SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms

Tobias Brandvik and Graham Pullan
Whittle Laboratory, Department of Engineering
University of Cambridge
Cambridge, UK
tb302@cam.ac.uk, gp10006@cam.ac.uk

***Abstract*—We present a new software framework for the implementation of applications that use stencil computations on block-structured grids to solve partial differential equations. A key feature of the framework is the extensive use of automatic source code generation which is used to achieve high performance on a range of leading multi-core processors. Results are presented for a simple model stencil running on Intel and AMD CPUs as well as the NVIDIA GT200 GPU. The generality of the framework is demonstrated through the implementation of a complete application consisting of many different stencil computations, taken from the field of computational fluid dynamics.**

***Keywords*-Partial differential equations; GPU; CUDA**

I. Introduction

The switch to multi-core has led to a processor landscape with many competing designs that are radically different from each other. Different trade-offs can be seen in all the important features of current mainstream processors, including core heterogenity, the use of parallel or in-order instruction processing, the on-chip memory hierarchy and the vector length of the arithmetic units themselves.

For software developers, the variety in processor designs poses a significant challenge. Not only should applications try to make efficient use of multiple on-chip cores, but it is also desirable to target as many different processors as possible, preferably without making changes to the source code in order to accommodate their different characteristics.

In this paper, the problem is considered in the context of applications that rely heavily on stencil computations on three-dimensional structured grids. We present a software framework called SBLOCK that eases the implementation of such applications for clusters of multi-core processors. The framework is capable of producing efficient stencil implementation for several different processors through automatic source code generation.

Results from performance benchmarks of the Intel Nehalem, AMD Opteron and NVIDIA GT200 processors are shown. In addition, we present a case study of a large-scale application taken from the field of turbomachinery computational fluid dynamics (CFD). As well as demonstrating the generality of the framework, the case study also shows the impact that the performance of modern multi-core processors can have on the design of real-world turbomachinery.

A. Stencil Computations

Stencil computations are an important building block for many scientific applications. In particular, they are frequently used in fields where the relevant partial differential equations (PDEs) can be solved using finite difference, finite element or finite volume techniques on structured grids. Examples of such fields are numerous and include computational finance, fluid dynamics and general relativity.

As an example, we consider the three-dimensional heat diffusion equation:

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T, \tag{1}$$

where T is temperature, t is time and α is a constant.

This equation can be solved on a structured grid with uniform spacing by a Jacobi iteration and a centred finite difference approximation to the derivative:

$$\begin{aligned} T^{n+1}_{i,j,k} = T^n_{i,j,k} + \frac{\alpha \Delta t}{\Delta l^2}(&T^n_{i+1,j,k} - 2T^n_{i,j,k} + T^n_{i-1,j,k} + \\ &T^n_{i,j+1,k} - 2T^n_{i,j,k} + T^n_{i,j-1,k} + \\ &T^n_{i,j,k+1} - 2T^n_{i,j,k} + T^n_{i,j,k-1}) \end{aligned} \tag{2}$$

where Δt is the time-step, Δl is the grid spacing, the subscripts refer to the grid node and the superscripts refer to the time. Implementing the finite difference approximation on a computer leads to a stencil computation, in which the output value at every grid node is a linear combination of the values at that node and the nodes surrounding it in space. A naive implementation of the stencil computation would consist of three nested loops, one for each grid direction, with the inner loop containing the stencil computation itself:

$$\begin{aligned} Y[i,j,k] = &AX[i,j,k] + B(X[i-1,j,k] + X[i+1,j,k] + \\ &X[i,j-1,k] + X[i,j+1,k] + X[i,j,k-1] + \\ &X[i,j,k+1]) \end{aligned} \tag{3}$$

where X and Y are three-dimensional arrays, and A and B are scalar constants.

B. Stencil-based PDE solvers

We will use Eqn. 3 as our model stencil computation throughout this paper as it is one of the simplest stencils that still correspond to a physical problem. However, most problems of interest are more complicated than this simple diffusion case. Usually, several variables (e.g. mass, momentum and energy) and additional physical phenomena (e.g. convection) are involved. Solving such problems requires a series of stencil computations with multiple inputs and outputs, as well as a series of different boundary conditions. In addition, if the problem involves a large range of length scales, the use of additional algorithms such as multigrid is often required. As a final difficulty, the underlying geometry can be both complicated and in relative motion, which requires multiple structured grids with some form of interpolation between them. An example of such a multi-block configuration is shown in Fig. 1, in which the volume around a turbine blade has been discretised using a topology consisting of six different blocks.

