GPU SOFTWARE AND ARCHITECTURE COMPARISONS FOR NUMERICAL SIMULATION OF PARTIAL DIFFERENTIAL EQUATIONS

JON B. MAY AND DONATO PERA

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila
via Vetoio (snc), Località Coppito, L'Aquila 67010, Italy

(received: 4 December 2017; revised: 26 December 2017;
accepted: 29 December 2017; published online: 15 January 2018)

Abstract: This paper will show a comparison between the Kepler, Maxwell and Pascal GPU architectures using CUDA-Fortran, with and without dynamic calls, to efficiently solve partial differential equations. The target is to show the possibility of using affordable hardware, such as the GTX670, GTX970 and GTX1080 NVIDIA GPUs, which are commonly found in personal and portable computers, for scientific applications. For simplicity we consider a standard wave equation where we use a second order finite difference method for the spatial and time discretizations to obtain the numerical solution. We found that, as we increase the spatial resolution of the domain we also increase the performance difference between the GPU and the Central Processing Unit (CPU).

Keywords: GPGPU, GPGPU, PDE

DOI: https://doi.org/10.17466/tq2018/22.1/c

1. Introduction

Mathematical models based on Partial Differential equations are often used to predict the behaviour or evolution of different problems, such as; cancer dynamics [1, 2], earthquake dynamics [3], and engineering and computer imaging problems [4]. Solutions to these problems can produce strikingly non-trivial patterns, therefore their numerical solution can often require a high spatial resolution to capture the detailed parameters related to the phenomena, and as a consequence, long computation times are often required when using a serial implementation of a numerical scheme. Parallel computation can dramatically improve the time and efficiency of some numerical methods such as finite difference algorithms, which are relatively simple to implement and apply to different models. For applied

scientists involved in setting up realistic experiments, the possibility of running fast comparable simulations using simple algorithms implemented on affordable processors is of primary interest, and that is where Graphical Processing Units (GPUs) can excel.

Parallel computing based on modern GPUs has the advantage of high performance at relatively low energy and monetary costs. In 2002, commodity graphic cards started to outperform Central Processing Units (CPUs). As GPUs grew faster and cheaper, the interest in harvesting their power for applications other than graphical display originated and, around 2006, what is known as GPGPU (General Purpose GPU computation, `http://gpgpu.org`) was born. By the year 2009 on market GPUs had a theoretical peak performance of more than a thousand single precision GFLOPs (10^9 floating point operations per second), almost ten times more than their multi-core CPU counterpart. Nowadays, GPUs found in personal computers and laptops can perform double precision computations with a ratio of speed over cost larger than any other parallel computing architecture. Additionally, GPUs are also energy efficient making them an affordable and portable option for parallel computation.

The codes used to study the performance of GPUs presented in this article were programmed using CUDA FORTRAN [5]. The CUDA platform (Compute Unified Device Architecture), introduced by NVIDIA in 2007, was designed to support GPU execution of programs and focuses on data parallelism [6]. With CUDA, graphics cards can be programmed with a medium-level language, that can be seen as an extension to C/C++/Fortran, without requiring a great deal of hardware expertise. We refer to [7] and [8] for a comprehensive introduction to GPU-based parallel computing, including details about the CUDA programming model and the architecture of current generation NVIDIA GPUs.

In this paper we give a performance comparison between the Kepler, Maxwell and Pascal GPU architectures. We use CUDA techniques to solve the wave equation, for which the numerical solution is obtained using a second order finite difference method for the spatial and time discretizations, and compare the speed up and efficiency of each code on each of the three architectures. The choice of time-explicit algorithms is due to their greater ease of implementation and performance, and despite their limitations related to reduced stability properties [1].

2. Mathematical model

The set of equations (1) show the mathematical model for the 2D wave equation that was used to compare the three GPUs,

$$\begin{cases} \frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), & (x,y) \in \Omega \ \& \ t \in (0,T] \\ u(x,y,0) = I(x,y), & (x,y) \in \Omega \\ \frac{\partial}{\partial t} u(x,y,0) = V(x,y), & (\mathrm{x},\mathrm{y}) \in \Omega \\ u = 0, & (x,y) \in \partial\Omega \ \& \ t \in (0,T] \end{cases} \tag{1}$$

where

$u = u(x,y,t)$ is the displacement,
$c = c(x,y)$ is the wave speed in the x and y directions,
T is the final time,
Ω represents the domain,
$\partial\Omega$ represents the boundary of the domain and
I & V are given functions for the initial displacement and velocity, respectively.

There are three commonly used types of finite difference methods; forward, backward and central differences. We will use the central differences method due to its stability and the fact that it is an explicit method. The central differences method, when applied to system (1), gives rise to the following scheme for the second derivative

$$\frac{u_{i,j}^{n+1} - 2u_{i,j}^{n} + u_{i,j}^{n-1}}{\Delta t^2} = c^2 \left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2} \right) \quad (2)$$

which can be rearranged to

$$u_{i,j}^{n+1} = -u_{i,j}^{n-1} + 2u_{i,j}^{n} + C_x^2 \left(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n} \right) + C_y^2 \left(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n} \right) \quad (3)$$

where $C_x = c_x \frac{\Delta t}{\Delta x}$ & $C_y = c_y \frac{\Delta t}{\Delta y}$ are the respective Courant numbers for the x and y directions. Equation (3) allows us to calculate the solution at position i,j at $t+1$. The stability of the explicit scheme rests on the value of these Courant numbers $C_x + C_y \leq C_{\max}$, the details of which will be shown in the following section.

