Evaluation of Two Parallel Finite Element Implementations of the Time-Dependent Advection Diffusion Problem: GPU versus Cluster Considering Time and Energy Consumption

Alberto F. De Souza[1], Lucas Veronese[1], Leonardo M. Lima[2], Claudine Badue[1] and Lucia Catabriga[1]

[1] Departamento de Informática, Universidade Federal do Espírito Santo, Vitória, Brazil
[2] Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo, Vitória, Brazil

Abstract. We analyze two parallel finite element implementations of the 2D time-dependent advection diffusion problem, one for multi-core clusters and one for CUDA-enabled GPUs, and compare their performances in terms of time and energy consumption. The parallel CUDA-enabled GPU implementation was derived from the multi-core cluster version. Our experimental results show that a desktop machine with a single CUDA-enabled GPU can achieve performance higher than a 24-machine (96 cores) cluster in this class of finite element problems. Also, the CUDA-enabled GPU implementation consumes less than one twentieth of the energy (Joules) consumed by the multi-core cluster implementation while solving a whole instance of the finite element problem.

1 Introduction

The advances of numerical modeling in the past decades have allowed scientists to solve problems of increasing complexity. Frequently, these problems require the solution of very large systems of equations at each time step and/or iteration. Because of that, a great effort has been made on the development of more efficient and optimized solution algorithms. But, along the past few decades, the underlying hardware for running these algorithms has changed significantly. A recent important development was the advent of the Compute Unified Device Architecture (CUDA) [12].

CUDA is a new Graphics Processing Unit (GPU) architecture that allows general purpose parallel programming through a small extension of the C programming language. The Single Instruction Multiple Thread (SIMT [12]—it is similar to SIMD, but more flexible on the use of resources) architecture of CUDA-enabled GPUs allows the implementation of scalable massively multithreaded general purpose C+CUDA code. Currently, CUDA-enabled GPUs possess arrays of hundreds of cores (called stream processors) and peak performance surpassing 1 Tflop/s. More than 200 million CUDA-enabled GPUs have been sold [10], which makes it the most successful high performance parallel computing platform in computing history and, perhaps, up to this point in time, one of the most disruptive computing technologies of this century—many relevant programs have been ported to C+CUDA and run orders of magnitude faster in CUDA-enabled GPUs than in multi-core CPUs.

In this paper, we analyze two parallel finite element implementations of the 2D time-dependent advection diffusion problem: one for multi-core clusters and one for CUDA-enabled GPUs [12]. We also compare their performances in terms of time and energy consumption.

The finite element method is one of the most used numerical techniques for finding approximated solutions of partial differential equations (PDE). In this method, the solution approach is based either on rendering the PDE into an approximating system of ordinary differential equations, which are then numerically integrated using standard techniques, such as the Euler's method [6].

The finite element formulation requires the solution of linear systems of equations involving millions of unknowns that are usually solved by Krylov space iterative update techniques [13], from which the most used is the Generalized Minimum Residual method (GMRES). One of the most time consuming operations of this solution strategy is the matrix-vector product, which can be computed on data stored according to global and local schemes. The most well known global scheme is the compressed storage row (CSR) [13], while the most well known local schemes are the element-by-element (EBE) and edge-based data structure (EDS) [2, 4]. The code for CSR is easily parallelized in different computer architectures. This type of implementation is often preferred to local schemes—matrix-vector products computed on EBE or EDS can be memory intensive, needing more operations than on CSR. However, particularly for large-scale nonlinear problems, EBE and EDS schemes have been very successful because they handle large sparse matrices in a simple and straightforward manner.

In this work, we consider the parallel finite element formulation of the 2D time-dependent advection diffusion equation. To solve the system of ordinary differential equations that results from the finite element formulation, we employ the well known implicit predictor/multicorrector scheme [15]. The sparse linear system of each time-step (stored in a Compressed Storage Row (CSR) scheme in both implementations) is solved by the GMRES method.

We implemented one code for multi-core clusters and, from that, one code for CUDA-enabled GPUs, and run them in a 24-machine (96 cores) cluster and in a 4-GPU desktop machine. Both implementations were written in C and use the MPI library for inter-core communication. Our simulations show that a desktop computer with a single GPU can outperform a 24-machine (96 cores) cluster of the same generation and that a 4-GPU desktop can offer more than twice the cluster performance. Also, with four GPUs, the CUDA-enabled implementation consumes less than one twentieth of the energy (Joules) consumed by the multi-core cluster implementation while solving a whole instance of the finite element problem. These results show that, currently, considering the benefits of shorter executing times, smaller energy consumption, smaller dimensions and maintenance costs, Multi-GPU desktop machines are better high performance computing platforms than small clusters without GPUs, even though they are somewhat harder to program.

