Scientific Programming 19 (2011) 185–197
DOI 10.3233/SPR-2011-0333
IOS Press

CaKernel – A parallel application programming framework for heterogenous computing architectures

Marek Blazewicz [a,*], Steven R. Brandt [b,c], Michal Kierzynka [a], Krzysztof Kurowski [a], Bogdan Ludwiczak [a], Jian Tao [b] and Jan Weglarz [a,d]

[a] *Poznań Supercomputing and Networking Center, Poznań, Poland*
[b] *Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, USA*
[c] *Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA*
[d] *Institute of Computing Science, Poznań University of Technology, Poznań, Poland*

Abstract. With the recent advent of new heterogeneous computing architectures there is still a lack of parallel problem solving environments that can help scientists to use easily and efficiently hybrid supercomputers. Many scientific simulations that use structured grids to solve partial differential equations in fact rely on stencil computations. Stencil computations have become crucial in solving many challenging problems in various domains, e.g., engineering or physics. Although many parallel stencil computing approaches have been proposed, in most cases they solve only particular problems. As a result, scientists are struggling when it comes to the subject of implementing a new stencil-based simulation, especially on high performance hybrid supercomputers. In response to the presented need we extend our previous work on a parallel programming framework for CUDA – CaCUDA that now supports OpenCL. We present CaKernel – a tool that simplifies the development of parallel scientific applications on hybrid systems. CaKernel is built on the highly scalable and portable Cactus framework. In the CaKernel framework, Cactus manages the inter-process communication via MPI while CaKernel manages the code running on Graphics Processing Units (GPUs) and interactions between them. As a non-trivial test case we have developed a 3D CFD code to demonstrate the performance and scalability of the automatically generated code.

Keywords: GPGPU programming, computational framework, HPC, finite difference, stencil computation, CUDA, OpenCL

1. Introduction

For the last decade, many scientists all over the world have been using homogeneous HPC computers organized into grid environments for their computational experiments [14]. Even though performance has grown exponentially, costs (the most significant of which is power) have grown prohibitively. However, a new class of massively parallel system has emerged in recent years, namely hybrid systems which combine classic computational architectures with new types of hardware accelerators. These new architectures offer a promise of increased performance at lower power. Moreover, some of these solutions are relatively cheap, which would make HPC computing more accessible for many users. However, developing or porting legacy applications for such systems is a very difficult task because the tools and libraries supporting these new architectures are either immature, or require a lot of efforts from programmers.

As of June 2011, three out of the top five fastest computers in the world use GPUs to achieve their peek-performance [22]. While there are only ten GPU machines on the current Top 500 list, this number is likely to grow rapidly. In the United States, NCSA has recently deployed its 153 TF GPU cluster, and TACC is planning to build a Peta-scale hybrid machine in 2012. In Europe, many supercomputing centers have recently provided the access to their hybrid supercomputers aiming at the hundreds of TF, e.g., CINECA, PSNC, HLRS to mention a few.

Despite the growing trend to make GPUs available for supercomputing, there is a paucity of tools to exploit the new hardware. CaKernel addresses this issue

*Corresponding author: M. Blazewicz, E-mail: marqs@man.poznan.pl.

by providing a high-level API for stencil computations. In a nutshell, the tool extends Cactus Computational Toolkit [7,9] by assigning one GPU to one MPI process and providing functions to deal with storage on GPU, synchronization among threads, communication between CPU and GPU, and optimization on GPU. Consequently, a scientist may use CaKernel to easily write any kind of stencil code for hybrid supercomputers. In order to achieve that, a developer has to create only the core stencil computation kernel, the surrounding boiler plate code is generated automatically by so-called kernel descriptor and code generator.

The rest of the paper is organized as follows. Section 2 presents related works. The next section contains a brief description of the GPU programming models. Section 4 describes the general idea of the Cactus framework, whereas the next section presents our contribution to the framework. In Section 6 an example application of CaKernel is laid out. Tests and results are presented in Section 7. The last section presents conclusions and possible extensions to CaKernel that we would like to introduce in the near future.

2. Related works

Many scientific applications that use structured grids to solve partial differential equations have kernels that rely on stencil computations. A framework for optimizing these kernels on parallelized multiple GPUs hybrid architectures can, therefore, have a significant impact on the advancement of science. P. Micikevicius has proposed an optimal 3D finite difference discretization of the wave equation in a CUDA environment [15]. He also proposed a way to minimize the latency of inter-node communication by overlapping slow PCI-Express data exchange with computations. He achieved this by dividing the computational domain along the slowest varying dimension. Thibault and Senocak have followed the idea of the domain division pattern and implemented a 3D CFD model based on finite difference discretization of Navier–Stokes equations parallelized on single computational node with 4 GPUs [21]. Jacobsen et al. have extended this model by adding the inter-node communication via MPI [13]. They have followed Micikevicius and overlapped the communication with computations as well as GPU-host with host-host data exchange in hybrid clusters. Unfortunately their computational model divides the domain along the slowest varying dimension only, which is not suitable for all numerical problems. For large computational domains, the size of the ghost zones becomes noticeable in comparison to the remainder of the domain, and communication costs become heavier than computational costs. This manifests in the non-linear scaling of their model. Therefore, it is a non-trivial task to obtain an optimal performance on the GPU, not to mention how to achieve a sustained performance on multiple-GPUs available in hybrid supercomputers.

In order to simplify many development efforts, various programming frameworks and tools have been developed and some of them were already discussed in [20]. However, it is worth to mention a relatively new framework dedicated for stencil computations called Mint [24]. It allows for annotation in the source code so stencil loops can be executed on a GPU. Another example framework is Ypnos [19] that performs automatic parallelization during compilation process. However, both frameworks target currently only single-GPU systems. As an alternative approach addressing programmer productivity and code scalability for hybrid supercomputers, we present CaKernel, an extension to the highly scalable and well-known Cactus framework. In general, CaKernel enables programmers to design and develop highly efficient stencil-based kernels than can be accelerated by many GPUs available in hybrid systems. CaKernel uses automatic code generation, drawing upon a highly optimized set of code templates, and frees scientific application developers from the need to understand the details of optimization in GPU programming. The Cactus framework frees the developer from the need to write basic infrastructure, e.g., parameter parsing, checkpointing and MPI communication. Thus, such combination means that developers of scientific applications can focus on their domain science rather than on complex technical details behind hybrid supercomputing.