2.1. Stability analysis

As stated, the stability of the model rests on the values of the Courant numbers, C_x and C_y, and in particular the following inequality

$$C_x + C_y \leq C_{\max} \quad (4)$$

We now show the stability analysis, performed on the Courant numbers; as $C_x = c_x \frac{\Delta t}{\Delta x}$, and C_y is equivalent in y we have

$$c_x \frac{\Delta t}{\Delta x} + c_y \frac{\Delta t}{\Delta y} \leq C_{\max} \quad (5)$$

One typically takes $C_{\max} = 1$, and therefore we are left with the following stability condition

$$c_x \frac{\Delta t}{\Delta x} + c_y \frac{\Delta t}{\Delta y} \leq 1 \quad (6)$$

In our domain we assume that the length of the domain in x and y is equal, $L_x = L_y$, and the number of elements in x and y is also equal, $EL_x = EL_y$. With these assumptions we have $\Delta x = \Delta y$. We also fix the final time, $T = 100$, and the number of time steps, tsteps $= 1000$, giving us $\Delta t = 0.1$, and use wave speed $C(x,y) = (1,1)$.

Using these assumptions we may reduce (6) to

$$0.2 \leq \Delta x \tag{7}$$

Our tests are run with the number of elements, EL_x and EL_y, taking the values 512, 1024, 2048, 4096 and 8192, in each test we fix $\Delta x = \frac{1}{4} = 0.25 \geq 0.2$ to satisfy the stability condition, we achieve this by requiring that $L_x = \frac{EL_x}{4}$.

2.2. Discretization of the physical domain

The discretization of the domain is rendered simple due to the constraints that we have placed on the variables in order to guarantee stability and the use of the finite difference method. Since EL_x and EL_y take the values 512, 1024, 2048, 4096 and 8192, and $L_x = \frac{EL_x}{4}$, we know that L_x must take the values 128, 256, 512, 1024 and 2048. As an example, in a domain $EL_x = EL_y = 512$ and $L_x = L_y = 128$ each element will have dimensions $\frac{1}{4} \times \frac{1}{4}$.

3. Performance evaluation

In this section we define the speed-up and the efficiency, two useful benchmarks for program performance.

Speed up is the ratio between the parallel and serial program times. According to the Amdahl's law [9] the speed-up is defined as follows:

$$S_p = \frac{S+P}{S+\frac{P}{M}} = \frac{1}{S+\frac{(1-S)}{M}} < \frac{1}{S} \tag{8}$$

where S and P are the serial and parallel sections of the program respectively, $(S+P=1)$, and M is the number of parallel processes run during a simulation. The theoretical limit for the speed-up is always equal to the number of parallel processes in which a given problem is divided. However, it is impossible to reach this limit due to the various hardware bottlenecks associated with computing architectures. In particular, the memory access time and data traffic will degrade the performance compared to the theoretical peak. Also the fact that most problems typically cannot be performed completely in parallel, generally some I/O operations or problem initialization will not be parallelizable, will further degrade the performance.

In order to obtain an estimate of the effectiveness of a given architecture it is useful to define the efficiency parameter. In our case we define the efficiency as the ratio between the speed-up and the number of processors available. Therefore, we have:

$$E_p = \frac{S_p}{p} \tag{9}$$

where S_p is the speed-up defined as above and p is the number of processors (or cores). As for the speed-up, there also exists a theoretical limit for the efficiency which is equal to one. However, the difficulty in reaching peak speed up means that having an efficiency of 1 is impossible.

In our problem to calculate the actual speed up we will proceed as follows:

The total program speed up:

$$S_{\text{pt}} = \frac{S_{\text{CPUtot}}}{S_{\text{CPU}} + P_{\text{GPU}}} \tag{10}$$

The CUDA Kernel speed up:

$$S_{\text{pk}} = \frac{P_{\text{CPU}}}{P_{\text{GPU}}} \tag{11}$$

Where

S_{CPUtot} is the total execution time of the program in serial on the CPU,
S_{CPU} is the execution time on the CPU of the serial part of the program,
P_{GPU} is the execution time on the GPU of the parallel part of the program (equal to the CUDA kernel(s) execution time(s)),

finally

P_{CPU} is the execution time in the CPU of the parallel part of the program.

Another important parameter to consider when evaluating the GPU performance is the memory bandwidth, which is defined as the rate at which the data can be transferred between the host and the device.

In our hardware-software configuration we have three NVIDIA GPUs;

- GTX670 (Kepler) with 1344 CUDA cores and 4Gb of RAM
- GTX970 (Maxwell) with 1664 CUDA cores and 4Gb of RAM
- GTX1080 (Pascal) with 2560 CUDA cores and 8Gb of RAM

Each is mounted on an HP DL585G7 4 AMD Opteron 6128, with 8 cores, clock frequency of 2.0 GHz and 64 Gb RAM, running OS Linux centOS 6.6 amd64. We compiled the programs using PGI 15.10 with NVIDIA CUDA 6.5 Linux 64 bit for the Kepler and Maxwell GPUs and PGI 16.7 with NVIDIA CUDA 8.0 Linux 64 bit for Pascal.

