscope with the electrical power is in Watt or Joule over second. We can see four phases with some amount of noise in the figure. Each phase corresponds to each state: *sleep*, *sum*, *product*, and *copy*. In order to calculate the energy consumption and the elapsed time of each state, we needed to construct an algorithm that can detect each step precisely. We can assume that the energy consumption of each phase can be represented by a constant value and then an algorithm needs to identify the time when the relevant jump occurs and the values before and after the jump. We have constructed such an algorithm by the following way. The algorithm identifies the relevant jump by computing mean value of the current phase and verifies the 10-th forwarding value. If the forward value is too far from the current mean value, the algorithm concludes there is a relevant jump. This algorithm is improved by considering the variation of the standard deviation

DOI: 10.1002/cpe

of the values. Figure 13(b) shows an example of an approximate function with three relevant jumps computed by the algorithm. We also get information of the timing of the execution from this curve.

4. EXPERIMENTAL RESULTS

In this section, we report the experiments to evaluate and analyze the effectiveness and robustness of our experimental protocol. We used a workstation equipped with an Intel Core i7 920 (Intel, Santa Clara, CA, USA) running with 2.67 GHz, which has four physical cores and four logical cores and 6 GB main memory and one dedicated Nvidia GTX275 GPU with 895 MB memory. The GPU card has capability of double precision arithmetic and is driven by `CUDA` 4.0 software.

4.1. Elementary operations

Our aim is to establish a model that can predict the execution time and the energy consumption of the program from the number of elementary operations in the source code. For development of our methodology, we first discuss the dependency of the execution time and the energy consumption of elementary operations on the size of data set (vectors) for elementary operations, which are executed on a single GPU. For multiple GPU environment, because GPUs are independently controlled by the host hardware, our data on a single GPU are easily extrapolated to the case of multiple GPUs.

To evaluate the elementary operations, we use the program detailed in Figure 12(a) for different size of the vectors. With our current protocol, the acquisition and analysis of the measurements consume considerable time; therefore, we prepared a script that automates the process.

4.1.1. Data transfers. The cost of transferring data between CPU and GPU is not negligible. We have measured the time in seconds and the power in Watts for transferring double precision data. Table I shows the time and power with several sizes of the data. Because the complexity of data transfer is linear, we can apply linear prediction for both time and power of GPU execution. From the table, we can determine the prediction of time in second for size of vector x, by

$$7.0 \times 10^{-5} + 10^{-8}x \tag{1}$$

with 0.9922 coefficient of determination. The prediction of the electrical power in Watt is given by

$$2.0 \times 10^{-5} + 10^{-7}x \tag{2}$$

with 0.9998 coefficient of determination.

4.1.2. Addition of vectors (double precision addition of vectors). An operation of addition of two vectors of double precision with a scalar weight $y_i = y_i + \alpha x_i$ $(1 \leqslant i \leqslant n)$ is called a DAXPY, which belongs to basic linear algebra subprograms level 1. This operation is completely parallelized without any overhead of parallelization. We note that CUDA hardware proceeds the fused multiply and addition operation that allows to compute $y = y + \alpha x$ by a dedicated hardware as one operation.

Table I. Double precision data transfers from CPU to graphic processing unit.

Size	Time (s)	Power (W)
1000	0.00012219	0.000786827
2000	0.000149068	0.000964973
5000	0.000172517	0.001386241
10,000	0.000321641	0.002139488
20,000	0.000418555	0.003222079
2,000,000	0.030316534	0.20801877
5,000,000	0.061184495	0.536819129

DOI: 10.1002/cpe

Table II shows the GPU time in seconds and the electrical power in Watt for DAXPY. Because the complexity of this addition is linear, the linear predictions of GPU execution time and electrical power are appropriate. These predictions are listed in Table IV.

4.1.3. Element-wise product. Element-wise product computes $c_i = x_i y_i$ $(1 \leqslant i \leqslant n)$, which we call DXMY in this paper. This operation is also completely parallelized as DAXPY. However, it needs to access three arrays for the operation, which results in more memory-intensive operation than DAXPY. Table III shows the GPU time in seconds and the electrical power in Watt. We can see the time for each size is exactly the same as one of DAXPY, but consumed energy is larger than DAXPY due to much more memory access. Because the complexity of DXMY operation is linear, the linear predictions of the GPU execution time and electrical power are appropriate, which are listed in Table IV.