At the end of the paper, a new CFD solver called Turbostream that contains all the above complications is presented. The new solver is based on an older solver called TBLOCK that is implemented in Fortran 77. The TBLOCK solver is a typical example of many current multi-block structured grid PDE solvers. It was originally a scalar code that was based on previous solvers first developed in the 1970s and 80s. Around the turn of the century, it was parallised using MPI to take advantage of clusters of commodity single-core x86 processors. With the recent switch to multi-core, however, if faces several challenges. In particular, its parallelisation approach is coarse-grained and operates on the block-level by assigning a discrete number of blocks to each core. Although this approach scales well across the multiple cores of current CPUs, we expect that it will have insufficient granularity to to take advantage of future CPUs that will have tens of cores within a few years. There is also the practical issue of load-balancing and running out of blocks when the core number goes up, both of which requires an extra pre- and post-processing step in which the blocks are subdivided before the simulation and then joined back together afterwards.

An additional problem is that TBLOCK relies on the compiler to automatically vectorise the code to take advantage of single instruction multiple data (SIMD) instruction sets such as the Streaming SIMD Extensions (SSE) found on the x86 architecture, which often leads to unsatisfactory results. Finally, the solver is unable to take advantage of novel processors such as GPUs that require a more fine-grained approach to parallelism and the use of different languages and runtime libraries (e.g. NVIDIA's CUDA or OpenCL from the Khronos group). It is worth pointing out at this point that the challenges posed by the current class of GPUs in many ways mirror those that we expect to face from future generations of CPUs.

Although we have raised these issues in the context of the TBLOCK solver, they are to varying degrees also applicable to many other multi-block structured grid PDE solvers. In the authors' field of turbomachinery CFD alone, similar points could be made about most popular codes currently used in industry and academia.

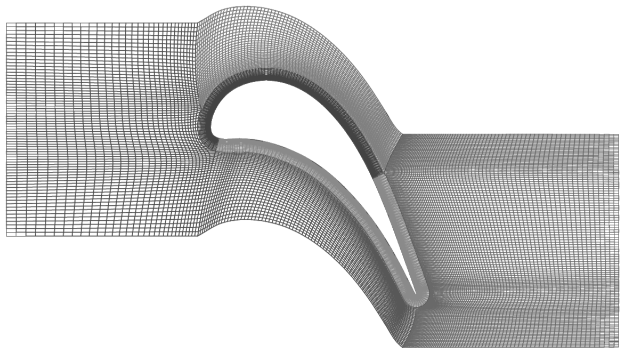


Fig. 1. A multi-block structured grid around a turbine blade.

C. Related Work and Contributions

The regular data access patterns of stencil computations, combined with their importance for scientific computing, has made them an attractive target for study by other researchers, and the software framework that we present in this paper uses many ideas from their work. For example, the run-time library component of SBLOCK is similar to the Distributed Array module in PETSc[1], or the structured grid driver in the Cactus framework[2]. A minor novelty in SBLOCK in this respect is its ability to work in either the conventional memory space of a CPU, or in the separate memory space of accelerators such as GPUs.

Regarding the stencil computations themselves, our work is most similar to Datta et al.[3], who presented the first optimized implementations of a stencil computation for a range of modern multi-core processors. They used a Perl script to generate implementations of the same model stencil that we consider in this paper, and produced versions of the script for each particular processor. Recently, Kamil et al.[4] have expanded this work into a more robust compiler framework, but this effort has so far only targeted simple kernels running on Intel and AMD CPUs. The source code generator in our work extends that of Kamil et al.[4] by tackling complicated stencils with an arbitrary number of inputs and outputs, and also handles branching statements in the inner loop. In addition, we support NVIDIA GPUs as well as Intel and AMD CPUs.

The main contribution of this paper, however, lies not with the individual parts of the software framework themselves, but rather in their combined application to a real-world PDE solver. To the authors' knowledge, the case study presented here is the first demonstration in the field of PDE solvers of a large-scale application that makes extensive use of automatic source code generation to achieve high performance on both conventional CPUs and accelerators such as NVIDIA GPUs.

II. Experimental testbed

The sections below detail the systems used for this work. The focus is on mainstream single-socket systems. Table I contains a summary of the properties of each system.

