The potential of graphical processing units to solve hydraulic network equations

P. A. Crous, J. E. van Zyl and Y. Roodt

ABSTRACT

The Engineering discipline has relied on computers to perform numerical calculations in many of its sub-disciplines over the last decades. The advent of graphical processing units (GPUs), parallel stream processors, has the potential to speed up generic simulations that facilitate engineering applications aside from traditional computer graphics applications, using GPGPU (general purpose programming on the GPU). The potential benefits of exploiting the GPU for general purpose computation require the program to be highly arithmetic intensive and also data independent. This paper looks at the specific application of the Conjugate Gradient method used in hydraulic network solvers on the GPU and compares the results to conventional central processing unit (CPU) implementations. The results indicate that the GPU becomes more efficient as the data set size increases. However, with the current hardware and the implementation of the Conjugate Gradient algorithm, the application of stream processing to hydraulic network solvers is only faster and more efficient for exceptionally large water distribution models, which are seldom found in practice.

Key words | computer programming, GPGPU, hydraulic modeling

P. A. Crous (corresponding author)
Y. Roodt
University of Johannesburg,
South Africa
E-mail: *bye.product@gmail.com*

J. E. van Zyl
University of Cape Town,
South Africa

INTRODUCTION

The multi-billion dollar gaming industry has created a continuous and strong demand for more realistic computer graphics and faster animations in PC games (Monti *et al.* 2005). As a result, graphical processing units (GPUs) have seen vast performance increases in processing capacity. At the time of writing, commercially available AMD GPUs were equipped with up to 3,200 parallel processors capable of 4.64×10^{12} single precision floating point operations per second, or 4,640 GFLOPs (AMD 2009). Thus the GPU has become a mass-parallel processing unit with thousands of processors.

Due to the overwhelming demand for GPUs in the gaming industry, GPUs have become considerably cheaper than central processing units (CPUs) with similar processing power. This has created an incentive for programmers to exploit the processing power of GPUs for non-graphical problems, such as those found in engineering disciplines. This movement was strengthened by certain GPU developments such as increased speed, increased parallel processors, IEEE 32-bit floating point precision and the unlocking of former fixed pipelines in graphics hardware (St-Laurent 2004; Owens *et al.* 2005). Large IT corporations, such as Nvidia, AMD, Intel, Microsoft, IBM, Apple and Toshiba, recognized the potential for GPUs, and have all made significant advances in the field in recent years (Evans 2009).

The aim of this study was to investigate the application of stream processing on the GPU to water distribution network solvers. The paper starts off by introducing GPUs and their potential in general purpose programming. It introduces necessary concepts and terminology, and an overview of the GPU programming paradigm. It then describes how stream processing on the GPU was applied within hydraulic network solvers using the Conjugate Gradient solver. Finally, the results and conclusions are presented.

doi: 10.2166/hydro.2011.023

GRAPHICAL PROCESSING UNIT (GPU)

The GPU is a dependent, auxiliary component of a computer designed for the mathematical manipulation of computer graphics, and thus does not possess the more wide-ranging capabilities of the CPU (Owens *et al.* 2007). Even though the primary purpose of the GPU is in the area of graphics, it is capable of running more general instructions and is considered a programmable floating-point processor.

The processing capacity of GPUs has been nearly doubling every year. This rate is 1.4 times greater than Moore's Law, which states that the CPU processor's performance doubles every 18 months (Ekman *et al.* 2005; Owens *et al.* 2007). Figure 1 shows the increase in processing capacities of Intel CPUs compared to ATI and Nvidia GPUs since 2006. The primary driver of the GPU performance increase has not only been clock rate, but also increases in the number of processors on the GPU, allowing for a higher level of parallelization.

