EPSRC
Engineering and Physical Sciences Research Council

InFoMM
Industrially Focused Mathematical Modelling

EPSRC Centre for Doctoral Training in Industrially Focused Mathematical Modelling

Development of Multi-GPU Algorithms

Federico Danieli

UNIVERSITY OF OXFORD

NVIDIA

Contents

1. Introduction

Background

NVIDIA is a leading company in the design and development of *Graphic Processing Units* (GPUs). While traditionally employed in computer graphics applications, over the last decade graphic cards have increasingly been used as computing accelerators to boost the performance of software that requires a large number of computations to be carried out in parallel. The reason why GPUs are so appealing for *High Performance Computing* (HPC) applications lies in their specific architecture: the graphic processors are composed of thousands of cores capable of handling many thousands of concurrent operations.

The speed-ups achieved by employing GPUs provide relevant reduction in the computational time necessary to run algorithms for a wide range of applications. These span from the most classical areas of numerical simulations to the relatively recent ones of big data analysis and deep learning. Certain technologies currently being developed, such as self-driving cars and real-time action-recognition, would be unthinkable without the aid of graphic processors.

Parallel Computing on GPUs

To aid programmers in the task of writing general-purpose code designed to run on GPUs, NVIDIA developed CUDA: this is a particular model for parallel programming, as well as a programming language, that facilitates the dialog between CPU and GPU, and makes it simpler for software developers to port familiar structures employed in classical CPU programming to uses on GPUs. On top of this, a variety of software libraries written in CUDA provide basic routines for the implementation of common algorithms, such as linear algebra applications, on a single GPU.

With more GPUs being connected together efficiently in the same cluster, the capability of employing multiple devices to work concurrently on the same algorithm unlocks a new possibility for parallelisation

However, the number of GPUs present in the same system, as well as the communication speed among NVIDIA graphic cards, is ever increasing, with some of the most powerful computers (like *Piz Daint* in Switzerland and *Titan* in the U.S.) featuring a large number of graphic processors. NVIDIA has also launched a new product, DGX-1, which combines the power of 8 interconnected graphic cards and is designed specifically with applications to machine learning in mind. As a consequence, there is increasing interest in coordinating different devices to work simultaneously on the same task, unlocking an additional level of parallelisation, not only within the single GPU, but also across different GPUs. This practice goes under the name of *multi-GPU programming*.

Our aim is to investigate the extension of some simple algorithms for solving linear algebra problems to the framework of multi-GPU programming.

Glossary of terms

- CPU and GPU: These are acronyms for *Central Processing Unit* and *Graphic Processing Unit*, respectively. They are both key components responsible for interpreting and executing the instructions listed in a program, but while the first is a general purpose processor that acts as the "brain" of a computer, the latter sits on a graphic card and is specialised for tasks involving display functions and graphic rendering. This is reflected in its particular hardware architecture, designed to perform thousands of operations in parallel. In many systems, CPUs work in conjunction with GPUs, delegating to them certain tasks that are better suited to be performed by the latter.
- Kernel: This denotes any function (to be intended in the programming sense, as a sequence of instructions) written using the CUDA language and designed to run on an NVIDIA GPU.
- Thread, Warp, Block: This is the hierarchy of work distribution within a GPU. The thread is the simplest unit responsible for executing a series of operations sequentially. When launching a kernel, parallelisation is achieved by firing multiple

threads which work concurrently on different data. Threads are grouped in Warps of size 32: threads within a warp are implicitly synchronised among themselves. Warps are further collected in Blocks that can be of any size.

- Registers, Shared Memory, Global Memory: This is the hierarchy of memory types within a GPU. When launched, each thread has an area of memory assigned to it, (the Registers), which it can employ to store local variables and temporary values for computations. This memory is the fastest to access, and threads within a warp can quickly read from each other's registers, but its size is limited. Threads belonging to the same block can use Shared Memory to exchange data: this is slower to use than registers, but larger in size. Finally, threads of different blocks need to resort to Global Memory, the slowest and largest one, for their communications. Each device has its own global memory, and also the dialogue between CPU and GPU (as well as among GPUs, when supported) happens using this memory.