A. Processors

Intel Core i7 920 (Nehalem): The processor considered here is the 2.66 GHz variant in Intel's Core i7 line-up of consumer CPUs. It uses a native quad-core architecture, containing four separate cores on a single die. Each core has a 64 KB L1 cache and a 256 KB L2 cache; there is also a single 8 MB L3 cache shared between all the cores. The cores have access to the main memory through three DDR3-1066 memory controllers, giving a total bandwidth of 25.6 GB/s. Each core has an addition and multiplication unit for 128-bit vectors, giving a peak double precision floating point capability of 42.6 GFLOP/s (85.1 GFLOP/s in single precision). In addition, each core supports two-way multithreading, enabling the processor to support 8 separate thread contexts.

AMD Phenom II X4 940: This processor is the 3.0 GHz variant in AMD's consumer range of quad-core CPUs. It is similar in design to the Intel Nehalem in having four separate cores on a single die. Each core has a 64 KB L1 cache and a 512 KB L2 cache; the cores also share a 6 MB L3 cache. In comparison to the Nehalem, the Phenom II has only two DDR3-1066 memory controllers, giving a total DRAM bandwidth of 17.1 GB/s. The floating point capability is similar, with the 128-bit addition and multiplication units giving a peak double precition performance of 48.0 GFLOP/s (96.0 GFLOP/s in single precision).

NVIDIA GTX 280 (GT200): The GT200 is the latest graphics processing unit (GPU) from NVIDIA. It consists of 30 multiprocessors (MP), each of which contain 8 scalar processing units (SP) and 16 KB of explicitly managed local storage (referred to as shared memory). Each MP has its own instruction counter and operates independently of the others. Each SP can schedule one multiply and one single-precision multiply-add operation per cycle, giving a peak performance 933.1 GFLOP/s at 1.296 GHz. There is only 1 DP unit per MP, resulting in a much smaller peak performance of 77.8 GLOPS/s in double precision. By using a wide 512-bit bus to the external GDDR3 memory, the GT200 achieves a maximum bandwidth of 141.7 GB/s.

B. Cluster

All multi-processor benchmarks were done on the University of Cambridge's GPU cluster. It consists of 32 Dell Precision T5500 servers with dual Intel Xeon 5550 2.66 GHz CPUs, connected through PCI-Express 2.0 to 32 NVIDIA Tesla S1070 GPU units, for a total of 64 CPUs and 128 GPUs. The servers communicate with each other using Mellanox QDR Infiniband.

III. Framework Design

The SBLOCK framework consists of two components: a run-time library and a source code generator. Since these two components have only minimal interaction with each other, we shall consider them separately.

A. Run-time Library

The run-time library provides an API that is used by the application for four different functions:

- memory management;
- calling stencil kernels provided by the code generator;
- MPI communication between processors in a cluster;
- reduction operations and parallel sparse matrix-vector multiplications (SpMV).

These functions operate on a data structure containing multiple three-dimensional blocks. Each block is surrounded by layers of so-called halo nodes, which can be of any size (the number of layers required increases with the size of the stencils used by the application). The use of halo nodes allows the blocks to appear as a single interconnected grid and is commonplace for stencil codes.

The run-time library is separated into two parts: the *host* library and the *device* library. The former is implemented in C and runs on the host which is always a traditional CPU. The latter is implemented in C if running on a CPU or in NVIDIA's CUDA language if running on a GPU. The application, which always runs on the CPU, interfaces with the run-time library primarily through the host library, but in certain cases it can make calls directly to the device library if required for performance. This host-device abstraction is useful since only a new device library has to be written to support a new type of processor – the host library which contains most of the control logic (such as that for MPI communication) always remains the same.

The control flow for a typical application using SBLOCK is as follows. At the start of execution, the program must declare to SBLOCK the size and structure of the blocks and halos, as well as how many data arrays it requires. The arrays contain one value for each node and can be of the C types char, int, float or double. In addition, it is possible to declare scalar values that are associated with either the whole grid or the individual blocks.

SBLOCK will then allocate the required memory on the host and the device, and after initializing this memory the application will usually start to invoke kernels. A kernel takes as its arguments the identifiers corresponding to the arrays and scalars. After a series of kernel calls, it is normally necessary to set the boundary conditions on the grid or to update the halo nodes. SBLOCK handles the latter task, automatically transferring data using MPI if the relevant blocks are on different processors, but the former is the responsibility of the application. Typically, the application will ask SBLOCK for the array values of a subset of a block. The application then computes the updated values corresponding to the boundary condition in question, and passes them back to SBLOCK. A potential problem occurs in this case when running on accelerators such as GPUs. The transfer of the array values has to occur across the PCI-Express bus, and this process can be a performance bottleneck. If, after profiling the application,