4.2. Evaluation of the execution time and energy consumption

Following the experimental results of data transfers, addition of vectors (DAXPY) and element wise product (DXMY), we propose a model of energy consumption.

Let us define the computation time $T_{compute}$ of arithmetic operations (DAXPY or DAXMY) by the unitary time $TU_{compute}$ multiplied by the size of the vectors n. Let $T_{CPU \leftrightarrow GPU}$ be the communications time. The execution time $T_{execute}$ becomes the sum of those three values.

Table II. Double precision addition of vectors (DAXPY) and element wise product (DXMY).

size	time (s)	power (W)
(a) Double precision addition of vectors (DAXPY)		
1,000	0.000005008	0.000305963
2,000	0.000006545	0.000390259
5,000	0.000006763	0.000408337
10,000	0.000006535	0.000420392
20,000	0.000009760	0.000571913
2,000,000	0.000456454	0.027676176
5,000,000	0.001142272	0.069889335
(b) Double precision element wise product (DXMY)		
1,000	0.00000512	0.000318909
2,000	0.000006673	0.000462524
5,000	0.000007001	0.000563243
10,000	0.000006603	0.000541402
20,000	0.000009795	0.000982404
2,000,000	0.000325738	0.046621519
5,000,000	0.001115328	0.160956136

Table III. Linear prediction of element-wise product.

	Equation	Coefficient of determination
Time (s)	$-6.668 \times 10^{-6} + 2.164 \times 10^{-10} x$	0.9857
Power (W)	$-1.417 \times 10^{-4} + 3.131 \times 10^{-8} x$	0.9856
*w*here x is the size of the vectors.		

Table IV. Linear prediction of double precision addition of vectors.

	Equation	Coefficient of determination
Time (s)	$4.905 \times 10^{-6} + 2.272 \times 10^{-10} x$	0.999
Power (W)	$2.682 \times 10^{-4} + 1.389 \times 10^{-8} x$	0.999
*w*here x is the size of the vectors.		

DOI: 10.1002/cpe

$$TU_{compute} = TU_{\mathrm{DAXPY}} \text{ or } TU_{\mathrm{DXMY}} \quad T_{compute} = n \times TU_{compute}, \tag{3}$$

$$T_{CPU \leftrightarrow GPU} = n * TU_{CPU \leftrightarrow GPU} \quad T_{execute} = T_{CPU} + T_{CPU \leftrightarrow GPU} + T_{compute}. \tag{4}$$

To define the whole energy consumption, in addition to the execution time, we need to consider the energy consumption corresponding to other phases,

- P_{idle}, power at rest: 47.4 W,
- P_{active}, power when the card is activated (memory allocated but not yet used): 58.43 W,
- P_{pause}, power during a pause in the calculation: 60.77 W, and
- P_{end}, power at the end of the program, before the process is completed: 59.14 W.

These values correspond to the states in Section 3.4 and Figure 13(b).

The energy consumption is calculated by the formula:

$$\begin{aligned} Energy = {} & P_{idle} \times t_{idle} + P_{active} \times t_{active} \\ & + P_{pause} \times t_{pause} + P_{end} \times t_{end} \\ & + P_{copy}(n) \times T_{copy} * n + P_{compute}(n) \times T_{compute} * n \end{aligned} \tag{5}$$

with t_{idle}, the time during the program runs without using the GPU card; t_{active}, the time during the memory will be allocated without being used; t_{pause}, the time during the GPU will not be used in the middle of the calculations, for example, computation is performed on the CPU; and t_{end}, the time during the main program continues to run and the GPU is not used.

5. EVALUATION

We deal with the most fundamental operations to solve a linear system with sparse matrix that is obtained from a discretization of PDEs. They are inner product (double precision dot product (DDOT)) and multiplication of sparse matrix to vector (SpMV), when we use Krylov subspace methods [15–17], which is the most efficient solver for three-dimensional large-scale problems. Here, we deal with conjugate gradient (CG) method, a member of Krylov subspace methods, for the solution of a linear equation with a symmetric positive definite matrix.