3. GPU programming models

Currently, a GPU, in comparison to a CPU of similar area, can execute more floating-point operations and transfer more data at the same time. This is because GPUs reduce the local storage and control complexity comparing to the CPU. These absences increase the burden on the programmer to achieve the device's performance potential.

CPUs typically provide two main levels of local memory, the large indexed address space called the

main memory, and a much smaller set of registers. The access to registers is guaranteed to be very fast, whereas the access to the main memory is often accelerated thanks to a hierarchy of caches. In contrast, GPUs have been optimized to deal with graphics. Thus, GPUs consist of many small multiprocessors that can run many threads that have a faster access to *global memory*, but caches are not managed automatically [18]. Therefore, application programmers using GPUs have to carefully design data structures and algorithms to use efficiently caches to obtain a very good performance. A modern CPU core can sustain execution of four (or more) instructions per cycle from a single thread. In contrast, current NVIDIA GPUs need at the very least 64 threads per multiprocessor, which is a rough equivalent of a CPU core, in order to fully utilize its computational resources. Even more threads are required to hide the latency of accessing the device's *global memory*. This aggravates the local storage problem and increases the impact of threads' own initialization code. A second impact of reduced control complexity is the loss of performance when branches within so-called *warp* do not share the same direction. A *warp* is a group of consecutive threads, usually 32 for NVIDIA's and 64 for AMD's GPUs.

Programming of GPUs is typically done using CUDA [18], OpenCL [16]. CUDA and OpenCL abstract the GPU to a lightweight multi-threaded machine with an elaborate memory hierarchy. CUDA was developed by NVIDIA for their own GPUs, starting with the G80 series. OpenCL is similar in many substantial ways to CUDA, but was developed by the independent Khronos Group. CUDA and OpenCL abstract important GPU memory hierarchy and execution organization features in similar ways. However, OpenCL was written as an independent standard that could be implemented for any vendor's hardware. It is currently used, for instance, as a main programming language for AMD's GPUs. In this paper, we present a framework for stencil computations which can use both CUDA and OpenCL.

3.1. CUDA

The main programming vehicle in CUDA is a C++-based language, called CUDA C. It is used to code routines, so-called *kernels*, that are executed on a GPU by a large collection of threads. Application CPU code launches *kernels* and also initiates data transfers between the CPU and GPU and between GPUs, if more than one is present. Kernel launches can specify dependencies forcing one kernel to wait for the completion of another, or indicating which two can execute concurrently. Dependencies can also be specified between kernels and memory transfers. Additionally, NVIDIA GPUDirect technology allows to transfer data directly between GPUs. The Unified Virtual Addressing, in turn, facilitates the memory access allowing for higher programmer productivity.

CUDA C provides the programmer with great freedom. But unless certain rules are followed, program speed on a typical GPU will be just a small fraction of its potential. These rules, based on the microarchitecture of the GPU that will run the code, restrict memory access ordering, and encourage the use of so-called *shared memory* to avoid multiple global accesses, sufficient thread parallelism for latency hiding, and code structuring to achieve warp-level branch convergence. For details see [18].

3.2. OpenCL

OpenCL (Open Computing Language) is an open standard defined by Khronos Group [16]. In contrast to the CUDA library supported mainly by NVIDIA it is a vendor-independent framework that enables the programmer to develop software that can be executed on heterogeneous platforms consisting of CPUs, GPUs and potentially other processing units like FPGAs. Despite the differences in philosophy and design, OpenCL is quite similar to CUDA. Kernels written in C99 standard are compiled in the program runtime and then can be executed in many threads on a single or multiple devices. OpenCL supports both task-based and data-based parallelism. It also defines an API to query and control devices of different architectures and from different manufacturers. The framework targets many architectures and as such is quite generic. However, in order to make a given algorithm run efficiently, the developer needs to take care of hardware specific parameters, likewise in CUDA [17].

4. Cactus computational framework

The Cactus computational framework [1,7,10] is an open source, modular, highly portable, programming environment for collaborative research using high performance computing. Cactus is distributed with a generic parallel computational toolkit providing parallelization, domain decomposition, coordinates, boundary conditions, interpolators, reduction operators and efficient I/O in different data formats.

4.1. Cactus architecture

A Cactus application consists of a central piece called the "flesh" and a collection of modules called "thorns" (see Fig. 1). The flesh provides a framework for defining and parsing parameters, for scheduling work, interoperation between C, C++, F77 and F90, as well as interaction with other thorns. A "driver thorn" is required for any Cactus application. It is responsible for a number of important runtime tasks including memory management, synchronization of grid functions and distribution of data. Thorns are described using a domain specific language (DSL) called the "Cactus Configuration Language" (CCL) [2]. The information in the CCL files includes the name of the implementation, the declaration of functions and parameters, the schedule of the routines, whether they require synchronization after execution, etc. Details can be found below in Section 4.2.

Cactus flesh is distributed with the Cactus computational toolkit (CCTK) which includes several arrangements of thorns (i.e., groups of thorns with related functionality) providing basic utilities. New functionality can be added to Cactus by simply extending existing thorns or creating new ones. The user may implement their own routines solving a given problem by the same means.

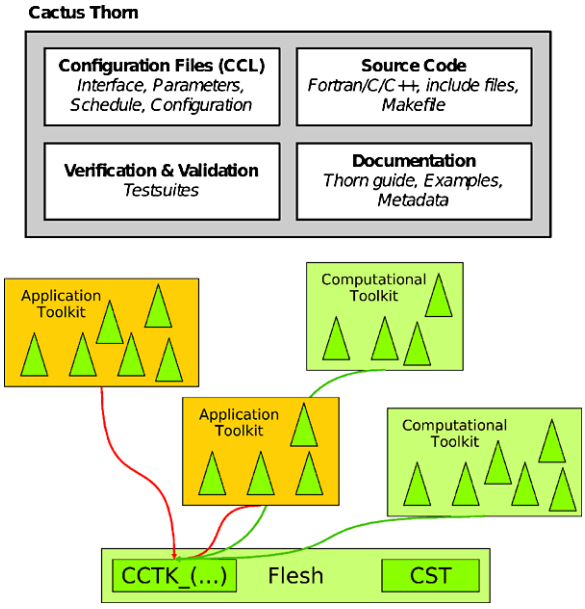

Fig. 1. The left diagram shows the internal structure of a typical Cactus thorn while the right one gives an overview of a typical Cactus application. In the right diagram, Cactus Specification Tool (CST) provides bindings for the flesh and all Cactus thorns. The Cactus Computational Toolkit (CCTK), which is distributed with the flesh, provides a range of computational capabilities, such as parallel I/O, data distribution, or checkpointing via the flesh. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0333.)