System	Intel CPU	AMD CPU	NVIDIA GPU
Processor	Core i7 920	Phenom II X4 940	GTX 280
No. cores	4	4	30
Max no. threads	8	4	1024
Clock freq. (GHz)	2.67	3.00	1.30
Peak GFLOP/s (DP/SP)	42.6/85.1	48.0/96.0	933.1/77.8
DRAM BW (GB/s)	25.6	17.1	141.7
Processor power* (Watts)	130	95	165
OS	CentOS 5.3	CentOS 5.3	CentOS 5.3
Compiler	gcc 4.1.2	gcc 4.1.2	nvcc 2.3

TABLE I
DETAILS OF SYSTEMS USED. (*BASED ON THE THERMAL DESIGN POWER FROM MANUFACTURERS' DATASHEET.)

the developer decides that a bottleneck is indeed present, two options are available. If the boundary condition can be expressed in the form of a SpMV, then the application can use SBLOCK's SpMV library directly, which will not transfer any data across the PCI-E bus unless it needs information from another processor. Alternatively, the application may bypass the host library and request a pointer to the array directly from the device library, and implement its own functions in NVIDIA's CUDA language to operate on this array.

Finally, at the end of a series of kernels and boundary conditions, the application will often need to perform reductions on the data stored at the grid nodes to work out statistics for the progress of the algorithm (e.g. the average change of the physical properties). Such operations can be performed efficiently by using the reduction functions provided by SBLOCK. On the CPU, SBLOCK uses its own implementations of reduction operations, while on the GPU those implemented by the Thrust library[5] are used.

B. Source Code Generation

From the point of view of an application developer, a stencil kernel is a single file written in the Python scripting language. The file defines the inputs and outputs of the kernel, as well as the expressions that constitute the computation performed by it. At compile-time, the definition is passed through the source code generator which outputs source code that can be further compiled into object code for the target processor.

The source-to-source compilation performed by the code generator uses the Cheetah templating system[6]. Cheetah is most commonly used to insert dynamic content into predefined HTML templates for web pages, but is equally applicable to automatic source code generation. The underlying idea is to create a template which contains the static content of a kernel implementation - i.e. everything that is common for all kernels on a particular device. In addition to this static content, small snippets of code written in the Cheetah templating language are also included. These take information from the kernel definition and insert them into the appropriate places to generate a valid source file that can be compiled into machine code. A different template has to be created for each different platform - i.e. there is one template that produces normal C code for multi-core CPUs and another that produces CUDA code for NVIDIA GPUs.

While simple, this approach achieves two important goals:

1) It enables multiple platforms to be supported using only a single, high-level definition of the computations performed by the solver.
2) Since the actual kernel implementation is hidden from the developer, SBLOCK is free to use any optimization strategy it wants without worrying about code readability.

C. Optimization Strategies

The optimization strategies used by the framework are a subset of those studied extensively by Datta et al.[3], so only a brief overview will be given here. For the CPU, the primary way of achieving high performance is though the use of domain decompositioning and multiple threads. We decompose the block-arrays into "sub-block" along the second and third block axis, but do not do so along the unit-stride direction to avoid interfering with the hardware prefetcher. The sub-blocks are distributed to the threads first along the second and then along the third axis so that each thread receives the sub-blocks that are closest to each other in the array. The threading implementation is POSIX threads. The code generator is also capable of producing vectorised code that takes advantage of the CPU's SSE units. This part of the generator requires the creation of an abstract syntax tree representation of the stencil definition, and this task is accomplished using the freely available *pycparser*[7] Python module. Currently, we do not make use of the SSE non-temporal store instruction (*movntps*), as we have found that it tends to reduce the performance of kernels with multiple output arrays (probably due to running out of space in the write-combining buffers).

As for the Nehalem and the Phenom, the main optimization strategy for the GT200 is the use of domain decompositioning and multi-threading. However, since the GT200 has a larger number of cores with smaller on-chip memories, the number of sub-blocks is much larger. In addition, the GT200's on-chip memory is completely managed by software – there is no automatic caching. To maximize the amount of reuse of data in the on-chip memories, we therefore use the "cyclical queue" strategy described by Williams et al.[8] for the IBM Cell processor. In the context of the GT200, this procedure involves starting one thread for each grid node in a plane of