2 Governing Equations and Finite Element Formulation

Let us consider the following time-dependent boundary-value problem defined in a domain $\Omega \in \Re^2$ with boundary Γ:

$$\frac{\partial u}{\partial t} + \boldsymbol{\beta}.\nabla u - \nabla.(\boldsymbol{\kappa}\nabla u) = f \quad \text{(time-dependent advection-diffusion equation)} \quad (1)$$

$$u = g \quad \text{on} \quad \Gamma_g \quad \text{(essential boundary condition)} \quad (2)$$

$$\boldsymbol{n}.\boldsymbol{\kappa}\nabla u = h \quad \text{on} \quad \Gamma_h \quad \text{(natural boundary condition)} \quad (3)$$

$$u(\boldsymbol{x}, 0) = u_o(\boldsymbol{x}) \quad \text{on} \quad \Omega \quad \text{(initial condition)} \quad (4)$$

where u represents the quantity being transported (e.g. concentration), $\boldsymbol{\beta}$ is the velocity field, and $\boldsymbol{\kappa}$ is the volumetric diffusivity. g and h are known functions of $\boldsymbol{x} = (x, y)$ and t, $\boldsymbol{n}$ is the unit outward normal vector at the boundary, and Γ_g and Γ_h are the complementary subsets of Γ where boundary conditions are prescribed.

Consider a finite element discretization of Ω into elements Ω_e, $e = 1, \ldots, n_{el}$, where n_{el} is the number of elements. Let the standard finite element approximation be given as

$$u^h(\boldsymbol{x}) \cong \sum_{i=1}^{nnodes} N_i(\boldsymbol{x}) u_i, \quad (5)$$

where $nnodes$ is the number of nodes, N_i is a shape function corresponding to node i, and u_i are the nodal values of u. Then, applying this approximation on the variational form of Equation (1), we arrive at a system of ordinary differential equations:

$$\boldsymbol{M}\boldsymbol{a} + \boldsymbol{K}\boldsymbol{v} = \boldsymbol{F}, \quad (6)$$

where $\boldsymbol{v} = \{u_1, u_2, \ldots, u_{nnodes}\}^t$ is the vector of nodal values of u, $\boldsymbol{a}$ is its time derivative, $\boldsymbol{M}$ is the "mass" matrix, $\boldsymbol{K}$ is the "stiffness" matrix, and $\boldsymbol{F}$ is the "load" vector [6]. In this work, we approximate the domain Ω using linear triangular elements. Thus, the global interpolation of Equation (5) is restricted to an element by

$$u^e(\boldsymbol{x}) \cong \sum_{i=1}^{3} N_i(\boldsymbol{x}) u_i, \quad (7)$$

where the superscript e means that u is restricted to an element, and N_1, N_2 and N_3 are the conventional shape functions [6]. Proceeding in the standard manner, matrices $\boldsymbol{M}$ and $\boldsymbol{K}$ and vector $\boldsymbol{F}$ are built from element contributions and it is convenient to identify their terms as:

$$\boldsymbol{M} = \mathop{\mathsf{A}}_{e=1}^{nel}(\boldsymbol{m}^e), \quad \boldsymbol{K} = \mathop{\mathsf{A}}_{e=1}^{nel}(\boldsymbol{k}^e) \quad \text{and} \quad \boldsymbol{F} = \mathop{\mathsf{A}}_{e=1}^{nel}(\boldsymbol{f}^e) \quad (8)$$

where A is the assembling operator and $\boldsymbol{m}^e$, $\boldsymbol{k}^e$ and $\boldsymbol{f}^e$ are the local contributions.

3 Solution Algorithm

To solve the time-dependent advection diffusion problem numerically employing the approach described in the previous section, we just have to solve the system of ordinary differential equations stated in Equation (6) towards a final time t_{final}. To do that, we use the Algorithm 1, which implements the well known implicit predictor/multicorrector solution scheme [15]. The algorithm: receives as input the initial values of $\boldsymbol{v}$ and $\boldsymbol{a}$ (see Equation (6)), t_{final}, Δt, the maximum number of multicorrection attempts, n_{max}, and the tolerance of the multicorrection phase, ϵ; and returns the values of $\boldsymbol{v}$ and $\boldsymbol{a}$ at t_{final}.