2. Solvers for Tridiagonal Systems of Equations

Tridiagonal systems are sets of equations of a particular kind which have a very simple structure and are therefore amenable to be solved in efficient ways. Although simple, they are ubiquitous in mathematical applications: they arise for example from the description of physical phenomena such as heat transfer and fluid flow. In these kind of systems, it is possible to order the equations in such a way that each of them depends only on three unknowns. As a consequence, each equation can be written as:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \qquad \forall i = 1, \dots, N,$$

where N is the number of equations in the system, and we assume $a_1 = c_N = 0$. If we collect the coefficients into a grid (or *matrix*) such that each row corresponds to an equation and each column to an unknown, then the only non-zero values are placed on the main diagonal (filled with the b_i, and running from top-left to bottom-right), as well as the ones above (the a_i) and below (the c_i), hence the name *tri-diagonal*.

The simple structure of tridiagonal systems makes them easy to solve. However, when designing algorithms to work on a GPU, we need to take into account the specific architecture of the device in order to make the best use of its computational power.

Given their simplified structure, many fast algorithms exist in the literature to solve them efficiently. The canonical choice is the *Thomas algorithm*, which consists of two parts: a *forward sweep* followed by a *backward sweep*. During the forward sweep, we proceed from top to bottom and combine each equation with the previous one to eliminate coefficient a_i. During the backward sweep, we proceed from bottom to top and combine each equation with the following one to eliminate coefficient c_i. At the end, we are left with a set of equations which are trivial to solve. A schematic of how the algorithm impacts the structure of the matrix is presented in Figure 1. This algorithm presents numerous advantages since it is extremely straightforward to implement and it does not require too many manipulations of the equations. However, its major drawback lies in its sequential nature: parallelisation can only be achieved if more matrices need to be solved at the same time, but at that point each thread should handle a whole matrix, which is too large to be contained in registers.

Figure 1: Schematic of the matrix structure when applying Thomas algorithm: original matrix (left), after the forward sweep (centre), after backward sweep (right). Each cross is a non-zero coefficient.

An alternative to the Thomas algorithm, which allows for parallelisation at a finer level (within the same matrix), is the *Parallel Cyclic Reduction* (PCR) algorithm. It operates in subsequent steps, where at each step each equation is combined with two others to

eliminate both coefficients a_i and c_i at the same time, at the cost of adding two more which are further away from the main diagonal. Eventually the off-diagonal coefficients are pushed "outside" the matrix, at which point a trivial system is recovered again, as shown in Figure 2. While this algorithm is more demanding in terms of the number of computations to be performed, each thread could take care of a single equation in the matrix, achieving fine-grained parallelisation. However, each step of the algorithm needs to be synchronised, since a single warp cannot take care of handling the whole matrix, and so the coordination between threads needs to be imposed explicitly, which comes at a cost. Moreover, threads in different warps need to exchange data using shared memory, which further impacts performance.

Figure 2: Schematic of the matrix structure when applying PCR algorithm: after first step (left), after second (centre), and at completion (right). Each cross is a non-zero coefficient.

To make the best use of the specific architecture of a GPU, a *Hybrid* algorithm is preferable, which combines the advantages of both the Thomas and PCR algorithms, while at the same time mitigating their drawbacks. In this hybrid algorithm, we divide the original matrix into chunks. As a first step, we apply the Thomas algorithm in the interior of each chunk, so that the equations in each chunk only depend on its first and last unknowns. As a second step, we collect the first and last equations of each chunk and the PCR algorithm is applied to them. Once PCR is completed, the values of the collected unknowns can be used to recover the remaining ones inside the chunk. The evolution of the matrix structure is presented in Figure 3. By assigning each chunk to a thread, the algorithm is flexible enough to allow for a whole matrix to be handled by a single warp. If that is the case, then there is no need to worry about explicit synchronisation, since this is automatically ensured. Moreover, the chunk size for the applications considered is usually small enough for it to fit within the registers.