4.2. Cactus configuration language

Cactus requires three files in CCL, interface.ccl, schedule.ccl and param.ccl, from each contributing component to create global data structures, setting runtime parameters, and binding all the C and Fortran subroutines interacting through the schedule tree.

- *interface.ccl* defines the thorn interface and inheritance along with variables and aliased functions. Thorns typically do not define relationships with other specific thorns, nor do they communicate directly with other thorns. Instead they define relationships with an interface, which may be provided by one of multiple thorns. This distinction exists so that thorns providing the same interface may be independently swapped without affecting each other. Interfaces in Cactus are fairly similar to abstract classes in Java or virtual base classes in C++.
- *schedule.ccl*: defines a schedule tree to provide a static workflow which orchestrates a computation in Cactus. Besides controlling when and how scheduled functions provided by thorns should be invoked by the Cactus scheduler, a schedule tree also controls when storage for so-called grid function should be allocated and freed. Application developers are responsible for scheduling their routines at an appropriate place in the tree (e.g., INIT, PRESTEP, EVOL, POSTSTEP, etc.). Cactus ensures that the routines are called at the correct time.
- *param.ccl*: defines parameters which can be specified in a Cactus parameter file and initialized at the start of a Cactus run. Cactus provides tools to parse the parameter file and check the range.

Besides the three required files, there is one more optional file,

- *configuration.ccl*: defines build–time dependencies in terms of provided and required capabilities, e.g., interfaces to Cactus–external libraries.

The CCL files will be parsed by the flesh during the compilation stage. After parsing the CCL files, the flesh enables component binding by generating source code to instantiate the different required thorn vari-

ables, parameters and functions, as well as checking required thorn dependencies. When Cactus starts up the flesh parses a user provided parameter file which defines which thorns are required and provides key/value pairs of parameter assignments. The flesh then activates only the required thorns, sets the given parameters using default values for parameters which are not specified in the parameter file, and creates the schedule of which functions provided by the activated thorns to run at which time. The Cactus flesh provides the main iteration loop for simulations (although this can be overloaded by any thorn). However, the flesh leaves memory allocation and parallelization to a driver thorn.

5. CaKernel programming framework

We extended the CaCUDA kernel abstraction presented in the paper by Blazewicz et al. [4,20] to implement a programming abstraction in Cactus for heterogeneous architectures with OpenCL. This abstraction enables automatic code generation from a set of highly optimized OpenCL and CUDA templates to simplify the development of scientific applications.

This kernel abstraction makes it easy to write and execute computational kernels, but also makes it possible to optimize the kernel without changing the kernel code itself. The whole optimization process is handled by swapping the templates or adjusting the kernel parameters. In the *Kernel Abstraction*, there are three major components: *Kernel Descriptor*, *Computation Templates* and *Code Generator*.

5.1. Kernel descriptor

The CaKernel kernel descriptor is similar in both format and function to other Cactus Configuration Language (CCL) files (see Section 4.2). The CaKernel kernel descriptor (*cakernel.ccl*) is used to declare the variables and parameters that will be needed in the computations. The kernel descriptor is defined by:

- *CCTK_KERNEL* <*name*>: This keyword begins the kernel descriptor definition. It is followed by the kernel name, then by additional parameters of the form <*key*>=<*value*>. These parameters describe performance characteristics and grid point dependencies. In the current implementation these are:
  - *TYPE*: States on the required neighborhood of each computational grid point. *3DBLOCK* declares that grid points in all directions are required for proper computations. *3DSTENCIL* defines that only grid points in cardinal directions are required. The last value, *BOUNDARY*, is optimized for physical boundary update.
  - *STENCIL*: This parameters defines the number of additional grid points required for computations in each direction.
  - *TILE*: This parameter defines the dimensions of the kernel computational block. In future versions of the framework, it will rather be used as an optimization hint than setting the parameter itself.
- *CCTK_KERNEL_VARIABLE*: This keyword begins declaration of variables required in current kernel definitions. The pointers to the variables will be automatically provided to the kernel and data synchronized with the host. Additional parameters of the form <*key*>=<*value*> may also be provided.
  - *CACHED*: When set to *YES* the variable is automatically fetched and stored in local kernel's memory, either in *shared* memory or *registers*, depending on the current optimization scheme.
  - *INTENT*: If the intent is *IN*, then the GPU variable will be updated and synchronized from the CPU before the kernel is executed. If the intent is *OUT*, the CPU variable will be updated from the GPU variable after the kernel is executed. If it is *INOUT*, both the effects of *IN* and *OUT* are combined. If the intent is *SEPERATEINOUT*, then two variables are used on the GPU to refer to a single variable on the CPU.
- *CCTK_KERNEL_PARAMETER*: This keyword begins the declaration parameters that will be automatically passed to the kernel.

5.2. Computation templates

Computation templates allow CaKernel to automatically generate OpenCL or CUDA code based on kernel descriptor and user's numerical code. Each template is optimized for a given type of computational and communication strategy. A single computational type, i.e., *3DSTENCIL*, *3DBLOCK* or *BOUNDARY*, specified in the kernel descriptor corresponds to one or more templates. Templates together with kernel de-

```
#define CaKernel_KERNEL_%{name}
    _Computations_Begin_s                \
 for(tmpj = 0; tmpj < tilez_to; tmpj++) \
 {                                       \
   __syncthreads();

#define CaKernel_KERNEL_%{name}
    _Iterate_Local_Tile_s                \
  %for_loop(tmpi,'-%{stencil_zn}',       \
    '%{stencil_zp}') %[                  \
      %var_loop("cached=yes") %[         \
      I3D_l(%vname, 0, 0, %var(tmpi)) = \
    I3D_l(%vname, 0, 0, %var(tmpi) + 1);\
    ]%                                   \
  ]%                                     \
  gk = gk2 + tmpj;

#define CaKernel_KERNEL_%{name}
    _Fetch_Front_Tile_To_Cache_s         \
  %var_loop("cached=yes") %[             \
    I3D_l(%vname, 0, 0, stncl_zp) =      \
    I3D(%vname, 0, 0, stncl_zp);]%       \
    __syncthreads();

#define CaKernel_KERNEL_%{name}
    _Limit_Threads_To_Compute_Begin_s    \
/* TODO Add your computations here */    \
/* TODO Store results to global array */\

#define CaKernel_KERNEL_%{name}
    _Limit_Threads_To_Compute_End_s      \
 }
#define CaKernel_KERNEL_%{name}
    _Computations_End_s                  \
}
```

Fig. 2. A section of a CaKernel template that can be used to generate a local stencil for the variables defined in the kernel descriptor in the shared memory.

scriptors are passed to the code generator that produces header files containing definitions of kernels in a form of C-language macros. An example template is shown in Fig. 2.