4. GPU architecture comparison

As stated, we have used three different NVIDIA architectures; Kepler, Maxwell and Pascal.

Kepler is a GPU microarchitecture developed by NVIDIA (2012), with a focus on energy efficiency, as the successor to the Fermi microarchitecture. NVIDIA's previous architecture was focused on increasing performance. With the Kepler architecture NVIDIA targeted their focus on efficiency, programmability and performance. The improvement in efficiency was achieved through the use of a unified GPU clock, simplified static scheduling of instruction and improved power management. Improvement in programmability was achieved with Kepler Hyper-Q, Dynamic Parallelism and multiple new Compute Capabilities 3.x functionality.

Maxwell introduces an all-new design for the Streaming Multiprocessor (SM) that dramatically improves power efficiency. Improvements to control logic partitioning, workload balancing, clock-gating granularity, instruction scheduling, number of instructions issued per clock cycle, and many other enhancements allow

the Maxwell SM to have improved performance compared to Kepler. The number of CUDA Cores per SM has been reduced to a power of two, however Maxwell has improved execution efficiency, performance per SM is usually within 10% of Kepler performance, and the improved area efficiency of the SM means CUDA cores per GPU will be substantially higher versus comparable Fermi or Kepler chips.

Pascal is the successor to the Maxwell architecture, it was released in April 2016. From a programming/hardware point of view the biggest leap is related to the dynamic call technique that is available from Maxwell GPUs. This technological improvement could lead to very large differences in final computing performance according to the potential reduction of time related to data transfer inside the computing systems. In this work we used the Kepler architectures on a GTX670 produced by ASUS, whereas we used the Maxwell and Pascal architectures on GTX970 and 1080 produced by Zotac, [10–13].

5. Software comparison

We implement two different versions of the same numerical algorithm for the solution of the wave equation through the use of the classical CUDA programming technique and CUDA programming with dynamic calls. We run the codes on NVIDIA Kepler, Maxwell and Pascal architectures using NVIDIA GTX670, GTX970 and GTX1080 graphics cards. In the following codes C_x and C_y are the wave speed in x and y, respectively, u is a 3D array of displacements, the third dimension represents time and is three elements deep as required by our chosen finite difference stencil. Therefore, the programs find $u(:,:,3)$ before updating the array by moving all information backwards in time and again solving $u(:,:,3)$. In the classical GPU code the solution is found on the device and then sent back to the CPU where the array u is updated before being returned to the device, this happens at every time step. We now report the most important parts related to the codes:

5.1. Classical GPU code

The classical GPU code requires some changes to the part of the program that is to be solved on a device. The code section below shows the subroutine used by each CUDA core to find the solution at the next time step. The single *if* statement is a result of the fact that each core will call the subroutine independently, and will only need to work on a single element each time, which is decided using the address of each individual CUDA core.

```
-----------------------------------------------------------------

GPU classical programming

-----------------------------------------------------------------

attributes(global) subroutine solve(u,Cx,Cy)
```

```
i = (blockidx%x-1)*blockdim%x + threadidx%x
j = (blockidx%y-1)*blockdim%y + threadidx%y
imax = size(u,1)
jmax = size(u,2)

if (i>1 .and. i<imax .and. j>1 .and. j<jmax) then

 d2tdx2 = u(i-1,j,2) - 2*u(i,j,2) + u(i+1,j,2)
 d2tdy2 = u(i,j-1,2) - 2*u(i,j,2) + u(i,j+1,2)

 u(i,j,3)= -u(i,j,1)+2*u(i,j,2)+(Cx**2*d2tdx2+Cy**2*d2tdy2)

end if

end subroutine
-----------------------------------------------------------------------
```

5.2. Dynamic GPU

The dynamic GPU code requires two further changes. Firstly the subroutine that updates u at each time step is re-written and moved onto the device in the same way as the solve subroutine is. Then, we need a new device kernel so that we can move the control of the time step counter onto the device. We achieve this by calling a subroutine (shown below) with only one CUDA core. This core then effectively works as the CPU counter in the classical GPU program. This allows us to move all of the work, and control, onto the device and negate any need to upload and download data, or send commands, at every time step, thus we now have a single upload and single download at the beginning and end of the program, respectively.

```
-----------------------------------------------------------------------

\acro{GPU} with dynamic calls
-----------------------------------------------------------------------

attributes(global) subroutine gpu_timer(u,Cx,Cy,tsteps)
tblock = dim3(32,8,1)
grid = dim3(ceiling(real(imax)/tblock%x),ceiling(real(jmax)/tblock%y),1)

do i = 1,tsteps-2
    call solve<<<grid,tblock>>>(u,Cx,Cy)
    call syncthreads()
    call update<<<grid,tblock>>>(u)
    call syncthreads()
end do
end subroutine
-----------------------------------------------------------------------
```

For each code we measured the total execution time, the upload and download bandwidths and the kernel execution time – including and excluding all data transfers. With these measurements, and the formulas (9) and (11), we calculate the speed-up related to the total program and the GPU kernels, as well as the kernel efficiency.

6. GTX670, GTX970, GTX1080 performance comparison

Although the problem used here is a simple one, and in fact can be moved in its entirety onto the GPU, we have moved only a small section of our code in order to give results that are more representable to other codes, where perhaps only a small section can be parallelized in such a way.