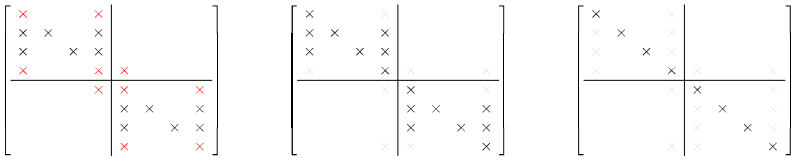

Figure 3: Schematic of the matrix structure when applying the Hybrid algorithm: after applying Thomas within the chunks (left), after applying PCR (centre), and after the final substitutions. Each cross is a non-zero coefficient. The red crosses on the left identify the sub-system PCR is applied to. The black lines identify the chunks the matrix is divided in.

Extension to Block-tridiagonal Systems

Block-tridiagonal systems are similar to their tridiagonal counterpart, and can be solved applying the same algorithms. However, both the computational cost and the memory requirements are larger in this case.

The structure of *block-tridiagonal systems* shares many similarities with that of tridiagonal ones. They arise from the same class of problems that tridiagonal systems come from, for example, in cases where more accurate approximations are chosen to simulate the evolution of a physical system.

Analogously to the tridiagonal case, the equations composing the system can be written as:

$$A_i \boldsymbol{x}_{i-1} + B_i \boldsymbol{x}_i + C_i \boldsymbol{x}_{i+1} = \boldsymbol{d}_i \qquad \forall i = 1, \dots, N/2,$$

where the sole difference lies in the fact that the variables are grouped into vectors and the variables' coefficients are matrices themselves of given size (they are the blocks in the name *block*-tridiagonal). For our applications, we limited ourselves to blocks of dimension 2x2, although similar considerations can be extended to larger sizes.

All three solvers we have discussed retain their structure even when applied to these kind of systems. However, what changes is the number of operations that need to be performed. In particular, scalar manipulations in the tridiagonal system have to be replaced by the corresponding matrix operations which are, in general, much more computationally intensive. On the one hand, this renders the use of a GPU even more appealing to carry on the solution of the system, but on the other hand it comes with a larger requirement for memory, given that now whole matrices need to be stored rather than single coefficients. However, for the applications we consider, a careful choice of the chunk sizes allows us to store all the relevant data within the registers.

3. Multi-GPU Algorithms

The Hybrid algorithm introduces a clear hierarchical structure in the distribution of work within a GPU: the single thread takes care of the operations within a single chunk, while the warp operates across chunks. This hierarchy can be extended to consider more than one GPU working on the task. For our applications, we considered an extension to the concurrent use of just two GPUs, although it can be easily scaled to an arbitrary number of GPUs.

The complete matrix is initially divided in two equal parts, with one GPU in charge of the top half and the other assigned to the bottom one. Within each GPU, the work proceeds as described in the normal hybrid algorithm, even though extra care needs to be dedicated to correctly treating the two "dangling" unknowns that act as a *bridge* between the two halves of the matrix, as shown in Figure 4. In general, the solution of the first half of the system depends on the value of the first unknown of the second half, and conversely for the second half. This introduces an additional complication, since the GPUs need to synchronise and exchange information among each other regarding this bridge system before they can complete their operations.

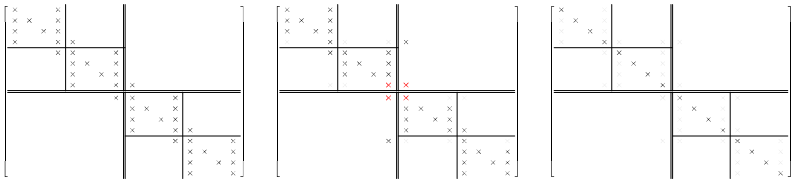

Figure 4: Schematic of the matrix structure when applying the multi-GPU Hybrid algorithm, after applying Thomas within the chunks (left), after applying PCR (centre), and after solving the bridge system and applying the final substitutions. Each cross is a non-zero coefficient. The red crosses in the centre identify the bridge equations. The black lines identify the chunks the matrix is divided in, while the thicker lines represent how the matrix is split among the two GPUs.