Another advantage of introducing the templates is that the user does not need to be aware of a GPU-specific architecture. Scientists using CaKernel can concentrate on their domains rather than on technical details of a particular hardware. Moreover, the same computations may be easily launched on other architectures without rewriting the code.

In addition to our previous work, we have added the templates hierarchy. The templates hierarchy facilitates inter-template dependencies definition. For example each computational template require device initialization, memory handling, etc. The templates responsible for these operations are "triggered" to be

```
{
  CCTK_KERNEL_TEMPLATE mem
    scope=group compile=no
    file="cacuda.vars-template.h" {
    triggers{
      mem$comp, comm$comp, shared
    }
  } "a template that defines the neccesary
      variables"

  CCTK_KERNEL_TEMPLATE comm$comp
    scope=group compile=yes
    file="cacuda.comm-template.cu" {
    schedule {
      "schedule CaKernel_CopyToDev in
          CCTK_BASEGRID after
          CaKernel_InitFunctions
      {
          LANG: C
      }\"Allocate memory for variables on
          devices\""
    }
    include { mem, shared }
  } "a template that performes the neccesary
      boundary data exchange"
  ...
}
```

Fig. 3. A subset of syntax for defining the template hierarchy structure.

evaluated if one of the computational schemes is to be used. The templates hierarchy is located in additional configuration file *deps.ccl* which syntax is similar to the CCL language. The file is read and interpreted during the compilation by code generator. From the user's point of view the template hierarchy enables compilation of the code for different architectures or different optimization schemes by changing only the name of the computational type in *cakernel.ccl* file. The code generator supports also a small fraction of regular expression syntax to enable multiple architecture and optimization schemes specification within one kernel descriptor definition. The example fragment of the template hierarchy definition syntax is presented in Fig. 3.

5.3. Code generator

The CaKernel framework uses Piraha [6] as a parser for the kernel descriptor and as a code generator, automatically creating OpenCL and CUDA-based macros from *cakernel.ccl* and templates. The Piraha Parsing Expression Grammar (PEG) is designed to resemble the Java regular expression API and syntax, but at the

```
g.compile("w", "([ \t\r\n]|#.*)*");
g.compile("w1", "([ \t\r\n]|#.*)+");
g.compile("KERNEL", "CCTK_CUDA_KERNEL{-w1}{
    name}{-w1}({key}{-w}={-w}{value}{-w})
    *\\{{-w}({VAR}{-w}|{PAR}{-w})*\\}{-w}");
g.compile("KERNELS", "^{-w}({KERNEL})*$");
g.compile("name", "[A-Za-z0-9_]+");
g.compile("key", "{name}");
g.compile("value", "{name}|{dquote}|{squote
    }");
g.compile("dquote", "\"(\\\\[^]|[^\\\\\"])
    *\"");
g.compile("squote", "'(\\\\[^]|[^\\\\'])*'");
g.compile("VAR", "CCTK_KERNEL_VARIABLE({-w
    1}({key}{-w}={-w}{value}{-w})*|)\\{{-w}{
    name}({-w},{-w}{name})*{-w}\\}{-w}{dquote
    }");
g.compile("PAR", "CCTK_KERNEL_PARAMETER({-w
    1}({key}{-w}={-w}{value}{-w})*|)\\{{-w}{
    name}({-w},{-w}{name})*{-w}\\}{-w}{dquote
    }");
g.compile("digit", "[0-9]+");
g.compile("any","[^]*");
g.compile("par","{key}{-w}={-w}{any}");
```

Fig. 4. Piraha PEG grammar for parsing the kernel abstraction.

same time it provides the full power of a grammar parser. Figure 4 presents the grammar defined in Piraha and used to parse the kernel definitions. Both OpenCL and CUDA implementations share the same parser and grammar, only the templates are separated. As a result, CaKernel is able to generate OpenCL as well as CUDA code with the same set of tools from the same kernel descriptor and user's code.

Additionally, the code generator gives us opportunity to optimize OpenCL and CUDA code during evaluation of a template by performing loop unrolling. Although recommended on GPUs, the compilers often are not capable to perform this optimization. This enhancement allowed us to significantly speed up our example application.

The code generator makes it possible to solve a particular problem with the preferred algorithm for finite difference calculations. The user only needs to write code for a single element in the computational domain and the automatically generated macros loop around all the points that are necessary to carry out local computations.

6. Example application

In order to present a practical application of proposed framework we implemented a Computational Fluid Dynamics (CFD) code using CaKernel. CFD is one of the branches of fluid mechanics which uses numerical methods and algorithms to solve and analyze fluid flows. Currently it is used in many areas of research including petroleum reservoir simulations, weather forecasting and optimizing aerodynamic shape of planes, cars and sport suits. In our research we have focused on a simple test case, namely the lid-driven cavity problem. Since the main purpose for developing this application is to test the performance of CaKernel environment, computations were performed in single precision to make optimal use of the GPUs ability.

6.1. Background and governing equations

The CFD simulation we chose to perform is based on the Navier–Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla\phi + \nu\nabla^2\mathbf{u} + \mathbf{f}, \tag{1}$$

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

where $\mathbf{u}$ is the velocity field, ν is the kinematic viscosity, $\mathbf{f}$ is the body force, ϕ is the modified pressure (pressure over density).

We followed the discretization described in Hirt [12] and Torrey [23]. The problem was chosen because the solution can be easily implemented on a uniformly structured grid with a small number of grid functions.

6.2. Code validation and verification

To verify the numerical computations as well as to test the overall performance, we've solved the lid-driven cavity problem with a Reynolds number of 100. As a verification test, we present a comparison of X component of the velocity field in the midsection along the Y-axis with those measured by Ghia [8] in Fig. 5.

6.3. Sample of kernels definitions

We present a small fraction and potential of the CaKernel framework with an example of a CFD application. For solving the Navier–Stokes, two kernels were implemented: explicit time integration and iterative solver of pressure Poisson equations. After creating the kernels' descriptors, the header files with the proper macros were automatically generated. An sample of generated code is presented in Fig. 6.