In the following tables and figures the * suffix, for example GPU*, denotes the optimized version of the code. The optimisation is simply directly storing the array to pinned memory on the host, no other attempt at optimisation is made. This can reduce data transfer times and increase the bandwidth because the host does not need to transfer data from pageable memory into pinned memory before communicating the data to the device.

For GPU and GPU* the reported bandwidths are the averages over the whole program time since each time step solved on the device has one upload and one download. The DYN and DYN* are dynamic programs and therefore have only a single upload and download at the beginning and end of the program, respectively, therefore we report the GFLOPs since these are computation intensive programs as opposed to the classical GPU which is transfer intensive.

Table 1 shows the serial time for the solve kernel and the total program times, we use the time in this table to compute the speed up for the GPU and DYN codes. The kernel times, reported in tables 2, 3, 4, 5 and 6, include data from all transfers. Table 7 shows the kernel time without data transfers.

It is worth noting that the GPU, and GPU*, kernel times shown in table 7 are only for the solve subroutine, whereas the dynamic, DYN and DYN*, times reported also include updating the array u at each time step. This explains why the dynamic kernel time is larger. However, when considered with the minimised data transfers and the fact that by updating the array on the device we will also achieve a speed up compared to the updating on the host, the extra seconds on the device reported here actually account for vast speed up elsewhere which can be seen in the following figures.

Figure 1 shows the upload and download bandwidths for the GPU and GPU* programs on each of the three architectures used. Figure 2 shows the GFLOPs (double precision) for the DYN and DYN* programs on the GTX970 and GTX1080 devices, the GTX670 is excluded as it does not support dynamic calls.

Figures 3 and 4 show the kernel speed up for each GPU program on each architecture calculated using equations in Section 3. Although it is not completely correct to calculate the speed up with these formulas as we are comparing the performance of a code run on different hardware, it is the simplest way for

Table 1. Total & Kernel times (s) – CPU

Grid	CPU	
	Kernel	Total
512 × 512	7.247	11.762
1024 × 1024	34.322	55.827
2048 × 2048	130.082	212.972
4096 × 4096	835.188	1060.124
8192 × 8192	3361.552	5170.051

Table 2. Total & Kernel times (s) – GTX670

Grid	GPU		GPU*	
	Kernel	Total	Kernel	Total
512 × 512	7.186	29.709	4.439	35.031
1024 × 1024	23.305	60.724	15.164	63.279
2048 × 2048	91.167	190.333	59.125	170.197
4096 × 4096	386.496	747.904	269.163	656.538
8192 × 8192	1552.581	2942.682	1012.924	2390.174

Table 3. Total & Kernel times (s) – GTX970

Grid	GPU		GPU*	
	Kernel	Total	Kernel	Total
512 × 512	7.707	29.154	4.404	40.734
1024 × 1024	25.834	60.945	15.539	63.983
2048 × 2048	102.984	202.535	62.302	180.524
4096 × 4096	488.000	888.885	275.630	685.531
8192 × 8192	1772.871	3195.492	1128.233	2689.674

Table 4. Total & Kernel times (s) – GTX1080

Grid	GPU		GPU*	
	Kernel	Total	Kernel	Total
512 × 512	9.346	32.582	6.511	41.932
1024 × 1024	32.913	68.340	23.867	74.164
2048 × 2048	123.982	222.521	94.126	205.972
4096 × 4096	568.662	946.442	400.474	781.307
8192 × 8192	2257.086	3748.056	1598.980	3079.058

standard users to understand performance difference. Figure 3 includes data transfers and we can see that we obtain a speed up of between 1.5× and 3.25×, however if data transfer times are removed, as in Figure 4 we obtain a speed up from 150× to almost 500× depending on the device architecture. This is interesting for two reasons; firstly it shows that if we are able to limit or optimize

Table 5. Total & Kernel times (s) – GTX970

Grid	DYN		DYN*	
	Kernel	Total	Kernel	Total
512 × 512	0.151	14.895	0.146	14.700
1024 × 1024	0.541	16.578	0.524	16.442
2048 × 2048	2.031	23.890	1.984	24.278
4096 × 4096	8.069	53.076	7.919	52.237
8192 × 8192	32.314	158.030	31.672	161.593

Table 6. Total & Kernel times (s) – GTX1080

Grid	DYN		DYN*	
	Kernel	Total	Kernel	Total
512 × 512	0.089	15.506	0.085	14.940
1024 × 1024	0.315	16.264	0.306	16.269
2048 × 2048	1.230	22.721	1.206	22.730
4096 × 4096	4.902	47.911	4.790	47.797
8192 × 8192	19.899	143.304	19.506	137.992

Table 7. Kernel times (s) – excluding data transfers

Grid	GPU			DYN		
	Kepler	Maxwell	Pascal	Kepler	Maxwell	Pascal
512 × 512	0.167	0.113	0.079	—	0.144	0.081
1024 × 1024	0.471	0.287	0.165	—	0.516	0.293
2048 × 2048	1.997	0.937	0.502	—	1.950	1.154
4096 × 4096	5.720	3.065	1.803	—	7.754	4.584
8192 × 8192	22.667	12.198	7.288	—	31.098	18.720

the data transfers, either through software techniques or hardware improvements, then there is a great potential for time improvement. Secondly we can see from Figure 3 that the GTX670 has the best performance, followed by the GTX970 and finally the GTX1080 if we include data transfers, however, Figure 4 shows that with the absence of transfer times the GTX1080 is now the best, followed by the GTX970 and the GTX760. This suggests that the newer generation GPU devices take more time receiving and storing the data and preparing to send the data back to the host and could hint at possible memory allocation inefficiencies.