Algorithm 1 Predictor/multicorrector

1: Data: $\boldsymbol{v}_0$ and $\boldsymbol{a}_0$, t_{final}, Δt, n_{max}, ϵ
2: $t = 0$, $n = 0$
3: $\boldsymbol{M}^* = \boldsymbol{M} + \alpha \Delta t \boldsymbol{K}$
4: **while** $t \leq t_{final}$ **do**
5: $\quad i = 0$
6: $\quad \boldsymbol{v}_{n+1}^{(i)} = \boldsymbol{v}_n + (1-\alpha)\Delta t \boldsymbol{a}_n$
7: $\quad \boldsymbol{a}_{n+1}^{(i)} = \boldsymbol{0}$
8: $\quad norm_d = 0$
9: $\quad$ **while** $i \leq n_{max}$ and $\|\boldsymbol{a}_{n+1}^{(i)}\| \geq \epsilon \times norm_d$ **do**
10: $\quad\quad \boldsymbol{b} = \boldsymbol{F} - \boldsymbol{M}\boldsymbol{a}_{n+1}^{(i)} - \boldsymbol{K}\boldsymbol{v}_{n+1}^{(i)}$
11: $\quad\quad$ Solve $\boldsymbol{M}^* \boldsymbol{d} = \boldsymbol{b}$
12: $\quad\quad \boldsymbol{a}_{n+1}^{(i+1)} = \boldsymbol{a}_{n+1}^{(i)} + \boldsymbol{d}$
13: $\quad\quad \boldsymbol{v}_{n+1}^{(i+1)} = \boldsymbol{v}_{n+1}^{(i)} + \alpha \Delta t \boldsymbol{d}$
14: $\quad\quad i = i + 1, norm_d = \|\boldsymbol{d}\|$
15: $\quad$ **end while**
16: $\quad \boldsymbol{a}_{n+1} = \boldsymbol{a}_{n+1}^{(i)}$
17: $\quad \boldsymbol{v}_{n+1} = \boldsymbol{v}_{n+1}^{(i)}$
18: $\quad t = t + \Delta t$, $n = n + 1$
19: **end while**

In Algorithm 1, the prediction phase (lines 5 to 8) calculates an initial guess of the nodal values $\boldsymbol{v}$ and $\boldsymbol{a}$ at iteration $n+1$, where n denotes a time step, and the multicorrection phase (lines 9 to 15) iteratively calculates new nodal approximations until a convergence criteria (line 9) is reached. The most time consuming step of the algorithm is solving the linear system derived from Equation (6), lines 10 and 11. In this linear system, $\boldsymbol{M}^*$ is denoted the effective matrix, $\boldsymbol{b}$ is the residual vector, and $\boldsymbol{d}$ is the correction of the nodal values of $\boldsymbol{a}$ from one multicorrection iteration to the next. Matrix $\boldsymbol{M}^*$ is constant in time and is computed in line 3. The residual vector $\boldsymbol{b}$, however, must be computed in every multicorrection step (line 10).

Apart from the solution of the linear system in line 11, the other time consuming operations of the algorithm are the matrix vector product in line 10, and the saxpy vector update operations of lines 6 (the number of multicorrection iterations is small, but one always have to remember the Amdhal's Law), 12 and 13. We solve the linear

system of line 11 using GMRES [13]. The most time consuming operations of GMRES are a matrix vector product per iteration, and several saxpy and vector inner products. Therefore, the most time consuming operations of the whole predictor/multicorrector algorithm are matrix vector products, saxpy and vector inner products. For more on the predictor/multicorrector algorithm see [15].

4 Parallel Implementations

To solve our problem in parallel, it is necessary to code all matrices and vectors in Algorithm 1 in a way that allows parallel access, and to calculate their most time consuming operations in parallel. In order to achieved this, we create a partition of non-overlapping sets of elements, Ω_e. For that, we discretized the domain into a mesh composed of linear triangular elements, $\top = \Omega_e$, where $\{\top_1, \top_2, \cdots, \top_p\}$ represents a partition of the triangulation in subdomains, p is the number of subdomains, and $\bigcup_{i=1}^{p} \top_i = \top$ and $\top_i \cap \top_j = \emptyset$ when $i \neq j$. By dividing the computation domain into p subdomains, it is possible to spread the workload between p different cores. That is, by partitioning the matrices $\boldsymbol{M}$, $\boldsymbol{K}$ and $\boldsymbol{M}^*$, and the vectors $\boldsymbol{v}$, $\boldsymbol{a}$ and $\boldsymbol{d}$ (see Equation (6) and Algorithm 1) independently over p cores (with core i working only on subdomain $\top_i$), one can spread the workload among the p different cores.

We rewrite all matrices and vectors presented in Algorithm 1 into block matrix and block vector forms employing the well known Schur complement decomposition [13], as suggested by Jimack and Touheed [8]. By doing that, a generic vector $\boldsymbol{u}$ (representing $\boldsymbol{v}$, $\boldsymbol{a}$ or $\boldsymbol{d}$) can be ordered in the following way:

$$\boldsymbol{u} = (\underline{u}_1, \underline{u}_2, \cdots, \underline{u}_i, \cdots, \underline{u}_p, \underline{u}_{\boldsymbol{S}})^T. \tag{9}$$

The nodes of the linear triangular elements of the mesh $\top$ can be classified into interior nodes, interface nodes and boundary nodes of the domain.

Figure 1 illustrates a mesh with 50 nodes and 74 triangular elements, where the domain was partitioned into 4 subdomains to be assigned to four cores. In this mesh, nodes I and J are interior nodes of cores 3 and 4, respectively, while node K is an interface node of cores 1, 3 and 4.