5.1. Linear algebra operations

DDOT computes a dot product of two vectors and belongs to basic linear algebra subprograms level 1 the same as DAXPY. However, implementation of DDOT requires some efforts to achieve good performance in parallel environment. The operation $\sum_i x_i y_i$ is distributed over several processors by decomposing indices into a union of blocks, and finally it is necessary to sum up scalar values from all processors. This operation implies a kind of synchronization between processors. Unfortunately, there is no dedicated hardware for synchronization between CUDA ALUs; the host CPU needs to participate to get the final value of the inner product.

SpMV performs a multiplication of the sparse matrix to the dense vector, which is the most important operation in Krylov subspace methods. To proceed SpMV, we need to select data structure for the sparse matrix. Compressed sparse row [15] format is widely used because of minimal memory usage [18] and simplicity of the implementation.

The sparse matrix A is stored in two integer arrays and one double precision array. Let n denote the size of row and nnz be the number of nonzeros. An integer array IA with size $n + 1$ stores starting with index of column data for each row i, where each value between $IA(i)$ and $IA(i+1)-1$ points index of JA and AA. The last value of $IA(n + 1)$ is set as $nnz + 1$ for the consistency. An integer array JA with size nnz stores column index, and a double precision array AA with size nnz stores the nonzero values. A value of $A_{i,j}$ is stored in $AA(k)$ whose column index j is accessed through k as $j = JA(k)$ for $IA(i) \leqslant k < IA(j)$. By using compressed sparse row format, SpMV operation is defined as

$$y_i = y_i + \sum_{IA(i) \leqslant k < IA(i+1)} AA(k) * x_{JA(k)} \quad \text{for } 1 \leqslant i \leqslant n.$$

DOI: 10.1002/cpe

5.2. *Linear algebra library for graphic processing unit*

We will compare the operations DDOT and SpMV and over all performance of CG method using two linear algebra libraries. One of the libraries is Cusp [19] that has GPU computation capability with C++ object-oriented interface and is provided as an open source. The other is our own library, *Alinea* [14, 20–23], which is optimized for GPU computation.

5.3. *Dot product*

Table V shows the kernel time in seconds, the electrical power in Watt, and the energy consumption in Joule (Watt × second) by the GPU. Figures 14(a) and 14(b), respectively, show graphical presentation of data of execution time in seconds and energy consumption in Joule, given in Table V. Here, the kernel is performed by both Alinea [24] and Cusp. The energy consumption in Joule is calculated by the multiplication of energy and time of the operation plus the fundamental energy consumption by the GPU. We can see that Alinea is faster than Cusp and consumed energy of Alinea is less than Cusp. Because the complexity of DDOT is linear, the linear prediction of the GPU execution time and energy consumption is appropriate and is given in Table VI . We can see the good performance of Alinea on energy consumption compared to Cusp, and we can notice that the coefficient of linear dependency in the prediction formula of Alinea is one-third of the one of Cusp.

5.4. *Tested sparse matrix*

Several researches have already performed experiments of SpMV operation on GPU [25–29], using the matrix markets data that contain various data in wide range of applications, but with somewhat

Table V. Double precision dot product (DDOT).

	Alinea			Cusp		
n	time (s)	P (W)	E (J)	time (s)	P (W)	E (J)
89,056	0.004	54.841	0.201	0.010	55.886	0.584
101,168	0.004	54.672	0.227	0.012	55.717	0.659
181,888	0.007	54.747	0.404	0.021	55.792	1.174
270,336	0.011	55.032	0.590	0.031	56.053	1.714
296,208	0.012	55.427	0.645	0.033	56.457	1.875
416,176	0.016	55.644	0.892	0.046	56.693	2.594
450,528	0.018	55.559	0.979	0.050	56.601	2.847
544,563	0.021	55.587	1.178	0.060	56.629	3.424
606,816	0.024	55.717	1.314	0.067	56.694	3.815
650,848	0.025	55.604	1.418	0.073	56.763	4.130
848,256	0.033	55.865	1.842	0.094	56.884	5.351
1,213,488	0.049	55.884	2.718	0.139	56.927	7.901
1,325,848	0.053	55.863	2.968	0.152	56.906	8.628