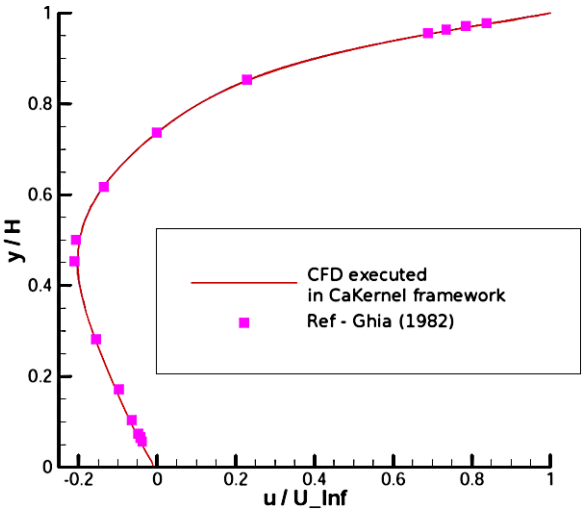

Fig. 5. Validation of the numerical results: the contours of the X component of the velocity field along Y-axis in the midsection. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0333.)

As one may see, data caching and looping over the grid points is done automatically without user's explicit interaction. This was facilitated by indexing macros, i.e., dereferencing the proper element of an array, either locate in main memory (*I3D*) or local (*I3D_l*). It is important to note that only the variables specified as *cached* in the kernel descriptor may be accessed using the *I3D_l* macro. Other uses would raise a compilation error. It is also worth mentioning that we are capable of efficiently accessing different types of local memory within a single *I3D_l* macro. This is very useful in facilitating programming on the GPGPU architecture, where significant limitations in cached memory require sophisticated optimizations patterns. The sample of indexing macros with combination of C++ templates is presented in Fig. 7. The proposed indexing pattern combined with generated kernel frame (see Fig. 6) results in stencils along the X and Y directions being stored in *shared* memory and stencils in the Z direction being stored in *registers*. Such indexing significantly simplifies the caching pattern proposed by P. Micikevicius [15]. A small section of a sample user's numerical code utilizing the automatically generated macros is shown in Fig. 8.

7. Performance results

The first step to create the CaKernel framework was to implement the standalone, CUDA-based version of the CFD application. We have discussed details of the

```
#define CAKERNEL_Update_Vel_Begin        \
__global__ void CAKERNEL_Update_Vel      \
const CCTK_REAL *vx,CCTK_REAL *vx_out,   \
                                   ...)\
{                                        \
 /* Declaring the 2D TILE of cached gf*/\
 __shared__ CCTK_REAL vx_sh[TX][TY];     \
/* Declaring the z+- stencil cached gf*/\
 CCTK_REAL vx_p1, vx_n1;                 \
/*Decl.ctrl. flow variables: indx,etc.*/\
 short li = threadIdx.x;                 \
 int gi = blockIdx.x *(CAKERNEL_Tilex - \
             stncl_xn - stncl_xp) + li;\
 bool fetch_data = gi < params.cagh_ni; \
  short tilez_to = TZ;                   \
  short tmpj;                            \
 if(fetch_data)                          \
 {                                       \
/* Caching the initial data: z-stencil*/\
  I3D_l(vx, 0, 0, 0) = I3D(vx, 0, 0,-1);\
  I3D_l(vx, 0, 0, 1) = I3D(vx, 0, 0, 0);\

# define CAKERNEL_Update_Vel_Comp_Begin \
 /* Proceeding along the z direction */ \
  for(tmpj = 0; tmpj < tilez_to; tmpj++)\
  {                                      \
   __syncthreads();                      \
 /* Rotating the cached variables */     \
I3D_l(vx, 0, 0,-1) = I3D_l(vx, 0, 0, 0);\
I3D_l(vx, 0, 0, 0) = I3D_l(vx, 0, 0, 1);\
   gk = gk2 + tmpj;                      \
 /* Fetching the front tile */           \
   I3D_l(vx, 0, 0, stncl_zp) =           \
               I3D(vx, 0, 0, stncl_zp);\
   __syncthreads();                      \
   if(compute)                           \
   {                                     \
 /* Your scientific code here */         \

# define CAKERNEL_Update_Vel_Comp_End   \
   }                                     \
  }
#define CAKERNEL_Update_Vel_End         \
}
```

Fig. 6. A part of automatically generated macros to loop over the grid points and caching the grid functions. The code has been simplified for better readability – only one grid function is considered and not all control flow variables are presented.

implementation and the performance results in [5]. This standalone application was used to test different computational patterns in order to identify the strategy to get the optimal performance. The application is capable of performing computations on different architectures: CPU (sequentially and with OpenMP) and GPU (CUDA). Among the nodes it uses MPI to make large scale simulations possible. This code obtains ap-

```
template<int i, int j, int k, typename t>
__device__ inline CI3D_l_t (t ptr_sh, const
 short &li, const short &lj, const short &lk,
 CCTK_REAL &v_p1, CCTK_REAL &v_n1)
{
    if(k == 0) return ptr_sh[j + lj][i + li];
    if(k == 1) return v_p1;
    if(k ==-1) return v_n1;
}

#define I3D_l(ptr, i, j, k) \
    I3D_l_t<i, j, k>(ptr##_sh, li, lj, lk,
        ptr##_p1 , ptr##_n1)
```

Fig. 7. A combination of macro with C++ template function for indexing cached grid functions. Thanks to optimization of the code during the compilation process, register variables are accessed via index as if they were explicitly referenced by a user.

```
#include <CaKernel_Update_Vel_3DBlock.h>
CAKERNEL_Update_Vel_Begin
/* users temporary variables */
  CCTK_REAL tmpf = 0, v_sum, v, va, vb;
CAKERNEL_Update_Vel_Computations_Begin
/* more user's code above */
  v = (I3D_l(vx,0,0,0) + I3D_l(vx,-1,0,0) +
      I3D_l(vx,0,1,0) + I3D_l(vx,-1,1,0)) /
      4.0;
  va = I3D_l(vy,1,0,0) - I3D_l(vy,0,0,0);
  vb = I3D_l(vy,0,0,0) - I3D_l(vy,-1,0,0);
  tmpf = params.cagh_dx * 2;
  v_sum -= v / tmpf * (va + vb + COPYSIGN
      (0.2, v) * (vb - va));
/* more user's code below */
CAKERNEL_Update_Vel_Computations_End
CAKERNEL_Update_Vel_End
```

Fig. 8. A section of a scientific code responsible for explicit time integration. The macros in the code are defined in an automatically generated header file.

proximately 1 GFLOPS when running the sequential CPU computations, almost linear scalability when running with OpenMP, and almost 100-fold speedup on a single NVIDIA GeForce GTX480 Fermi card.