In Equation (9), the sub-vector $\underline{u}_i$ is associated with the interior nodes in $\top_i$, $i = 1, 2, \cdots, p$; while $u_{\boldsymbol{S}}$, in turn, is defined as $\underline{u}_{\boldsymbol{S}} = \bigcup_{i=1}^{p} \underline{u}_{s(i)}$, an assembly of others sub-vectors that are associated with the interface nodes of each subdomain $\top_i$, $i = 1, 2, \cdots, p$. That is, each sub-vector $u_{s(i)}$ holds the interface nodes of $\top_i$. Boundary nodes are not unknowns and need not be represented in $\boldsymbol{u}$. Also following the approach suggested by Jimack and Touheed [8], a generic matrix $\boldsymbol{A}$ ($\boldsymbol{M}$, $\boldsymbol{K}$ and $\boldsymbol{M}^*$) can be written in a block matrix form as:

$$\boldsymbol{A} = \begin{bmatrix} A_1 & & & B_1 \\ & A_2 & & B_2 \\ & & \ddots & \vdots \\ & & A_p & B_p \\ C_1 & C_2 & \cdots & C_p & A_{\boldsymbol{S}} \end{bmatrix} \tag{10}$$

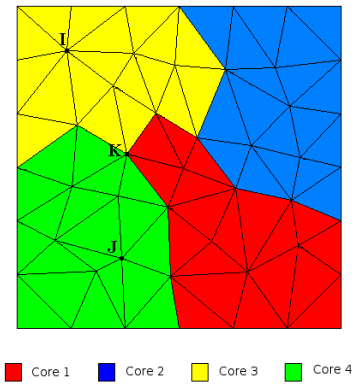


Fig. 1. Example of the partitioned mesh with 4 subdomains.

where the block-arrowhead structure of the new matrix comes from the local support of the finite element basis functions. In Equation (10), the sub-matrices A_i, B_i, C_i and $A_{\boldsymbol{S}}$ are sparse matrices that are stored using a CSR data structure.

The sub-matrix A_i stores the contribution of the interior nodes of core i on the interior nodes of core i. The matrix B_i stores the contribution of the interior nodes of core i on the interface nodes of core i. The sub-matrix C_i stores the contribution of the interface nodes of core i on the interior nodes of core i. Finally, the sub-matrix $A_{\boldsymbol{S}}$, an assembly of a set of blocks distributed over the p cores, is defined as $A_{\boldsymbol{S}} = \bigcup_{i=1}^{p} A_{s(i)}$, where the sub-matrix $A_{s(i)}$ stores the contribution of the interface nodes of core i on the interface nodes of core i.

With this approach, each of the sub-vectors $\underline{u}_i$ and $\underline{u}_{s(i)}$, and each of the sub-matrix A_i, B_i, C_i, $A_{s(i)}$ may be computed entirely by core i, for $i = 1, 2, \cdots, p$. One can also observe that core i will work only on the elements of its own subdomain $\top_i$. Assuming that the partition $\top$ is built in such way that each core deals with approximately the same number of elements and the number of vertices lying on the partition boundary is as small as possible, the amount of calculations performed by each core i will be balanced and the amount of communication will be minimized.

Following the same procedure explained above for a generic vector $\boldsymbol{u}$ and a generic matrix $\boldsymbol{A}$, we rewrite all the matrices ($\boldsymbol{M}$, $\boldsymbol{K}$ and $\boldsymbol{M}^*$) and vectors ($\boldsymbol{v}$, $\boldsymbol{a}$ or $\boldsymbol{d}$) of Algorithm 1 in a block matrix form and execute the most time consuming operations of the whole predictor/multicorrector algorithm—matrix vector product, saxpy vector update and vector inner product—in parallel.

Using the domain partitioning presented above, a matrix-vector product, $\boldsymbol{v} = \boldsymbol{A}\boldsymbol{u}$, can be computed in parallel by computing both expressions on Equation (11) below (see also Equations (9) and (10))

$$\underline{v}_i = A_i \underline{u}_i + B_i \underline{u}_{s(i)} \quad \text{and} \quad \underline{v}_{s(i)} = A_{s(i)} \underline{u}_{s(i)} + C_i \underline{u}_i \tag{11}$$

on each core $i = 1, 2, \cdots, p$. Also, using the domain partitioning presented, a saxpy vector update, $\boldsymbol{v} = \boldsymbol{v} + \lambda \boldsymbol{u}$, can be formulated as

$$\underline{v}_i = \underline{v}_i + \lambda \underline{u}_i \quad \text{and} \quad \underline{v}_{s(i)} = \underline{v}_{s(i)} + \lambda \underline{u}_{s(i)} \tag{12}$$

for $i = 1, 2, \cdots, p$, where λ is a real number. Finally, using the domain partitioning presented, a vector inner product, $scalar = \boldsymbol{u} \cdot \boldsymbol{v}$, can be computed on each core as

$$scalar = \sum_{i=1}^{p} (\underline{u}_i \cdot \underline{v}_i + \underline{u}_{s(i)} \cdot \underline{v}_{s(i)}) \tag{13}$$

for $i = 1, 2, \cdots, p$. It is important to note that this last operation requires a global communication because its result is a scalar that always must be known by all cores. This communication is a global reduction, which computes the sum of the contributions to the inner product coming from each core, and then provides each core with a copy of this sum.