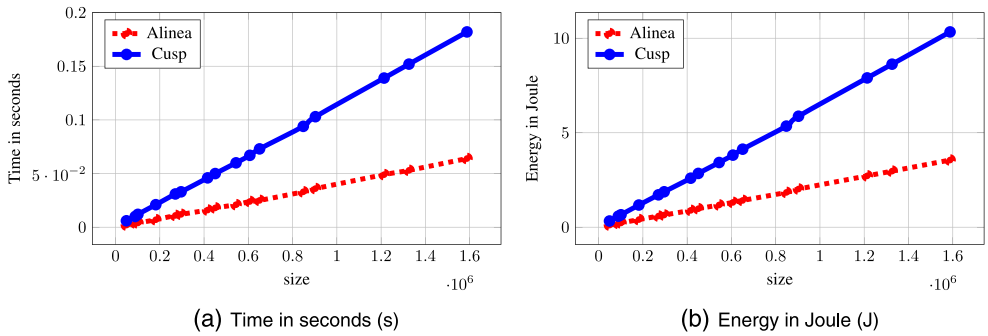


Figure 14. Double precision dot product. (a) Time in seconds (s) and (b) energy in Joule (J).

DOI: 10.1002/cpe

Table VI. Linear prediction of dot product.

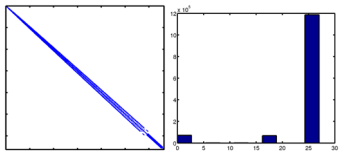
	Equation	Coefficient of determination
Alinea time (s)	$-2.216 \times 10^{-4} + 4.002 \times 10^{-8}x$	0.9996
Cusp time (s)	$-6.325 \times 10^{-4} + 1.142 \times 10^{-7}x$	0.9996
Alinea energy (J)	$-1.897 \times 10^{-2} + 2.241 \times 10^{-6}x$	0.9996
Cusp energy (J)	$-5.477 \times 10^{-2} + 6.514 \times 10^{-6}x$	0.9996

*w*here x is the size of the vectors.

Table VII. Sketches of the matrices.

n	nnz	nnz/n	$\sigma(nnz/n)$
89,056	2,010,320	22.573	8.805
101,168	2,310,912	22.752	8.666
181,888	4,268,192	23.466	8.051
270,336	6,456,960	23.885	7.638
296,208	7,101,472	23.975	7.544
416,176	10,107,680	24.287	7.199
450,528	10,972,752	24.355	7.119
544,563	13,348,267	24.512	6.932
606,816	14,926,000	24.597	6.826
650,848	16,044,032	24.651	6.758
848,256	21,073,920	24.844	6.504
1,213,488	30,434,592	25.080	6.171
1,325,848	33,321,792	25.132	6.094

Table VIII. Example pattern of matrices.

gravi 1325848 h=1,325,848 nz = 33,321,792 density = 0.002 bandwidth = 33389 max row = 27 nz/h = 25.132 nz/h stddev = 6.094	

moderate sizes. However, our target is to understand efficiency of GPU computation for realistic engineering and scientific problems that are described by PDEs.

We consider large matrices arising from the finite element discretization [30, 31] from a realistic scientific problem. The properties of matrices with different mesh sizes are summarized in Table VII, and examples of pattern are given in Table VIII. Numbers, n, nnz, nnz/n, and $\sigma(nnz/n)$ correspond to the size of the matrix, the number of nonzero elements, the mean density of nonzero entries, and the standard deviation of nnz/n, respectively. In Table VIII, the second column represents the sparse matrix pattern, and the third column represents the distribution of the nonzero elements. All those matrices have the same structure of nonzero pattern. The sizes of generated matrices vary from 49,248 to 1,325,848 rows, and numbers of nonzero values vary from 2,010,320 to 33,321,792.

5.5. Sparse matrix vector multiplication

Table IX shows the kernel time in seconds, the electrical power in Watt, and the energy consumption in Joule. Both results by Alinea and Cusp for SpMV with double precision are listed. Figures 15(a) and 15(b), respectively, show graphical presentation of data of execution time in seconds and energy consumption in Joule, given in Table IX. We can see that Alinea achieves better results than Cusp

DOI: 10.1002/cpe

Table IX. Double precision SpMV with CSR format.