When using multiple GPUs to work on the same task, we need to worry about how to synchronise the devices. This can be achieved at CPU or at GPU level.

The most straightforward way to achieve the required coordination involves the CPU taking care of the synchronisation between the GPUs. In this case, the whole algorithm is split into two different kernels, one each performing the operations required before and after the solution of the bridge system, respectively. The CPU is responsible for firing the first kernel on both GPUs, waiting for it to complete on the devices, triggering the exchange of relevant information between them, and finally fire the second kernel and thus complete the algorithm. However easy this is to achieve, such implementation comes at a cost. Temporary data needs to be stored in the global memory of the devices at the end of the first kernel, and then read in again at the beginning of the second one. There is also more dialogue required between CPU and GPU, which is usually a bottleneck.

An alternative approach, which removes the necessity of storing temporary variables, consists in having the synchronisation happening directly at GPU level as part of the whole routine for the solution of the tridiagonal system. This can be achieved if each GPU has access to the global memory of the others, as is usually the case, especially in newest systems. It suffices to dedicate a specific area of memory to hosting a flag, which is triggered when one GPU has completed its task, in order to notify the other GPU that it is

now safe to read the necessary data that needs to be exchanged among them before proceeding with the algorithm.

Despite the increase in data that needs to be exchanged between GPUs in the block-tridiagonal case, the mechanism for synchronisation can be applied in the exact same way.

4. Comparison of the Algorithms

As a first experiment, we are interested in comparing the performances of the two different approaches to achieve synchronisation among the devices: at GPU level, using one single kernel, or at CPU level, using two. In Figure 5, we show the execution time for each algorithm for a variety of matrix sizes. In the case of the two-kernel synchronisation (2k in the figure), the total time is further broken down into computational time for each kernel, as well as that required by the CPU to achieve coordination.

Synchronisation at GPU level is to be preferred over the CPU one: experiments showed that the latter requires consistently much more time

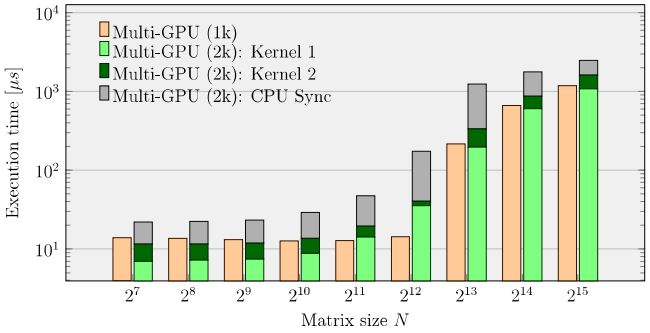

Figure 5: Comparison of total execution time of the solver when using GPU (orange) or CPU (green and grey) for synchronisation. For the latter, the time is further split to identify the cost of first kernel (light green), second kernel (dark green) and synchronisation at CPU level (grey).

From Figure 5, we can easily see how storing temporary results is detrimental to the performance of the algorithm, especially for matrices of larger size: the execution time required by the two-kernel approach is in fact consistently higher than the single-kernel counterpart. Moreover, the synchronisation on the CPU by itself takes most of the time. This clearly indicates that GPU-level synchronisation is the preferable choice.

Of particular interest is the comparison between the single-GPU and the multi-GPU implementations. The corresponding results are reported in Figure 6. By varying the matrix size, we can notice how using different types of memory impacts performance. With $N > 2^{12}$ and $N > 2^{10}$ for the tridiagonal (left) and block-tridiagonal (right) case respectively, the matrix is too large to fit on registers. This is reflected in Figure 6 by the sudden increase observed in the required execution time.