Figure 5 shows the efficiency, calculated without data transfers. The GPU devices increase in efficiency as we move from the oldest to newest generation, from around 12% with the GTX670 to almost 20% with the GTX1080. This suggests that not only do we have more cores on the newer device but we are able to use them more efficiently.

Figures 7a and 8a show the kernel speed up for the dynamic program. The kernel in this case is both solving at each time step and updating the array in

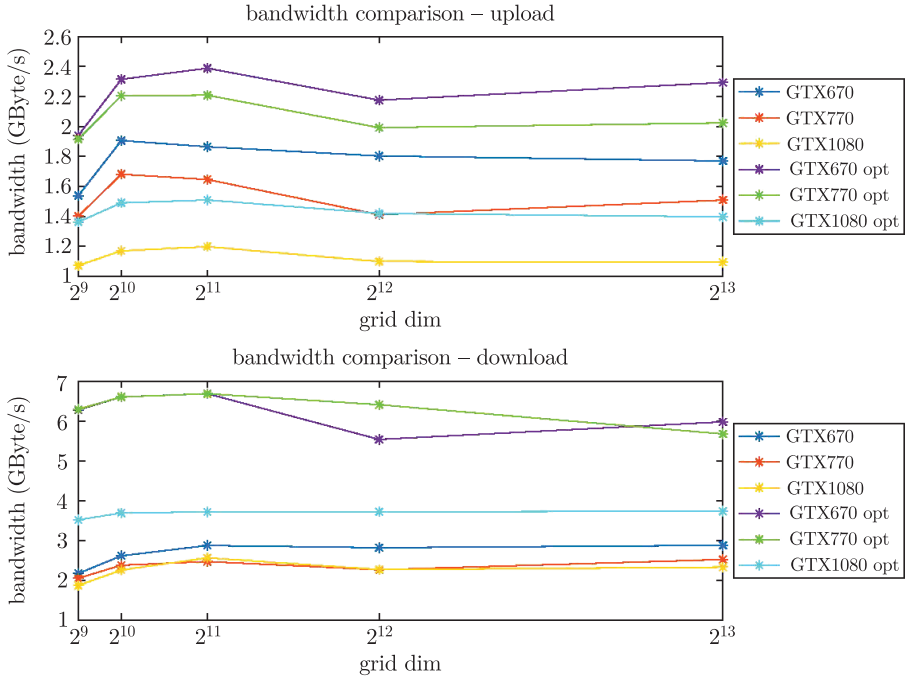

Figure 1. Classic program bandwidth

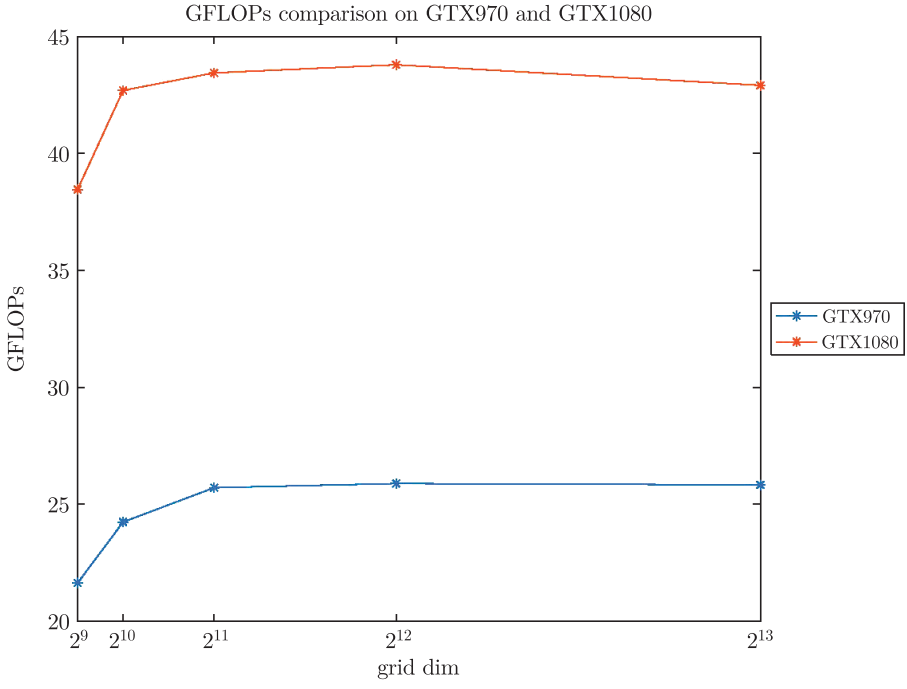

Figure 2. Dynamic program GFLOPs for Maxwell and Pascal

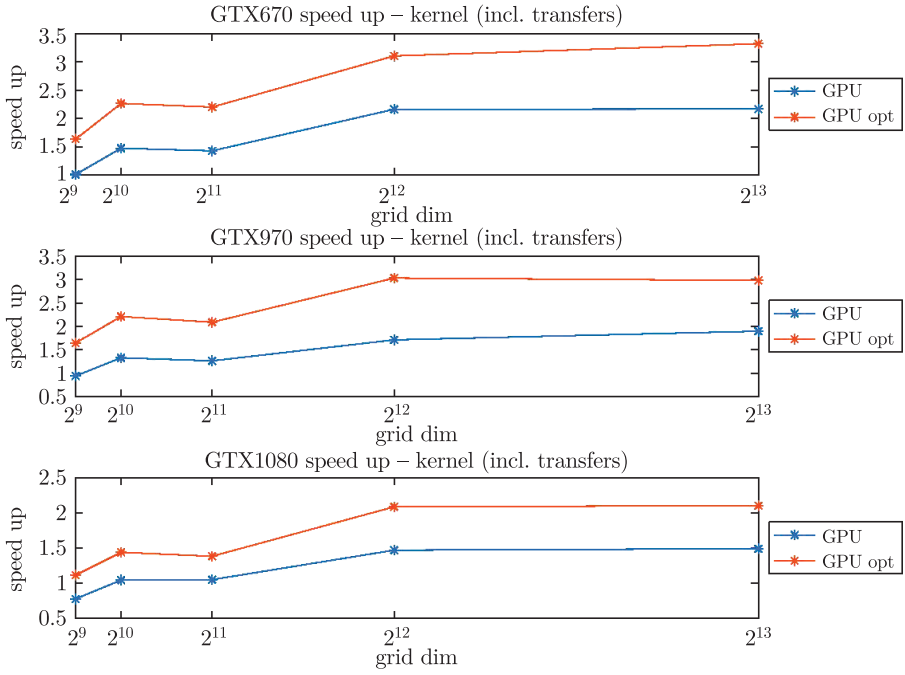

Figure 3. Kernel speed up (incl. transfers) across the three architectures

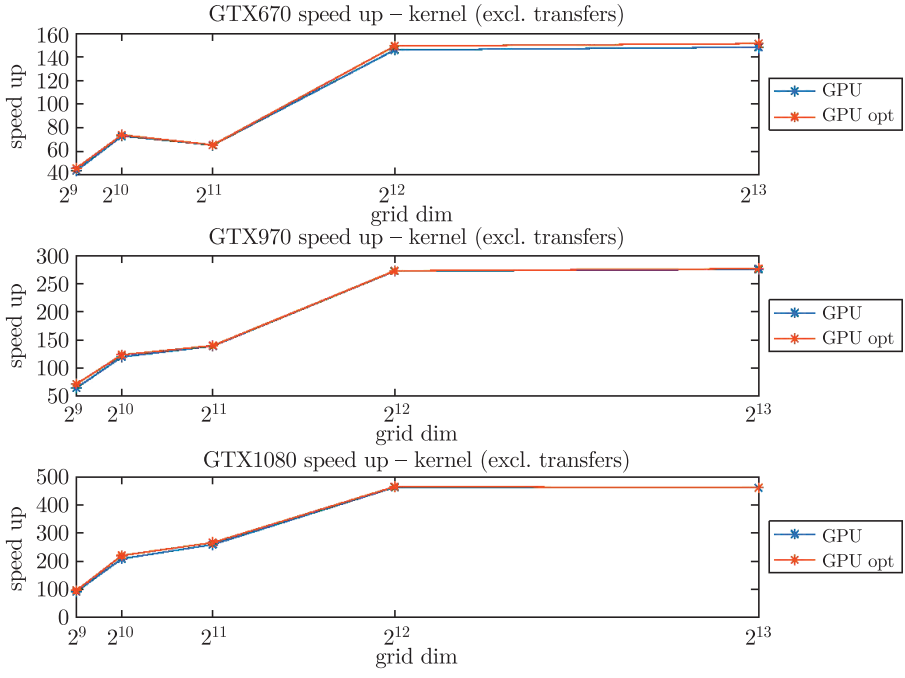

Figure 4. Kernel speed up (excl. transfers) across the three architectures

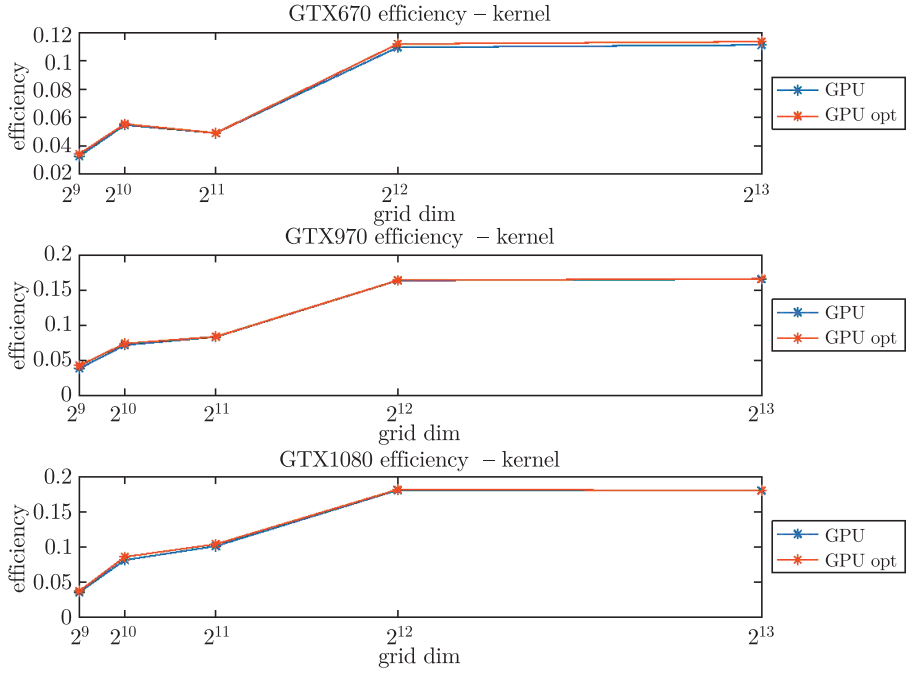

Figure 5. Kernel efficiency across the three architectures

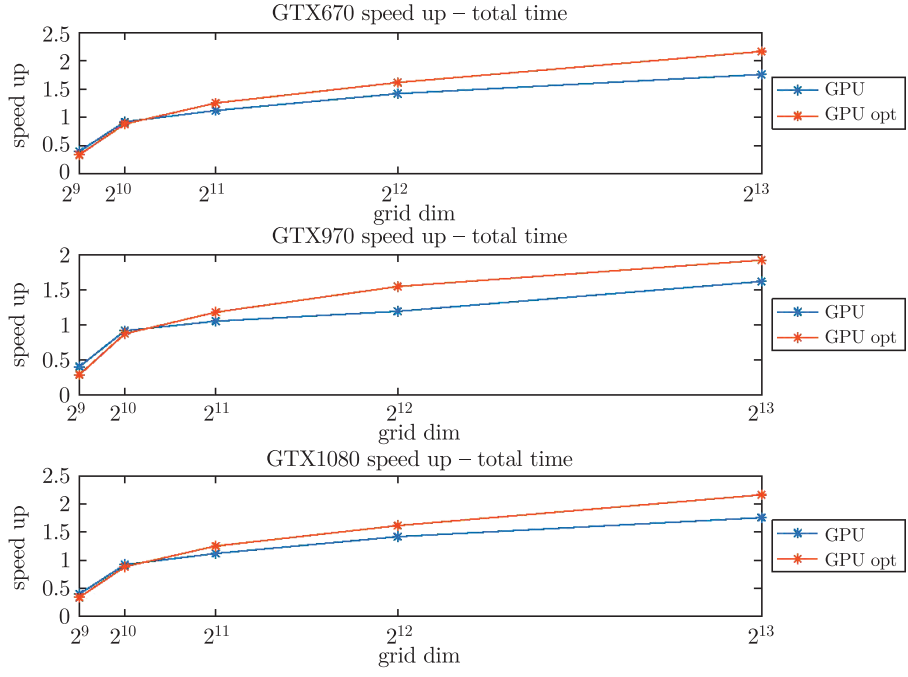

Figure 6. Total time speed up across the three architectures

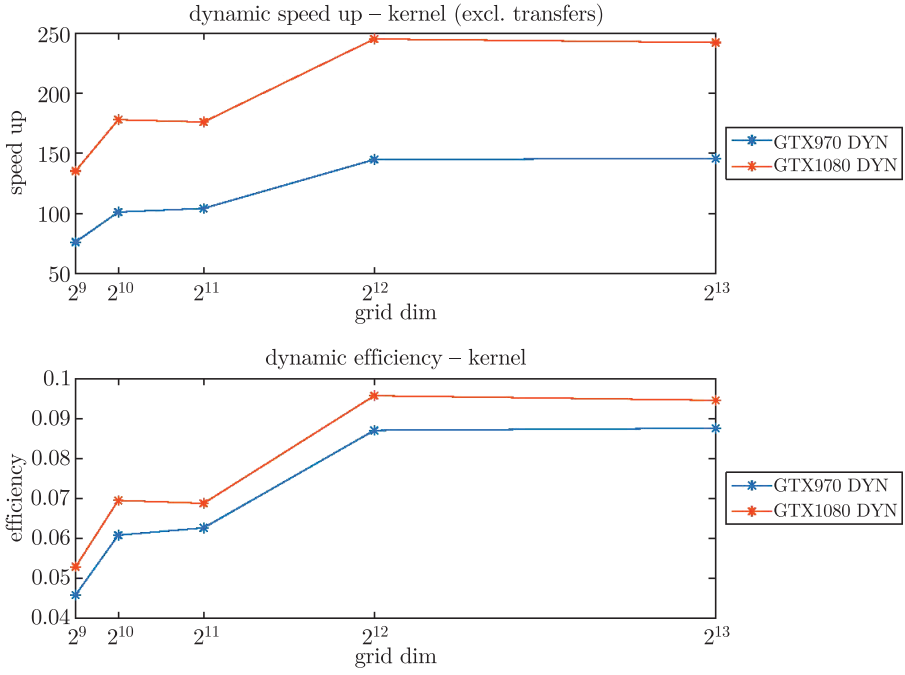

Figure 7. Dynamic kernel speed up and efficiency

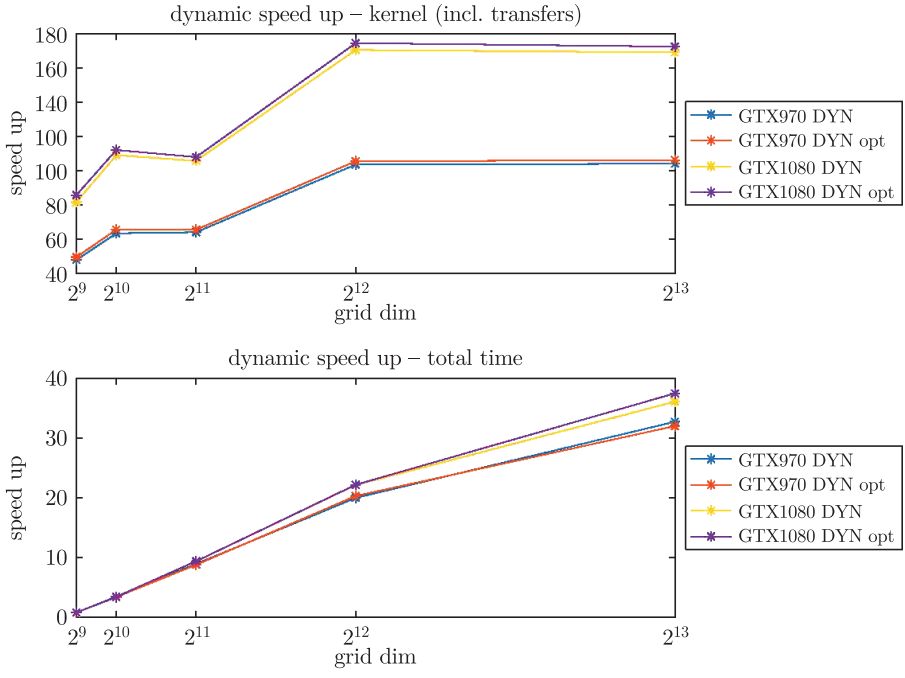