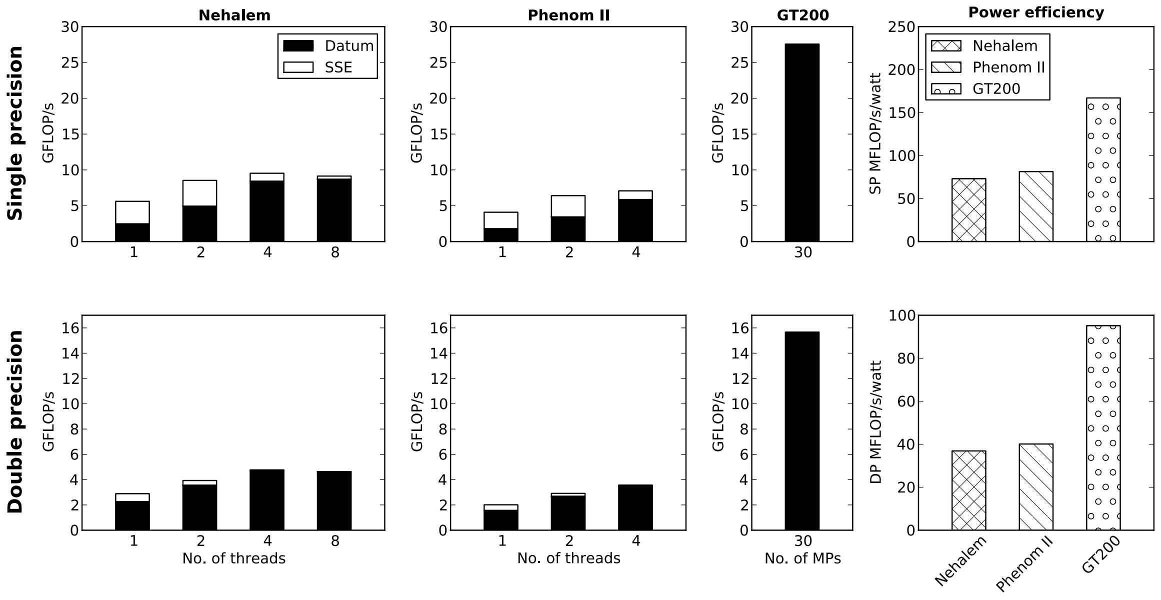


Fig. 2. Performance for model stencil in single and double precision.

a sub-block. These threads then iterate upwards in the sub-block together, each time discarding one plane of data and fetching a new one to be stored in its place. At each iteration, the threads compute the specified stencil operation using data stored in the on-chip memory planes and write out the result.

IV. Numerical Experiments

To demonstrate the effectiveness of the kernel generator, we first consider its performance for the model stencil in both single and double precision (Fig. 2). The grid used consists of a single 256^3 block, which is much larger than what can fit in the on-chip memory of any of the processors. We aim primarily to show the effectiveness of multi-threading by comparing the performance of the parallelised version to the scalar case. For the two CPUs, the impact of the use of SSE is also shown. It should be noted that for the GT200 there is no simple scalar implementation due to the multi-threaded nature of the CUDA programming model and the use of software-managed caches, so only the parallel case is shown. The power efficiency of each processor is also shown. Here, we have chosen to simply use the thermal design power (TDP) as quoted by the manufacturers themselves, with the understanding that the effective real-world power consumption of a whole system is a factor of many other variables that we do not account for (e.g. idle power vs. peak power, cooling requirements and power for memory modules).

A. Model stencil performance

Comparing the scaling across cores for the non-SSE CPU implementations, we achieve a performance increase of around 200-230% for both single and double precision when using all available cores as compared to the single-core case. This result comes from utilising more of the available bandwidth when running on multiple cores. The effect of SSE vectorisation is significant if there is enough bandwidth available to make the stencil compute-bound. However, since the total bandwidth utilisation does not scale linearly with core count, SSE is less effective when using multiple threads.

The top performance of the Nehalem is 30% higher than that of the Phenom II, which is in accordance with the difference in the processors' peak bandwidth. The performance advantage of the GT200 is in the range of 300-440% of the CPUs, which comes as a result of its unrivalled bandwidth of 141.7 GB/s. In terms of power efficiency, the two CPUs are similar while the GT200 is better by around a factor of two, which is lower than its raw performance advantage because it also consumes more power.

For all processors, the performance penalty of using double precision instead of single precision is significant: a 98% reduction is seen for the fully threaded and SSE-optimised CPU implementations, and 76% for the GPU implementation. This result demonstrates the importance for applications to only used double precision if it is necessary.