While designing the CaKernel environment, we focused on the computational patterns proven to be efficient in stencil computations. The patterns were further generalized to fit a wider variety of numerical problems. Implementation of the CFD application in the newly created CaKernel environment results in a similar computational performance to the highly optimized standalone implementation. A more detailed discussion of performance results in CaCUDA environment, i.e., a CUDA-only predecessor to the CaKernel environment, can be found in our previous work [4].

In this paper we present the detailed performance measurement of the CFD application that is developed with CaKernel. The biggest emphasis is given to scaling, i.e., the ability of the framework to improve the performance depending on the growth of computational resources. Particularly weak and strong scaling evaluations were performed which are known to be good performance tests of parallel computations. The theoretical background of such approach may be found in: [3,11]. All the computations were performed on a cubic domain to fit largest variety of computational problems, challenge the domain division in all directions simultaneously and avoid favoring the GPU-specific capability of most efficient data transfer along the slowest varying dimensions (Z-axis). In order to limit the amount of copied data between CPU and GPU to minimum, the boundary data exchange was performed using special CUDA and OpenCL APIs to transfer 3D slices. The latency introduced by additional data transfer between GPU and CPU via PCI Express 2.0 bus was hidden by increasing the size of ghost zones and performing redundant computations. The optimal size of ghost zones that result in the best performance could be estimated by carrying out several testing runs.

The computations were performed on up to 16 nodes, each containing two NVIDIA Tesla M2050 GPUs. The nodes were interconnected using 40 Gbps InfiniBand network.

7.1. Weak scaling

Weak scaling is a form of scalability testing with a fixed problem size per computing unit. The growth of the domain during the weak scaling test was kept proportional in all directions to preserve the cubic shape of the domain and the same number of grid points per node. From preliminary tests we have observed that the biggest impact on the overall performance of computations executed on multiple GPUs is the decomposition of the computational domain. As we observed, the problem of finding the optimal division consists of two subproblems with contradictory solutions:

- Minimizing the ratio of ghost cells: We need to minimize the number of boundary which have to be transferred between neighboring nodes. In practice it leads to dividing the domain into shapes similar to cubes.
- Limiting the number of divisions in the X direction to zero: Efficient utilization of the NVIDIA

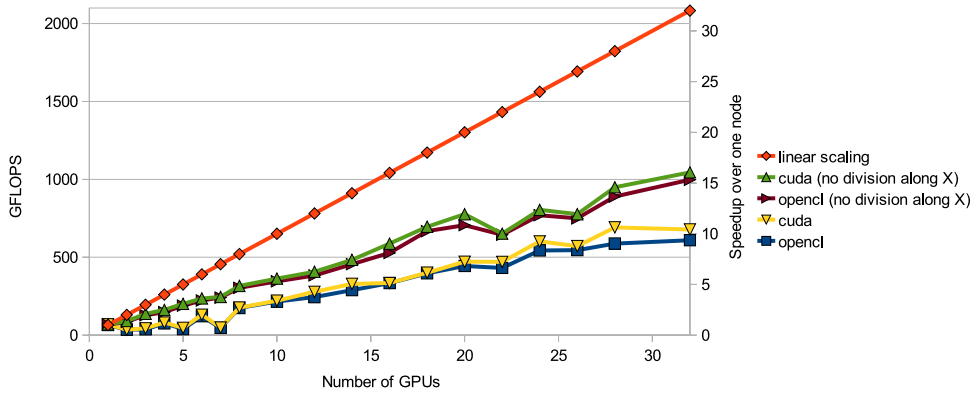

Fig. 9. The graph presents the results of the weak scaling test for both CUDA and OpenCL implementations. The domain size on each GPU was equal to 192^3. The chart shows the performance (in GFLOPS) and the speedup over a single GPU depending on the number of GPUs used. Two cases were considered: with and without domain division along the X direction. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0333.)

GPUs architecture requires accessing consecutive elements in memory by consecutive threads in the block. Unfortunately, neither the CUDA nor the OpenCL API require explicit definitions of blocks and threads properties while performing the 3D slice data exchanges. The impact of improper memory transfers is noticeable. Dividing the domain to fulfill the requirements of proper inter-node load balancing and additionally fitting the specifics of GPU architecture is not a trivial task and requires explicit modifications to the parts of the Cactus flesh which are responsible for domain division.

As one may observe in Fig. 9 the better solution (i.e., the one without domain division along the X direction) of the *CaKernel* framework scales approximately half as well as the theoretical optimal result (i.e., linear scaling). Despite of satisfactory scaling results for GPUs architecture specifics (i.e., additional latency and overhead introduced by data exchange between CPU and GPU via PCI Express) it may be improved by following one of two techniques:

- Extending the per node subdomain size: In our weak scaling test we used the smallest per node subdomain (192^3, 189 MB and approximately 16% of the NVIDIA Tesla 2050 GPU capacity) that properly leveraged all of the GPUs computational resources in our particular computational problem. Using larger per-node subdomains could reduce the ratio of redundant computations and amount of data exchanged between nodes to computations performed on the central part of the subdomain;
- Resolving the second subproblem of domain division: This can be accomplished either by solving problems with non-cubic domains and aligning the smallest side of the domain along the X dimension, or by improving the data exchange on the boundaries along the X-axis.

Another interesting feature observed during the weak scaling test is presented in the Fig. 10. The graphs point out the pros and cons of dividing the domain following each of identified domain division subproblems and its impact on computations and communication performance. As one may observe, the communication time has great impact on the overall performance of computations performed over domain divided along the X dimension. It needs to be mentioned that the division in the X dimension was only divided into two regions. Nevertheless the time needed for computations was approximately equal to time required for data exchange between nodes. That is why it so important to find the optimal domain division.

The graphs also present another important feature: growing time of computations, i.e., explicit time integration and Poisson pressure solver along with increasing the number of nodes taking part in computations. The reason of such behavior is the growing number of redundant computations introduced either by suboptimal domain division or increasing number of neighbors for each node.