	Alinea			Cusp		
n	time (s)	P (W)	E (J)	time (s)	P (W)	E (J)
89,056	0.001	142.526	0.120	0.001	150.716	0.161
101,168	0.001	144.229	0.136	0.001	151.423	0.183
181,888	0.002	147.086	0.242	0.002	153.193	0.330
270,336	0.002	148.453	0.363	0.003	154.243	0.493
296,208	0.003	148.591	0.401	0.003	154.756	0.541
416,176	0.004	148.866	0.566	0.005	155.473	0.763
450,528	0.004	150.368	0.617	0.005	155.742	0.827
544,563	0.005	151.245	0.754	0.006	156.787	1.007
606,816	0.006	150.650	0.837	0.007	156.451	1.118
650,848	0.006	149.420	0.891	0.008	156.229	1.198
848,256	0.008	149.289	1.176	0.010	155.357	1.552
1,213,488	0.011	149.354	1.676	0.014	155.280	2.219
1,325,848	0.012	145.663	1.807	0.016	155.419	2.427

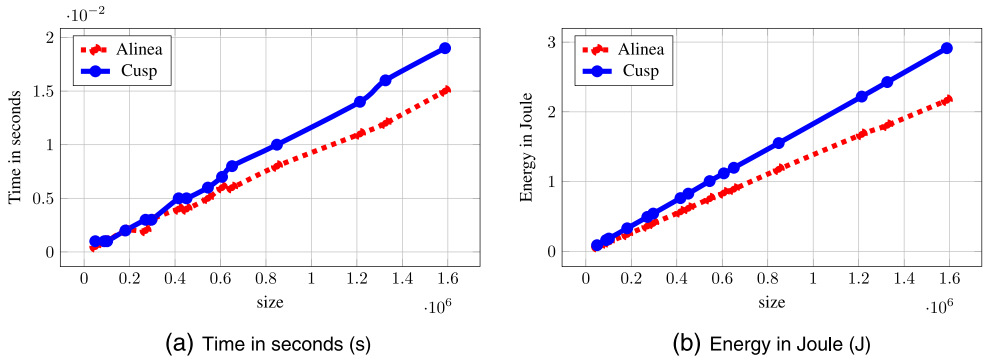


Figure 15. Double precision sparse matrix–vector multiplication (compressed sparse row format). (a) Time in seconds (s) and (b) energy in Joule (J).

Table X. Linear prediction of sparse matrix–vector multiplication.

	Equation	Coefficient of determination
Alinea time (s)	$-3.552 \times 10^{-5} + 9.286 \times 10^{-9}x$	0.9996
Cusp time (s)	$1.165 \times 10^{-5} + 1.177 \times 10^{-8}x$	1.000
Alinea energy (J)	$-1.223 \times 10^{-3} + 1.371 \times 10^{-6}x$	0.9999
Cusp energy (J)	$-6.667 \times 10^{-4} + 1.834 \times 10^{-6}x$	1.000

*w*here x is the size of the problem.

in terms of execution time and energy consumption. Especially, Alinea has big advantage in energy consumption, 25% less energy for the largest size.

Complexity of SpMV for the matrix studied here, for example, Poisson equation, is linear because nonzero of each row is almost constant for all sizes of the matrices. The linear prediction of the GPU execution time and energy consumption for double precision SpMV are given in Table X.

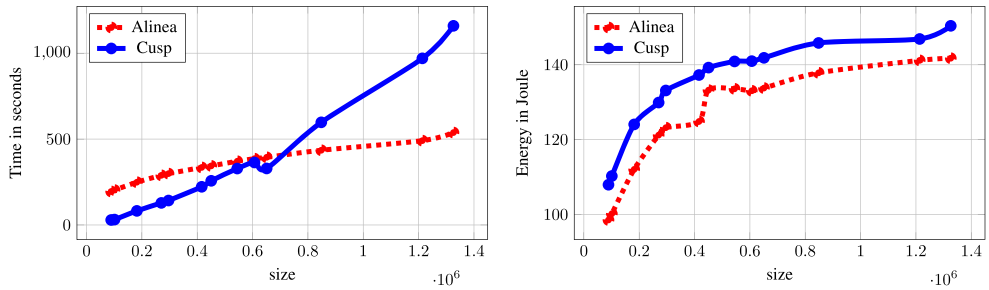
5.6. Conjugate gradient method

We performed CG solver with setting the residual tolerance $\varepsilon = 10^{-6}$ by both Alinea and Cusp. Table XI shows number of iterations, time in seconds, the electrical power in Watt, and the energy consumption by the GPU in Joule. Figures 16(a) and 16(b), respectively, show graphical presentation of data of execution time in seconds and energy consumption in Joule, given in Table XI.