V. Application Case Study

The application considered here is a solver for compressible flows that uses block-structured grids. The solver is called Turbostream[9] and is a re-implementation using the SBLOCK framework of an older solver called TBLOCK which was originally developed by Denton[10]. The solvers are aimed primarily at predicting flows in turbomachines, but are also applicable to other types of flow such as that around wings and propellers. The TBLOCK solver is widely used in both industry and in academia, and is the latest in a long line of previous codes by Denton that are known collectively as

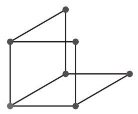
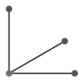
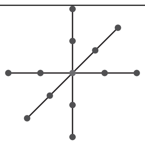
Kernel	Stencil	SP FLOPS	Bytes	AI	Performance (SP GFLOP/s, GB/s)						Power efficiency (SP MFLOP/s/watt)		
					Nehalem		Phenom II		GT200		Nehalem	Phenom II	GT200
Convective fluxes		260	139	1.87	10.7	5.71	5.62	3.00	71.7	38.3	82.2	59.2	434
Flux summation		8	28	0.29	3.55	12.2	2.83	9.76	13.6	47.0	27.3	29.8	82.7
Artificial diffusion		242	44	5.5	16.92	3.08	10.75	1.95	99.4	18.1	130	113	602

TABLE II
PROPERTIES AND PERFORMANCE OF THE MOST IMPORTANT STENCILS IN THE SOLVER.

the "Denton codes". A complete description of the algorithm used by the solver is given by Klostermeier[11] while shorter overviews and examples of its application to turbomachinery research have been published by Reid et al.[12] and Rosic et al.[13]. In addition, the motivation for the current method can be traced through a series of papers by Denton[14][15][16]. Here, we give a basic overview of the original solver, focusing particularly on the aspects relevant to its re-implementation using the SBLOCK framework.

A. Solver overview

Turbomachines, whether for propulsion of for power generation, consist of many rows of airfoils (referred to as "blades"). Typically, alternating rows are either stationary or rotating, and each row has between 10 and 200 blades. The aim of the flow solver is to predict the flow between the blades by solving the compressible Navier-Stokes equations, so that more efficient blades can be designed. The solver uses a finite volume approach in which the space between the blades is divided into many small hexahedra (referred to as "cells"). The algorithm starts with an initial guess of the flow and marches forward in time, each time evaluating the equations for each cell. This process is done by evaluating the fluxes of mass, momentum and energy through the faces of the cell, and hence computing a change in these properties for that cell.

Conceptually, we split the kernels of the algorithm into two different groups. The first group are the stencil kernels - these fit neatly into the SBLOCK framework and are handled by the SBLOCK source code generator. The second group contains kernels with more complicated data access patterns - we call these "non-stencil" kernels. We describe both of these groups in the two following sections. It should also be noted here that the stencil and the non-stencil kernels, both in the original TBLOCK implementation and in the new Turbostream implementation, use only single precision.

B. Stencil kernels

The main stencil kernels are those that evaluate the fluxes through the faces of each cell; these fluxes have two components – convective and diffusive. The diffusive flux is further split into two parts, one is due to physical diffusion while the other is due to "artificial diffusion" that is added to stabilize the numerical scheme. Once the fluxes through each face are set, another kernel is then used to sum the fluxes through the faces of each cell.

1) Stencil kernel characteristics: In the following discussion, we will consider the kernels for the convective fluxes, flux summation and the artificial diffusion. Although the full algorithm also requires other stencils, these three make up approximately 60% of the total run-time. Table II summarises the properties of each stencil; we focus on the amount of arithmetic (flops) and memory traffic (bytes) required by each kernel, as well as the arithmetic intensity (AI) which is the ratio of flops to bytes. The stencil access patterns themselves are also shown. By comparing the arithmetic intensity to the peak FLOP/s and peak GB/s offered by each processor (see Table I), we expect all the kernels to be memory-bound on all processors. The one possible exception is the artificial diffusion kernel which has a high AI of 5.5, which is slightly higher than the flops/byte ratio of the two CPUs. However, the large size of the stencil requires additional memory bandwidth, so we still expect it to be memory-bound.

2) Stencil kernel performance: Table II also lists the performance and power-efficiency of each kernel. As pointed out by Kamil et al.[4], the maximum possible performance of a memory-bound kernel is the product of the kernel's arithmetic intensity and the peak bandwidth of the processor. In reality, the useful bandwidth will be lower than the peak due to the extra halo nodes that have to be fetched for each sub-block. Since the number of halo nodes grows with the size of the stencil, we expect large stencils to have a worse bandwidth utilisation than small ones, and this is indeed the trend seen in Table II. The flux summation kernel, which has the smallest stencil, achieves the highest useful bandwidth of the three kernels, obtaining 12.2 GB/s on the Nehalem, 9.76 GB/s on the Phenom II and 47 GB/s on the GT200. The convective flux kernel has a slightly larger stencil, and consequently a lower bandwidth. Finally, the artificial dissipation kernel achieves the lowest bandwidth of all three, but also has a much larger stencil than the others.