7.2. *Strong scaling*

Strong scaling is a form of scalability which keeps the problem size fixed while the number of processors is increased. In this test we used a cubic domain with

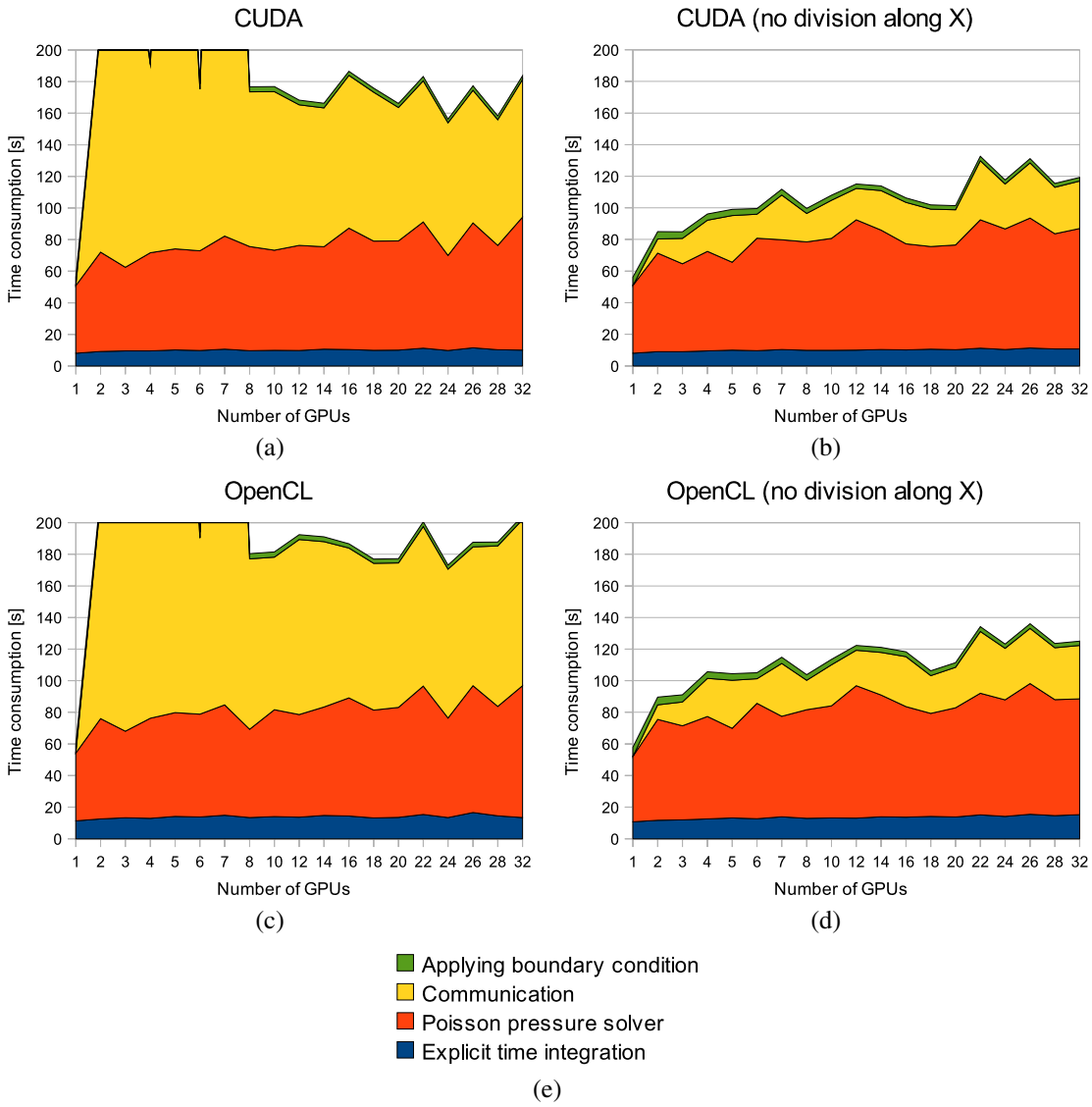

Fig. 10. The figures present time consumption of each computational step for domain division. In (a) and (c) domains are divided into cubes. In (b) and (d) we avoid cutting the domain along the X direction. All graphs present functions of time consumption (in seconds) for each functional part of execution, measured over 1000 iterations, depending on the number of GPUs taking part in computations. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0333.)

side equal to 512 grid points. Following the experience gained from weak scaling tests, we have followed the optimal solution for domain division and avoided splitting the domain along the X direction (see Section 7.1 for details). In Fig. 11 the framework scales even better than during the weak scaling and obtained approximately 64% of the linear performance when computing on 32 nodes. The main reason of such great scaling performance is that the NVIDIA GPUs perform better on smaller amounts of data. In this particular test case the computations on single node are performed on data which occupies the full capacity of the NVIDIA GPU. In practice, i.e., taking into consideration the maximum potential of the GPUs to solve this particular problem, the strong scaling performance is approximately 45% when comparing to linear scaling.

8. Conclusions

By extending our previous work on a parallel programming framework for CUDA, we designed and successfully implemented a new computational kernel

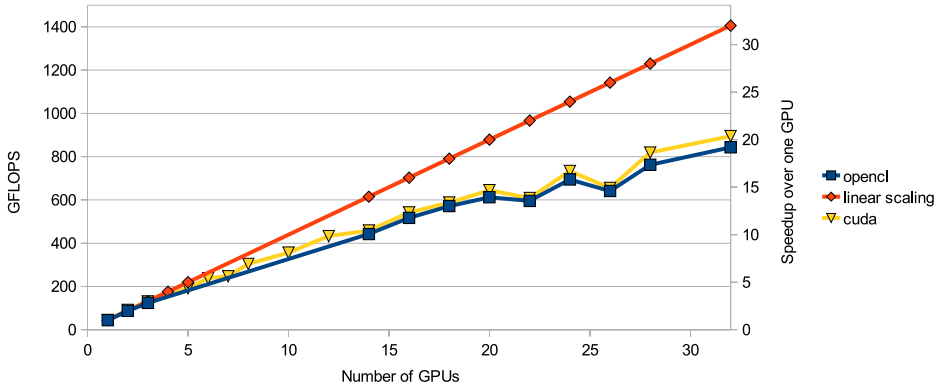

Fig. 11. The graph presents the results of the strong scaling test. The domain size was fixed and equal to 512^3. The chart shows the performance (in GFLOPS) and the speedup over a single GPU depending on the number of GPUs used. Both CUDA and OpenCL implementations were tested. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0333.)