In addition to global reductions required by inner products, our Multi-Core Cluster implementation performs core-to-core communication before every matrix vector product (lines 10 and 11 of Algorithm 1) in order to communicate the value of the interface nodes—we use `MPI_send` and `MPI_Recv` for that. Thanks to the assembly presented in Equation 4, the data that needs to be communicated is clearly specified (interface nodes). The partitioning of the work between the cores is made before the whole computation using METIS [9]. Please refer to our internal technical report for more details about our multi-core cluster implementation (http://www.lcad.inf.ufes.br/∼alberto /techrep01-11.pdf).

The CUDA-enabled GPU parallel version was derived from the Multi-Core Cluster parallel version and, therefore, follows the same principles described above. It was implemented in C+CUDA and, as we wanted to run it in multi-core desktop computers with multiple GPUs (or clusters of multi-core machines each of which with one or more GPUs), it takes advantage of the multiple cores for distributing the domain (or subdomains in the case of a cluster) between multiple GPUs (one subdomain per GPU) and employs MPI for inter-core communication. We choose to do this way (i) to avoid large modifications in the Multi-Core Cluster version in the process of morphing it into the C+CUDA version, and (ii) to transform our multi-core cluster code into a code that runs in clusters of multi-core machines each of which with multiple GPUs. For this process, we basically moved the main functions of the Multi-Core Cluster version into CUDA kernels and optimized the use of the GPU memory hierarchy.

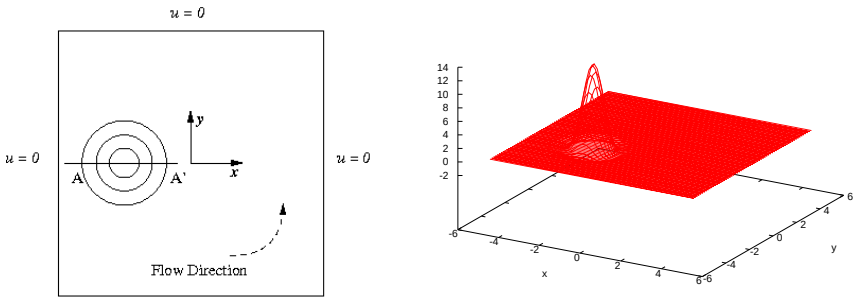
The main strategy adopted in the design of the C+CUDA version was (i) to identify the most time consuming operations of the predictor/multicorrector (Figure 1) and GMRES algorithms, (ii) to parallelize and optimize these operations, and (iii) to try and avoid data transfer between the CPU and GPU memories as much as possible.

We identified the most time consuming operations of the predictor/multicorrector and GMRES algorithms—the matrix-vector product, $\boldsymbol{v} = \boldsymbol{A}\boldsymbol{u}$, and the vector inner product, $scalar = \boldsymbol{u} \cdot \boldsymbol{v}$—using `gprof`. Please refer to our internal technical report for details about the C+CUDA implementation (http://www.lcad.inf.ufes.br/∼alberto /techrep01-11.pdf).

5 Experimental Evaluation

The Multi-Core Cluster implementation was run on the Enterprise 3 cluster of the *Laboratório de Computação de Alto Desempenho* (LCAD) at UFES. Enterprise 3 is a 24-node cluster of 24 quad-core Intel 2 Q6600 machines (96 cores), with 2.4GHz clock frequency, 4MB L2 and 4GB of DRAM, interconnected with a 48-Port 4200G 3COM Gigabit Ethernet switch. The C+CUDA implementation was run on LCAD's BOXX Personal Supercomputer, which is a quad-core AMD Phenon X4 9950 of 2.6GHz, with 2MB L2, 8GB of DRAM, and four GPU NVIDIA Tesla C1060 PCIE boards, with 240 1.3GHz CUDA cores and 4GB DRAM each.

In our experimental evaluation we solved a standard test problem for transient dominated advection flow, named rotating cone problem. The problem is described in Figure 2(a) (see [1] for details). In our experiments, the velocity field is $\boldsymbol{\beta} = (-y, x)^T$ and the diffusivity is $\boldsymbol{\kappa} = k\boldsymbol{I}$, where $k = 10^{-6}$. The exact solution consists of a rigid rotation of a cone about the center of the square domain $[-5, 5] \times [-5, 5]$. Figure 2(b) shows the solution obtained after 7 seconds of simulation.