Figure 8. Dynamic kernel (incl. transfers) and total time speed up

time. Figure 7a shows a kernel speed up of between 150× and 250× depending on the device, which translate into efficiencies of 9% to 10%, as shown in Figure 7b. Figure 8a includes the data transfer time in the speed up calculations which still show significant values of 100× to 180×.

Figures 6 and 8b may be the most important for the general user as they show the improvement in total time of the programs. Figure 6 shows that the GPU* code outperforms the standard GPU program, however the results are comparable and we achieve a speed up of around 2×. In Figure 8b the difference between the DYN and DYN* programs is negligible, since we have only a single upload and download, however the speed up is significant at between 30× and 40×. Studying tables 1, 5 and 6 shows that this is a reduction from 5170s to 161.5s and 138s for the GTX970 and the GTX1080 respectively.

It is clear then that moving as much work, and control, as possible onto the device can give significant performance benefits. Depending on the device and method chosen it is possible to obtain a speed up from 2× to 30× in total time, from 160× to 500× for the solve kernel and from 100× to 250× for the solve and update kernel in the dynamic case. Therefore, it is obvious that if a code is to be partially solved on a device, care should be taken to design the code in such a way as to minimize the data transfers between the host and the device, otherwise the potential speed up resulting from the use of the device will be lost.

7. Conclusions

We have shown differences across three GPU architectures; Kepler, Maxwell and Pascal, using a classical GPU coding approach to solve the wave equation (1). We have demonstrated the possible performance benefit of moving all work onto the GPU through the use of dynamic calls. Through the optimization of data transfers, as in GPU*, and the optimized use of the GPU, as with the dynamic code, it is possible to achieve significant time improvements, however, this requires more complex programming and therefore more time to program, and, in the case of the dynamic code, requires the ability to move all connected work to the device.

Acknowledgements

We thank the University of L'Aquila and the DISIM department for the use of the High Performance Computing Lab and the Caliban cluster (http://caliban.dm.univaq.it). All figures are realized by using Octave 4.0.0 [14].

References