Comparing the three processors against each other, the results echo those for the model kernel. The main difference is that the GT200 has a larger performance advantage over the CPUs for the real solver kernels than for the model kernels.

C. Non-stencil kernels

The non-stencil kernels include both boundary conditions and the multigrid routines.

1) Boundary conditions: Here we focus on the physical boundary conditions that have to be implemented by the developer, and not the exchange of halo nodes between blocks which is automatically handled by SBLOCK.

The four most often used boundary conditions in the solver are as follows: inflow, outflow, mixing plane, sliding plane. The first two conditions are straightforward to implement as they involve only the flow variables that are local to the face of the block that the boundary condition is being applied to. Since these boundary conditions are not computationally expensive, we only maintain a single implementation for the CPU that is used regardless of whether the stencil kernels run on a CPU or a GPU.

The latter two conditions are more complicated. Both mixing planes and sliding planes are used to interface blocks that are part of adjacent blade rows in relative motion. Mixing planes are used when performing so-called “steady-state” solutions in which the flow is not changing in time and each blade row sees an average of the flow in the next blade row. The way in which the flow is averaged is not straightforward, so we implement this boundary condition only on the CPU. If the next row is on another processor, MPI is used to get the necessary information from it. Sliding planes are used for time-accurate simulations, in which the flow is interpolated directly from one row to another. Since no averaging is necessary, we precompute the weights used in the interpolation for each grid node and store them as a sparse matrix. In this way, we can use SBLOCK’s optimised SpMV functions which increases both performance and portability.

2) Multigrid: Multigrid is an algorithm that can be used to great effect for problems involving both short and long length scales. Although a full introduction to the algorithm is beyond the scope of this paper, a brief overview is given below.

The idea for the multigrid algorithm originates from the observation that most solvers are good at reducing the high-frequency errors in a solution, but poor at reducing low-frequency errors. The technique used in the multigrid algorithm is to transfer the solution from the original fine grid to a coarser one, thereby increasing the effective frequency of the original low-frequency errors which leads to better convergence of the solver. From the point of view of the developer, using multigrid means doing frequent transfers from a fine grid to a coarse grid (called “restriction” in the multigrid literature) and then back again (“interpolation”). Rather than maintaining separate implementations of these transfer operators for each processor, we use the SpMV functions provided by SBLOCK in the same way as for the sliding planes.

Processor	TBLOCK performance	Turbostream performance
Nehalem	1.21	1.48
Phenom II	1	0.89
GT200	-	10.2

TABLE III
OVERALL PERFORMANCE OF SOLVERS, RELATIVE TO TBLOCK ON PHENOM II.

D. Overall performance

Table III shows the overall performance of the new Turbostream compared to that of the old TBLOCK solver. We have chosen to normalise all results by the slowest TBLOCK performance, which is that on the Phenom II. The most important metric is the speed-up of Turbostream running on the GT200 compared to that of TBLOCK running on the two CPUs. Compared to the Phenom II, Turbostream has a performance increase of 10.2, while compared to the Nehalem the performance increase is 8.42. This speed-up is significant for the application of the solver to practical design work. In typical usage, obtaining the flow field around a single turbine blade takes around an hour using the TBLOCK solver on a single CPU. Using Turbostream on the GT200, the time to solution is reduced by roughly an order of magnitude, changing it to just a few minutes.

Turbostream running on the two CPUs achieves comparable performance to the TBLOCK solver. This result is mainly a demonstration of the ability of current quad-core CPUs to offer good scaling across cores for traditional MPI-based PDE solvers using a block-based decomposition strategy. How long this will continue to be the case as CPU core counts continue to increase remains to be seen, but it seems likely that the threaded approach used by Turbostream will eventually win out.