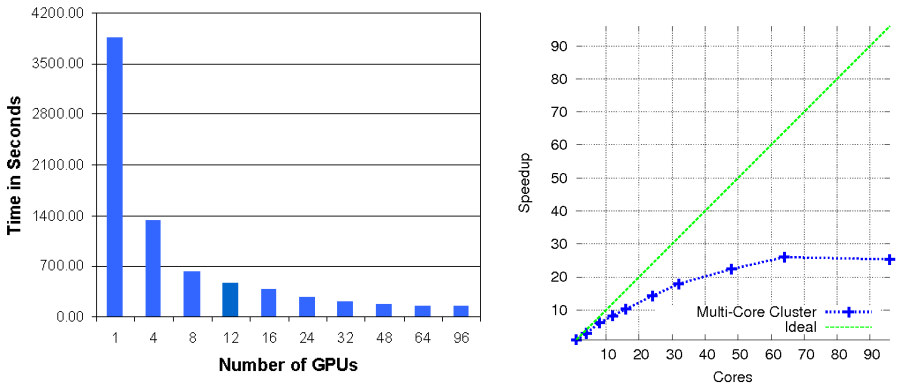


(a) Description (b) Solution

Fig. 2. Description and solution of the rotating cone problem

To evaluate the performances in terms of time of the machines examined on the solution of a large size problem, we consider the rotating cone problem in a regular mesh of 1024×1024 cells, totalizing $2,097,152$ elements, $1,050,625$ nodes and $1,046,529$ unknowns with $\Delta t = 10^{-2}$, the $t_{final} = 7$, GMRES and predictor-multicorrector tolerances equal to 10^{-3}; and number of restart vectors for GMRES equal to 10. The observed number of GMRES iterations for each correction was around 15.

Figure 3(a) shows the time it takes to solve this problem with the Multi-Core Cluster implementation running on the Enterprise 3 configured with 1, 4, 8, 12, 16, 24, 32, 48, 64 and 96 cores, while Figure 3(b) shows the speedups obtained with 4, 8, 16, 32, 64 and 96 cores. In the graph of Figure 3(a), the x-axis is the number of cores, while the y-axis is the time it takes to solve the problem in seconds. As the graph shows, there is an almost linear reduction of the time it takes to solve the problem as the number of



(a) Times (b) Speedups

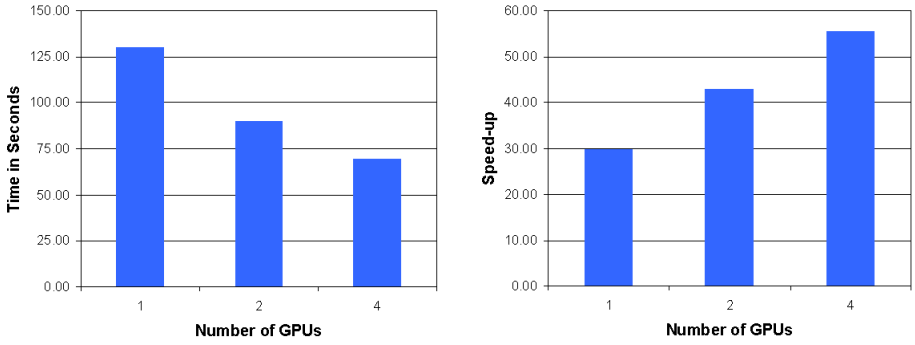
Fig. 3. Enterprise 3 times and Speedups

processors increases from 1 to 8. However, the performance gains obtained increasing the number of cores from 8 onwards decreases as the number of cores increases. This can be more easily appreciated by examining the graph of Figure 3(b). In this graph, the x-axis is the number of cores, while the y-axis is the speedup. As the graph of Figure 3(b) shows, although the speedup starts augmenting linearly, as the number of cores increases, the speedup levels of—there is no gain as one goes from 64 to 96 cores. This is to be expected because, as the number of cores increases, the amount of inter-machine communication increases, while the amount of compute work per core decreases. So, the time spend waiting for data transfer (communication) ends up surpassing the time doing computation.

Figure 4(a) shows the time it takes to solve this problem with the C+CUDA implementation running on the BOXX Personal Supercomputer configured with 1, 2 and 4 GPUs, while Figure 4(b) shows the speedups obtained with 1, 2 and 4 GPUs—these speedups were computed against a single Enterprise 3 core.

In the graph of Figure 4(a), the x-axis is the number of GPUs, while the y-axis is the time it takes to solve the problem in seconds. As the graph shows, the time it takes to solve the problem decreases as the number of GPUs increases, but not linearly. This is to be expected since the multi GPU C+CUDA implementation uses the PCI Express bus to transfer interface nodes data between the multi-core CPU and the GPUs and, as the number of GPUs increases, this bus becomes a bottleneck. Figure 4(b) presents the speedups obtained with the BOXX Personal Supercomputer configured with different numbers of GPUs (the reference time is that of a Enterprise 3 single core). In this graph, the x-axis is the number of GPUs, while the y-axis is the speedup. As the graph shows, speedups close to 60 were obtained with C+CUDA.