DOI: 10.1002/cpe

Table XI. Double precision conjugate gradient with compressed sparse row format.

		Alinea			Cusp		
n	Number of iterations	Time (s)	P (W)	E (J)	Time (s)	P (W)	E (J)
89,056	195	0.295	98.528	29.066	0.421	107.948	45.414
101,168	203	0.313	100.201	31.363	0.471	110.271	51.888
181,888	249	0.732	112.051	82.021	0.833	124.062	103.381
270,336	289	1.060	121.207	128.480	1.297	129.905	168.455
296,208	299	1.155	123.190	142.284	1.436	133.129	191.199
416,176	335	1.780	124.854	222.240	2.148	137.259	294.833
450,528	344	1.930	133.225	257.124	2.360	139.240	328.605
544,563	371	2.460	133.564	328.568	3.003	140.889	423.019
606,816	386	2.735	133.052	363.898	3.446	140.968	485.776
650,848	394	2.460	133.772	329.078	3.741	141.852	530.609
848,256	435	4.336	137.843	597.618	5.250	145.828	765.595
1,213,488	491	6.872	141.169	970.115	8.396	146.889	1,233.248
1,325,848	538	8.180	141.747	1,159.492	9.945	150.403	1,495.759



(a) Time in seconds (s) (b) Energy in Joule (J)

Figure 16. Double precision conjugate gradient (compressed sparse row format). (a) Time in seconds (s) and (b) energy in Joule (J).

Table XII. Predictions of computational time and energy consumption of conjugate gradient solver.

	Equation
Alinea time (s)	$1.408 \times 10^{-2} + 4.287 \times 10^{-8} x^{1.35}$
Cusp time (s)	$8.060 \times 10^{-2} + 5.231 \times 10^{-8} x^{1.35}$
Alinea energy (J)	$-1.881 \times 10^{1} + 6.156 \times 10^{-6} x^{1.35}$
Cusp energy (J)	$-1.1773 \times 10^{-1} + 7.890 \times 10^{-6} x^{1.35}$

*w*here x is the size of the problem.

From Table XI, we can derive dependency of number of iteration on the size of problem, number of iterations $= n^{0.35}$. The coefficient 0.35 has good agreement to the theoretical estimation of the condition number of the Poisson equation in three dimensions. This value leads to the complexity of the whole CG solver as $n^{1.35}$. Table XII gives the linear prediction of the GPU execution time and energy consumption for double precision CG.

We can conclude that implementation of Alinea is much efficient than Cusp in the sense of energy consumption whereas both implementations with Alinea and Cusp follow the same numerical characteristic with iteration number of CG. Computational time of Alinea is 18% less than Cusp, and consumed energy to get approximate solution is 22% less for the largest size of our experiment. The coefficient of prediction of energy consumption also shows better efficiency of Aliena than Cusp. Note that when the mesh size is more fine, GPU is more efficient compared to CPU. Unfortunately, when the mesh is too fine, the corresponding assembled matrix becomes too large for GPU memory. In this case, domain decomposition method [32–35] based on iterative methods is an issue. Authors

DOI: 10.1002/cpe

present how Schwarz domain decomposition method is designed efficiently on GPU and clearly proved the interest of optimized Schwarz methods on a cluster of GPUs [36]. Papadrakakis *et al* proposed in [37] an implementation of finite element tearing and interconnecting (FETI) domain decomposition methods in hybrid CPU–GPU architectures.

6. CONCLUSION

In this paper, we have established an experimental protocol for measuring the power consumption of GPU during computation. Due to the lack of monitoring sensor on GPU for state and energy consumption, we used amperometric clamps to directly measure the current provided to GPU. Precise phase analysis of the time-dependent current of GPU combined with efficient noise reduction technique provided accurate timing of GPU states and intensity of the current of each state.

By applying our experimental protocol to fundamental and basic GPU operations, DAXPY and element-wise product, which are completely parallelized within the GPU, and to memory copy operations between CPU and GPU, we have determined the energy consumption of the GPU for idling state, activated state, pausing state, and finalizing state of the GPU. It was interesting to discover that GPU consumes certain amount of energy even though during its idling state, by maintaining the state of the GPU global memory. In addition, the model for the prediction of the energy consumption proposed in this paper, which is based on arithmetic complexity, has shown good agreement with the measured energy consumption.