abstraction called CaKernel. Our solution is e for developing highly efficient large-scale applications with stencil-based computations on hybrid supercomputing systems that have already reached the level of Petaflops. In particular, we created a set of highly optimized OpenCL templates for highly efficient stencil computations. Users of CaKernel can now easily switch from CUDA to OpenCL by modifying only one control parameter. Thus, they can ran their scientific applications on different hybrid systems independently for the installed accelerated hardware. As a vendor-independent framework, OpenCL supports a wide range of hardware platforms with a mixture of CPUs, GPUs, or even FPGAs etc. This in turn makes CaKernel a more powerful parallel programming framework on heterogeneous systems. Both the CUDA and OpenCL implementations of CaKernel have been tested and benchmarked with a 3D CFD code. Although the performance and scalability results were very good as we demonstrated in previous sections, we are still working on various improvements to CaKarnel. In the near future we would like to improve procedures responsible for the fully automatically generated code. Additionally, we would like to perform more comprehensive benchmarks using CaKernel on the biggest hybrid systems. Despite the fact that CaKernel was primarily designed and implemented for applications involving intensive stencil-based computations, other types of applications will added in the future using the template-based code generation approach.

Acknowledgements

This work is supported by the NG-CHC project (NSF award 1010640) through the Louisiana Board of Regents and the NSF award PIF-0904015 (CIGR). This work is also supported by the UCoMS project under award number MNiSW (Polish Ministry of Science and Higher Education) No. 469 1 N – USA/2009 in close collaboration with US research institutions involved in the US Department of Energy (DOE) funded grant under award number DE-FG02-04ER46136 and the Board of Regents, State of Louisiana, under contract no. DOE/LEQSF(2004-07). The authors want to thank our colleagues at both CCT and PSNC for great ideas and discussions. The authors want to thank Soon-Heum Ko from the National Supercomputing Center at Linköping in Sweden for helping validating the CFD code. This research was supported in part by PL-Grid Infrastructure. This work used also the computer resources provided by LSU/LONI, and we wish to extend special thanks for the hard work and support we received from the HPC staff.

References

[1] G. Allen, T. Goodale, G. Lanfermann, T. Radke, D. Rideout and J. Thornburg, *Cactus Users' Guide*, 2004, available at: http://www.cactuscode.org/Guides/Stable/UsersGuide/UsersGuideStable.pdf.

[2] G. Allen, T. Goodale, F. Löffler, D. Rideout, E. Schnetter and E.L. Seidel, Component specification in the Cactus framework: the Cactus configuration language, in: *Grid2010: Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, 2010, arXiv:1009.1341.

[3] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: *AFIPS Conference Proceedings*, Atlantic City, NJ, April 18–20, 1967, ACM, New York, 1967, pp. 483–485.

[4] M. Blazewicz, S.R. Brandt, P. Diener, D.M. Koppelman, K. Kurowski, F. Löffler, E. Schnetter and J. Tao, A massive

data parallel computational framework on petascale/exascale hybrid computer systems, in: *International Conference on Parallel Computing*, Ghent, Belgium, 2011.

[5] M. Blazewicz, K. Kurowski, B. Ludwiczak and K. Napierala, High performance computing on new accelerated hardware architectures, *Computational Methods in Science and Technology*, Special Issue (2010), 71–79.

[6] S.R. Brandt and G. Allen, Piraha: a simplified grammar parser for component little languages, in: *2010 11th ACM/IEEE International Conference on Grid Computing*, 2011.

[7] Cactus Computational Toolkit, available at: http://www.cactuscode.org.

[8] U. Ghia, K.N. Ghia and C.T. Shin, High-re solutions for incompressible flow using the Navier–Stokes equations and a multigrid method, *Journal of Computational Physics* **48**(3) (1982), 387–411.

[9] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel and J. Shalf, The Cactus framework and toolkit: Design and applications, in: *Vector and Parallel Processing – VECPAR'2002, 5th International Conference*, Lecture Notes in Computer Science, Vol. 2565, Springer-Verlag, Berlin, 2003.

[10] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel and J. Shalf, The Cactus framework and toolkit: Design and applications, in: *High Performance Computing for Computational Science – VECPAR 2002, 5th International Conference*, Porto, Portugal, June 26–28, 2002, Springer-Verlag, Berlin, 2003, pp. 197–227.

[11] J.L. Gustafson, Reevaluating Amdahl's law, *Communications of the ACM* **31** (1988), 532–533.

[12] C.W. Hirt and B.D. Nichols, Volume of Fluid (VOF) method for the dynamics of free boundaries, *Journal of Computational Physics*, 1981.

[13] D.A. Jacobsen, J.C. Thibault and I. Senocak, An MPI-CUDA implementation for massively parallel incompressible flow computations on Multi-GPU clusters, American Institute of Aeronautics and Astronautics, 2010.

[14] K. Kurowski, B. Ludwiczak, J. Nabrzyski, A. Oleksiak and J. Pukacki, Dynamic grid scheduling with job migration and rescheduling in the gridlab resource management system, *Sci. Program.* **12** (2004), 263–273.

[15] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, Technical report, NVIDIA, 2009.

[16] A. Munshi (ed.), The OpenCL specification, version 1.1, The Khronos Group, 2011.

[17] NVIDIA Corporation, *OpenCL Programming Guide for the CUDA Architecture*, 2010.

[18] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, 2011.

[19] D.A. Orchard, M. Bolingbroke and A. Mycroft, Ypnos: declarative, parallel structured grid programming, in: *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP'10*, 2010, pp. 15–24.

[20] J. Tao, M. Blazewicz and S.R. Brandt, GPGPU stencil computation kernel abstraction and implementation for the development of large scale scientific applications on hybrid systems, in: *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 25–29 February, 2012, New Orleans, LA, USA, 2011, submitted.

[21] J.C. Thibault and I. Senocak, CUDA implementation of a Navier–Stokes solver on multi- GPU desktop platforms for incompressible flows, American Institute of Aeronautics and Astronautics, 2009.

[22] Top 500 Supercomputer Sites.

[23] M.D. Torrey, L.D. Cloutman, R.C. Mjolsness and C.W. Hir, NASA-VOF2D: a computer program incompressible flows with free surfaces, Technical report, Los Alamos National Laboratory, 1985.

[24] D. Unat, X. Cai and S.B. Baden, Mint: realizing CUDA performance in 3D stencil methods with annotated c, in: *Proceedings of the International Conference on Supercomputing, ICS'11*, 2011, pp. 214–224.