VI. TEST CASE AND PARALLEL PERFORMANCE

In the last ten years, large-scale, time-accurate simulations with multiple blade-rows have become more common during the design process. Such simulations require more memory than that which is typically available on a single desktop, making the efficient use of clusters an important requirement of any solver. To demonstrate the effectiveness of the SBLOCK framework in such situations, both weak and strong scaling benchmarks have been performed across 64 GPUs on the University of Cambridge’s GPU cluster (Fig. 3). The benchmarks use a real turbomachinery simulation taken from

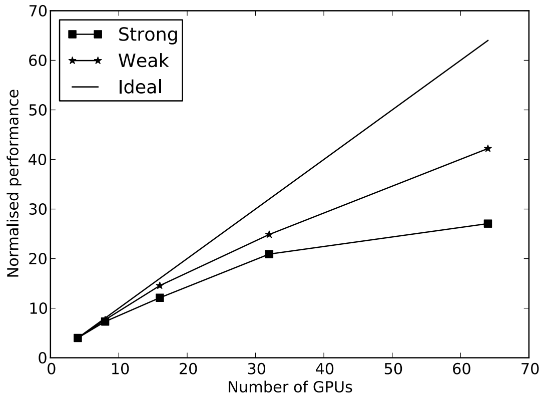


Fig. 3. Multi-GPU performance

Rosic et al. [13]. In its baseline configuration, the case contains 72 blocks and four million grid nodes, but can be scaled up as required - the largest case for the weak scaling on 64 GPUs contains 160 million grid nodes and 2880 blocks. As can be seen, the strong scaling tails off significantly after 16 GPUs. It should be noted that since we are using a real simulation, the load imbalance for the strong scaling in terms of grid nodes per GPU increases as we add more GPUs, resulting in a 20% imbalance at 64 GPUs. This penalty has not been corrected for, as it will inevitably be present to some degree for all real simulations. The weak scaling is adequate up to 16 GPUs, but more work remains to improve it for 32 and 64 GPUs.

VII. Conclusions

We have shown that a generalised software framework that combines the use of automatic source code generation with a runtime library can be used to implement stencil-based PDE solvers that run on a wide variety of processors, including accelerators. Good performance has been demonstrated for a range of stencil kernels with different stencil sizes and arithmetic intensities. Scaling across a large GPU cluster has also been shown, achieving good results up to 16 GPUs and continued scaling up to 64 GPUs.

By using the framework to re-implement an already existing CFD solver originally written in Fortran 77, speed-ups of around an order of magnitude were achieved for the new implementation running on an NVIDIA GT200 GPU as compared to the old implementation running on either a Nehalem or Phenom II CPU. Such a step change in performance has a significant impact on the engineering design cycle, in this case reducing the time-to-solution for typical turbomachinery simulations from an hour to just a few minutes.

Acknowledgment

The authors would like to thank NVIDIA for providing the GPU hardware used in this work through the University of Cambridge CUDA Centre of Excellence.

References

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[2] 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*. Springer, 2003.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Autotuning on State-of-the-art Multicore Architectures," in *Proceedings of Supercomputing 2008*, 2008.

[4] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, E. W. Bethel, and Prabhat, "A Generalized Framework for Auto-tuning Stencil Computations," in *Proceedings of Cray User Group Conference*, 2009, lBNL-2078E.

[5] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library," 2009, version 1.1. [Online]. Available: http://www.meganewtons.com/

[6] *The Cheetah Templating System*. [Online]. Available: http://www.cheetahtemplate.org/

[7] E. Bendersky, *pycparser*. [Online]. Available: http://code.google.com/p/pycparser/

[8] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific Computing Kernels on the Cell Processor," *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 263–298, 2007.

[9] T. Brandvik and G. Pullan, "An Accelerated 3D Navier-Stokes Solver for Flows in Turbomachines." ASME Paper GT2009-60052, 2009.

[10] J. D. Denton, "The Effects of Lean and Sweep on Transonic Fan Performance," *TASK Quart.*, pp. 7–23, 2002.

[11] C. Klostermeier, "Investigation into the capability of large eddy simulation for turbomachinery design," Ph.D. dissertation, University of Cambridge, 2008.

[12] K. Reid, J. D. Denton, G. Pullan, E. Curtis, and J. Longley, "The Interaction of Turbine Inter-Platform Leakage Flow With the Mainstream Flow," *ASME Journal of Turbomachinery*, vol. 129, no. 2, pp. 303–310, 2007.

[13] B. Rosic, J. D. Denton, and G. Pullan, "The Importance of Shroud Leakage Modeling in Multistage Turbine Calculations," *ASME Journal of Turbomachinery*, vol. 128, no. 4, pp. 699–707, 2006.

[14] J. D. Denton, "An Improved Time Marching Method for Turbomachinery Flow Calculation," *ASME Paper 82-GT-239*, 1982.

[15] ——, "The Use of a Distributed Body Force to Simulate Viscous Effects in 3D Flow Calculations," *ASME Paper 86-GT-144*, 1986.

[16] ——, "The Calculation of Three Dimensional Viscous Flows through Multistage Turbines," *ASME Paper 90-GT-19*, 1990.