Comparison of Alinea, a library developed by the authors, and Cusp, an open-source library, has shown the importance of the optimization for energy consumption; Alinea outperforms Cusp by 20% less energy consumption and by 10% less computation time for the samer calculations.

ACKNOWLEDGEMENTS

The authors acknowledge the CUDA Research Center at Ecole Centrale Paris (France) for its support and for providing the computing facilities. The authors also acknowledge Alban Desmaison, Jean-Christophe Léchenet, François Mayer, Thomas Saint-Paul, Haifa Ben Salem, and Thomas Zhu for their great help in the numerical experiments.

REFERENCES

1. Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience* 2003; **15**(9):803–820.
2. Li N, Suchomel B, Osei-Kuffuor D, Li R, Saad Y. ITSOL, 2010. (Available from: www-users.cs.umn.edu/~saad/software/ITSOL/index.html) (Accessed on 22 September 2015).
3. Balay S, Adams MF, Brown J, Brune P, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zhang H. PETSc users manual. *Technical Report ANL-95/11 - Revision 3.4*, Argonne National Laboratory: Argonne, IL, USA, 2013.
4. Liu W, Du Z, Hiao Y, David AB, Xu C. A waterfall model to achieve energy efficient tasks mapping for large scale GPU clusters. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum,* Shanghai, China*, 16-20 may 2011*, IEEE, 2011; 82 –92.
5. Huo H, Sheng C, Hu X, Wu B. An energy efficient task scheduling scheme for heterogeneous GPU-enhanced clusters. *Proceedings of the International Conference on Systems and Informatics,* Yantai, China *May 19-20, 2012*, IEEE, 2012; 623 –627.
6. Juan MC, Gines DG, Jose MG. Energy efficiency analysis of GPUs. *Proceedings of IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum,* Shanghai, China*, 21-25 May 2012*, 2012; 1014 –1022.
7. Subramaniam B, Saunders W, Scogland T, Feng W. Trends in energy-efficient computing: a perspective from the green500. *Proceedings of the 4th International Green Computing Conference,* Arlington, VA, USA*, June, 27-29 2013*, IEEE, 2013; 1–8.
8. Nvidia Corporation. CUDA programming guide (4.0), 2011. (Available from: http://developer.nvidia.com/cuda-toolkit-40) (Accessed on 22 September 2015).
9. PCI-SIG. PCI express 3.0 frequently asked questions PCI-SIG, 2012. (Available from: http:http://kavi.pcisig.com/news_room/faqs/pcie3.0_faq/PCI-SIG_PCIe_3_0_FAQ_Final_07102012.pdf) (Accessed on 22 September 2015).
10. PCI-SIG. PCI express architecture frequently asked questions, 2005. (Available from: http://kavi.pcisig.com/news_room/faqs/PCI_Express_FAQ.pdf) (Accessed on 22 September 2015).