To better appreciate the benefits of CUDA-enabled GPUs and C+CUDA, we plot on the graph of Figure 5 the speedups obtained with the BOXX Personal Supercomputer against the best performing Enterprise 3 cluster. In the graph of Figure 5, the x-axis is the number of GPUs, while the y-axis is the time it takes to solve the problem with



(a) Times (b) Speedups

Fig. 4. BOXX Personal Supercomputer times and Speedups.

Enterprise 3 divided by the time it takes to solve the problem with the BOXX Personal Supercomputer configured with different numbers of GPUs. As the graph of Figure 5 shows, a desktop machine with a single GPU can outperform a 24-machine cluster (96 cores). Also, a desktop machine with four GPUs can deliver more the twice the performance of a 24-machine cluster (96 cores).

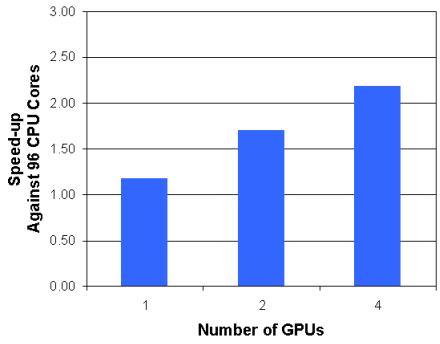


Fig. 5. BOXX Personal Supercomputer speedups: C+CUDA x Multi-Core Cluster.

To compare the performance of our Multi-Core Cluster implementation with that of our C+CUDA implementation in terms of energy consumption, we run the rotating cone problem in a regular mesh of 2048×2048 cells in both machines and measured the total current drained by each at 10-second intervals using a Digital Clamp Meter Minipa, Model ET-3880, while measuring the voltage. Figure 6 shows the measurement setup employed with each machine (voltage measurement not shown).

By numerically integrating the current $\times$ voltage (power in Watts) required by the machines in the period of time they took to solve the rotating cone problem, we were able to estimate the total energy (in Joules) consumed by each machine. The amount of Joules consumed by Enterprise 3 (all 96 cores) was equal to approximately 5,545,530 Joules (45 Amperes $\times$ 114 Volts $\times$ 1,081 Seconds). The amount of Joules consumed by the BOXX Personal Supercomputer on equivalent circumstances (127 Volts, but different currents and times for each number of GPUs) was measured for 1, 2 and 4 GPUs. Figure 7 shows the energy consumed by each machine configuration.

As Figure 7 shows, the amount of Joules decreases as the number of GPUs increases. This is to be expected, since the time to solve the problem diminishes. Note that we did not remove the unused GPU boards during these experiments and, even when not doing useful computation, the GPUs consume a significant amount of energy. Note also that the energy consumed for the whole machine was measured in all cases, and the ratio computation/energy consumption becomes worth with fewer GPUs doing useful work.

Finally, Figure 8 presents a comparison between the amount of energy consumed by Enterprise 3 versus (divided by) the amount of energy consumed by the BOXX Personal Supercomputer while solving the rotating cone problem with 1, 2 and 4 GPUs. As the graph of Figure 8 shows, the BOXX Personal Supercomputer consumes more than 20 times less energy than the Enterprise 3 cluster while solving the same problem. This result shows that, currently, considering the benefits of shorter executing times, smaller energy consumption, and smaller size and maintenance costs, Multi-GPU desktop machines are better high performance computing platforms than small clusters without GPUs such as Enterprise 3, even though they are somewhat harder to program. It is important to note that our C+CUDA code runs unmodified in clusters of multi-core machines each of which with multiple GPUs (it is, in fact, a C+CUDA+MPI code).

(a) (b)

Fig. 6. Power (current) measurement setup. (6(a)) Cluster setup: the total current consumed by Enterprise 3 was measured on the neutral wire of its power distribution panel. (6(b)) BOXX Personal Supercomputer setup: the total current consumed by it was measured on the neutral wire of its power cord.

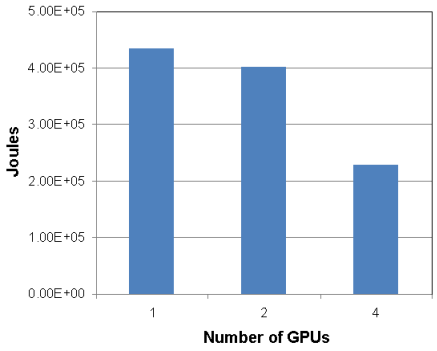


Fig. 7. Joules consumed while running the rotating cone problem with the BOXX Personal Supercomputer with 1, 2 and 4 GPUs. The unused GPU boards were not removed during the experiments.