[1] Pera D 2013 *Parallel numerical simulations of anisotropic and heterogeneous diffusion equations with GPGPU*, PhD Thesis
[2] Roniotis A, Marias K, Sakkalis V, Tsibidis G D and Zervakis M 2009 *31st Annual International Conference of the IEEE EMBS*
[3] Rubio F, Hanzich M, Farrès A, Puente de la J and Cela J M 2014 *Computers and Geosciences* **70** 181
[4] Weickert J 1998 *Anisotropic Diffusion in Image Processing, ECMI Series*, Teubner-Verlag
[5] https://www.pgroup.com/resources/cudafortran.html

[6] *CUDA Programming Manual NVIDIA 2010*
[7] Kirk D and Hwu W-M 2010 *Programming Massively Parallel Processors: A Hands-on Approach NVIDIA*
[8] Sanders J and Kandrot E 2010 *CUDA by example An Introduction to General Purpose GPU Programming*, Addison-Wesley
[9] Kupferschmid M 2010 *Classical Fortran: Programming for Engineering and Scientific Applications, Second Edition*, CRC Press
[10] *White Paper NVIDIA Kepler GK110, http://www.nvidia.com*
[11] *White Paper NVIDIA Maxwell GTX980, http://www.nvidia.com*
[12] *White Paper NVIDIA Tesla P100, http://www.nvidia.com*
[13] https://en.wikipedia.org/wiki/Kepler
[14] Eaton J W, Bateman D, Hauberg S and Wehbring R 2015 *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations*, CreateSpace Independent Publishing Platform