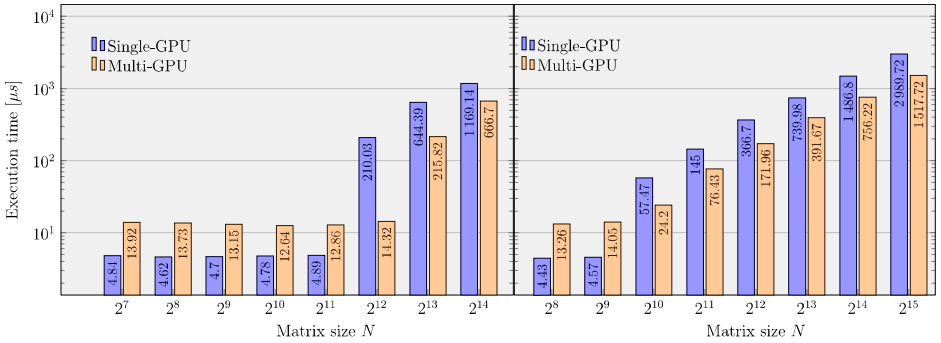

Figure 6: Comparison of total execution time of the solver: single- (blue) and multi-GPU (orange) implementation, for tridiagonal (left) and block-tridiagonal (right) systems.

The multi-GPU implementation is a viable alternative in case the matrix coefficients are already split among different devices

We also notice that, for smaller matrix sizes, the single-GPU implementation definitely outperforms the multi-GPU counterpart: this is due to the additional cost of the synchronisation among the devices. As a consequence, we deduce that the multi-GPU implementation does not provide an improvement in performance over single-GPU implementation for the general solution of tridiagonal systems. Nonetheless, for the applications we are interested in, the tridiagonal solver routine is embedded in a more complex software. In general, it is not unusual for the computation of the matrix coefficients to be a very demanding task, which is already distributed among different devices. If this is the case, a single-GPU approach would first require collecting all the data on the same device, perform the necessary computations, and then redistribute the solution to the other devices. This is not necessary with a multi-GPU implementation of the solver. The large amount of data transfers between GPUs greatly impacts the global performance of the algorithm, so a more fair comparison would keep this additional cost into account.

5. Discussion, Conclusions & Recommendations

We have implemented and tested various algorithms for the solution of tridiagonal and block-tridiagonal systems on a multi-GPU environment. We paid particular attention to investigating different techniques to achieve the necessary synchronisation among GPUs.

The results collected unmistakably point towards an implementation that favours direct dialogue between GPUs rather than relying on the supervision of the CPU for coordination. The former is beneficial since it removes the necessity of storing temporary data, and it reduces CPU interaction. We expect this result to become even more relevant as communication speed among GPUs improves.

Within the multi-GPU implementation, the additional task of synchronising the GPUs is very detrimental to the kernel execution time, which makes the single-GPU approach preferable in general. However, in applications where the tridiagonal solver is a subroutine of a more complicated code that already employs multiple GPUs, then this algorithm becomes a viable alternative. Its main strength lies in the preference towards local computations and the consequent small amount of communication that needs to be performed between GPUs.

6. Potential Impact

The majority of our analysis is relevant not only for the specific case, but also for implementations of other algorithms that rely on multi-GPU programming in general. Most of the solutions developed for our test case, in particular regarding synchronisation among GPUs are, in fact, versatile programming techniques commonly employed in many other applications as well.

The code we have developed is an initial proof of concept for tridiagonal solvers designed to work across multiple GPUs, and it requires tuning and extensions to tackle general cases before it can be embedded in real-world applications. Nonetheless, it represents a well-developed starting point and it provided numerous insights on the performance to be expected from this type of algorithms.

Thomas Bradley, Director of Developer Technology at NVIDIA, commented "*As computational models become more complex and problems become larger, High Performance Computing techniques are becoming increasingly important in myriad applications across all industries. The computing platforms we use to address these problems are increasingly parallel, and as such it is important that mathematical modelling takes into account the practicalities of implementing the algorithms efficiently. This work joins the mathematical formulation of a problem with the implementation, with both fine-grain parallelism within a GPU and coarser-grain across multiple GPUs. The methods can be applied not only to tridiagonal problems, which arise in many real-world models, but also in the development of related algorithms*".