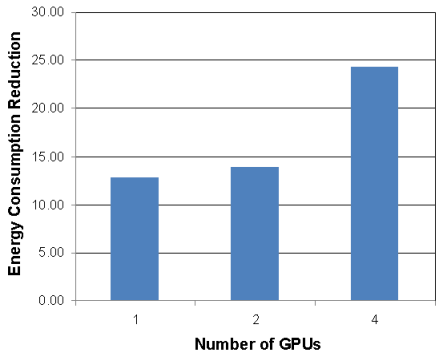


Fig. 8. Energy reduction observed while solving the rotating cone problem in the BOXX Personal Supercomputer for 1, 2 and 4 GPUs when compared with the 96-core Enterprise 3 Cluster.

6 Related Work

Since the introduction of CUDA, a number of works have demonstrated that the use of GPUs can accelerate computational fluid dynamics (CFD) simulations ([3, 11, 14, 16]). Recently, Jacobsen et al. [7] have exploited some of the advanced features of MPI and CUDA programming to overlap both GPU data transfer and MPI communications with computations on the GPU. Their results demonstrated that multi-GPU clusters can substantially accelerate CFD simulations. In this work, we compared a Multi-Core Cluster without CUDA-enabled GPUs with a desktop machine with CUDA-enabled GPUs and showed that the way pointed by the work of Jacobsen et al. and others [7] is perhaps the current only way forward in the high performance CFD simulation field.

Little research has been conducted on the evaluation of energy consumption of GPUs against that of clusters. Huang et al. [5] analyzed two parallel implementations

of a biological code that calculates the electrostatic properties of molecules—a multithreaded CPU version (for a single multi-core machine) and a GPU version—and compared their performance in terms of execution time, energy consumption, and energy efficiency. Their results showed that the GPU version performs the best in all three aspects. In this work, we showed that a parallel CUDA-enabled GPU implementation consumes considerably less energy (Joules) than a parallel multi-core cluster implementation while solving a whole instance of the finite element problem.

7 Conclusions

We used a finite element formulation to solve the 2D time-dependent advection diffusion equation in Multi-Core Clusters and CUDA-enabled GPUs. Our experimental results have shown that a desktop computer with a single GPU can outperform a 24-machine cluster of the same generation and that a 4-GPU desktop can offer more than twice the cluster performance (performance in terms of time to compute a solution). Our experimental results have also shown that a 4-GPU desktop can consume less than one twentieth of the energy (Joules) consumed by a 24-machine cluster while solving a whole instance of this relevant finite element problem. The techniques we employed for the problem tackled in this paper can be employed in much harder problems. In future works, we will examine multidimensional compressible problems governed by the Navier-Stokes equations.

8 Acknowledgments

We thank CNPq-Brazil (grants 552630/2011-0, 309831/2007-5, 314485/2009-0, 309172/2009-8) and FAPES-Brazil (grant 48511579/2009) for their support to this work.

References

1. A.N. Brooks and T.J.R. Hughes. Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 32:199–259, 1982.
2. L. Catabriga and A.L.G.A. Coutinho. Implicit SUPG solution of Euler equations using edge-based data structures. *Computer Methods in Applied Mechanics and Engineering*, 191:3477–3490, 2002.
3. Jonathan M. Cohen and M. Jeroen Molemaker. A fast double precision CFD code using CUDA. In *Proceedings of the 21st Parallel Computational Fluid Dynamics*, Monffett Fiel, California, 2010.
4. A.L.G.A. Coutinho, M.A.D. Martins, J. L. D. Alves, L. Landau, and A. Moraes. Edge-based finite element techniques for nonlinear solid mechanics problems. *International Journal for Numerical Methods in Engineering*, 50:2053–2068, 2001.
5. S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2009.
6. T.J.R Hughes. *The Finite Element Method. Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

7. Dana A. Jacobsen, Julien C. Thibault, and Inanc Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting*, Orlando, Florida, 2010.
8. P. K. Jimack and N. Touheed. Developing parallel finite element software using mpi. In B.H.V. Topping and L. Lammer, editors, *High Performance Computing for Computational Mechanics*, pages 15–38. Saxe-Coburg Publications, 2000.
9. G. Karypis and V. Kumar. Multilevel k-way partioning scheme for irregular graphs. Technical Report 95-064, Department of Computer Science, University of Minnesota, 1995.
10. D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach*. Elsevier, 2010.
11. A. Klockner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comput. Phys.*, 228:7863–7882, 2009.
12. NVIDIA. *NVIDIA CUDA 3.0 - Programming Guide*. NVIDIA Corporation, 2010.
13. Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.
14. Inanc Senocak, Julien Thibault, and Matthew Caylor. Rapid-response urban CFD simulations using a GPU computing paradigm on desktop supercomputer. In *Proceedings of the Eighth Symposium on the Urban Environment*, Phoenix, Arizona, 2009.
15. T.E. Tezduyar and T.J.R. Hughes. Finite element formulations for convection dominated flows with particular emphasis on the compressible Euler equations. In *Proceedings of AIAA 21st Aerospace Sciences Meeting*, AIAA Paper 83-0125, Reno, Nevada, 1983.
16. Julien C. Thibault1 and Inanc Senocak. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *Proceedings of the 7th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, Orlando, Florida, 2009.