DOI: 10.1002/cpe

11. Nvidia Corporation. CUDA best practices guide (4.0), 2011. (Available from: http://developer.nvidia.com/cuda-toolkit-40).
12. Guo P, Wang L. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. *Proceedings of the International Conference on Computational and Information Sciences, Chengdu, China, 17-19 Dec. 2010*, Chengdu, China, 2010; 1154–1157.
13. Davidson A, Zhang Y, Owens JD. An auto-tuned method for solving large tridiagonal systems on the GPU. *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, Anchorage, AK, 16-20 May 2011*, IEEE Computer Society, Anchorage, AK, USA, 2011; 956–965.
14. Cheik Ahamed AK, Magoulès F. Fast sparse matrix-vector multiplication on graphics processing unit for finite element analysis. *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications , Liverpool, UK, June 25–27, 2012*, IEEE Computer Society, Liverpool, UK, 2012; 1307–1314.
15. Saad Y. *Iterative Methods for Sparse Linear Systems* (2nd edn). Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2003.
16. Anzt H, Heuveline V, Rocker B. Mixed precision iterative refinement methods for linear systems:convergence analysis based on Krylov subspace methods. In *Applied Parallel and Scientific Computing*, Vol. 7134, Jnasson K (ed.). Springer Berlin Heidelberg, 2012; 237–247.
17. Barraza CIA. Solución Paralela en un GPU para la Ecuación de Advección-Difusión-Reacción mediante Diferencias Finitas Explícitas e Implícitas. *Difusión cientifica del Área de Ingeniería y Tecnologías (DIFU100ci), Universidad Autónoma de Zacatecas* 2013; **7**(2):33–40.
18. Djinevski L, Arsenovski S, Ristov S, Gusev M. Optimal configuration of GPU cache memory to maximize the performance, 2013.
19. Bell Nathan, Garland Michael. Cusp: generic parallel algorithms for sparse matrix and graph computations, 2012. (Available from: http://cusp-library.googlecode.com) (Accessed on 22 September 2015).
20. Cheik Ahamed AK, Magoulès F. Iterative methods for sparse linear systems on graphics processing unit. *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications,* Liverpool, UK*, June 25-27, 2012*, IEEE Computer Society, 2012; 836–842.
21. Cheik Ahamed AK, Magoulès F. Iterative Krylov methods for gravity problems on graphics processing unit. *Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, Kingston,* London, UK*, September 2–4, 2013*, IEEE Computer Society, 2013; 16–20.
22. Cheik Ahamed AK, Magoulès F. Iterative Krylov methods for acoustic problems on graphics processing unit. *Proceedings of the 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science,* Xianning, China*, November 24-27, 2014*, IEEE Computer Society, 2014; 19–23.
23. Magoulès F, Cheik Ahamed AK, Putanowicz R. Auto-tuned Krylov methods on cluster of graphics processing unit. *International Journal of Computer Mathematics* 2015; **92**(6):1222–1250.
24. Magoulès F, Ahamed AKC. Alinea: an advanced linear algebra library for massively parallel computations on graphics processing units. *International Journal of High Performance Computing Applications* 2015; **29**(3):284–310.
25. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors, Portland, OR, USA 14-20 nov. 2009. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09. ACM, 2009; 18:1–18:11.
26. Zhuowei W, Xianbin X, Wuqing Z, Yuping Z, Shuibing H. Optimizing sparse matrix-vector multiplication on CUDA. *Proceedings of the 2nd International Conference on Education Technology and Computer,* Shanghai, China*, 22-24 June 2010*, Vol. 4, IEEE, 2010; 109–113.
27. Tao Y, Deng Y, Mu S, Zhang Z, Zhu M, Xiao L, Ruan L. GPU accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication. *Concurrency and Computation: Practice and Experience* 2015; **27**(14):3771–3789.
28. Zimon J, Zoworka M. Implementation of selected numerical algorithms for solving sparse matrixes using CUDA technology. In *2013 International Symposium on Electrodynamic and Mechatronic System (SELM),* Opole-Zawiercie, Poland *15-18 May 2013*. IEEE, 2013; 77–78.
29. Djinevski L, Arsenovski S, Ristov S, Gusev M. Performance drawbacks for matrix multiplication using set associative cache in GPU devices. *2013 36th International Convention on Information & Communication Technology Electronics & Microelectronics (Mipro),* Opatija, Croatia *20-24 May 2013*, IEEE, 2013; 193–198.
30. Cecka C, Lew AJ, Darve E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 2011; **85**(5):640–669.
31. Chen D, Jiang H, Zhang Z, Pan R. A data structure for finite element method. *Transactions of China Electrotechnical Society* 2015; **30**(1):1–7.
32. Quarteroni A, Valli A. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press: Oxford, UK, 1999.
33. Maday Y, Magoulès F. Absorbing interface conditions for domain decomposition methods: a general presentation. *Computer Methods in Applied Mechanics and Engineering* 2006; **195**(29–32):3880–3900.
34. Magoulès F, Roux FX. Lagrangian formulation of domain decomposition methods: a unified theory. *Applied Mathematical Modelling* 2006; **30**(7):593–615.
35. Toselli A, Widlund O. *Domain Decomposition Methods*, Computational Mathematics, vol. 34. Springer-Verlag: Berlin Heidelberg, 2004.

DOI: 10.1002/cpe

36. Magoulès F, Cheik Ahamed AK, Putanowicz R. Optimized Schwarz method without overlap for the gravitational potential equation on cluster of graphics processing unit. *International Journal of Computer Mathematics* 2015: 1–26.
37. Papadrakakis M, Stavroulakis G, Karatarakis A. A new era in scientific computing: domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering* 2011; **200**(13-16): 1490–1508.

DOI: 10.1002/cpe