It is important to note that parallel processing is not unique to GPUs, but can also be implemented on CPUs through threads that run on multiple processors on the same computer or on a cluster of computers. However, generally CPU parallelization is limited by the number of processors on the computer's motherboard (at the time of writing the commercially available limit is eight processors), and cost. Consider, for instance, a typical 4-core CPU, the Intel Core i7-975, costing $999 upon release (Anandtech 2009) which can process up to 55.36 GFLOPs (Intel 2009). In comparison, current GPUs such as the Nvidia GTX 295 and AMD Radeon HD5970 are substantially more powerful with processing capacities of 1,788 and 4,640 GFLOPs respectively, while at the same time significantly cheaper, with release date prices of $550 and $650 respectively (GPUReview.com 2010).

GPU applications have already outperformed CPU-only clusters in a variety of different applications. One notable example at the University of Antwerp in Belgium is the creation of a supercomputer using 4 Nvidia 9800GX2 GPUs (with the same processing capacity as a cluster of 300 Intel Core 2 Duo 2.4GHz CPUs), for less than $5,750 for use in tomography, a technique to produce three-dimensional images of internal organs through the combination of a large number of x-rays (De Maesschalck 2008; University of Antwerp 2009). Recent studies (see Guidolin *et al.* 2010; Wu & Eftekharian 2011) in the field of hydroinformatics have used CUDA for hydraulic network modeling.

There are a number of important differences between the operation of the CPU and GPU. The GPU stores memory in a specialized and rigid memory structure, called texture memory, which acts like the CPU's random access memory (Karsten & Trier 2004). In addition, loops

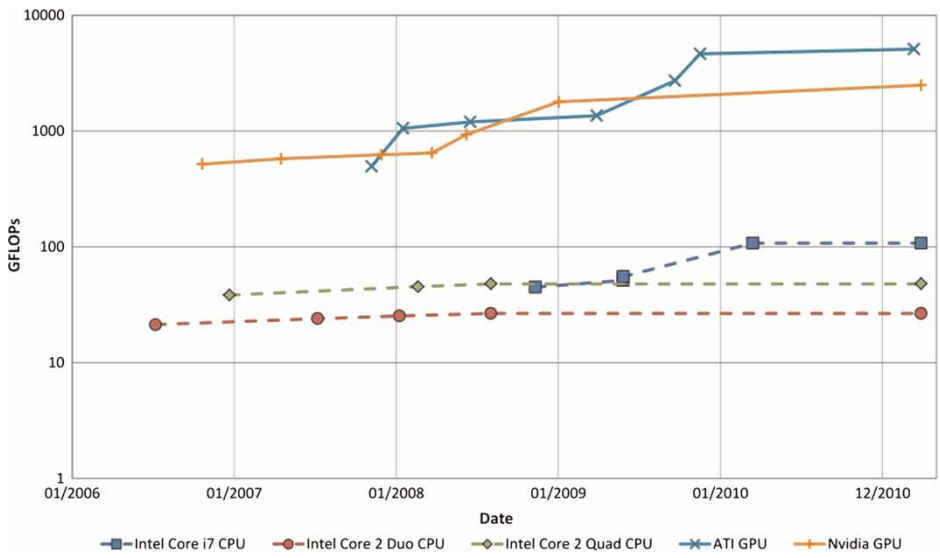

Figure 1 | Growth in CPU and GPU processing capacity.

used in the conventional programming model on a CPU are replaced with shader programs, or kernels, on the GPU. Shader programs are instruction sets that continually operate on data streams sent to the processor until all the data have been processed. The GPU processors can all run the same shader program simultaneously, and thus process the data in a highly parallel manner (Karsten & Trier 2004; Harris 2005). Finally, computations performed in functions on the CPU are also performed in shader programs on the GPU. A program can be made up of several shader programs as several functions combine to produce a program.

GPUs are not likely to threaten the prominence of CPUs since they are developed for very specific applications. Stream processing on the GPU should rather be seen to complement the CPU as a co-processor for certain types of applications. Nonetheless, it is likely that GPUs will play an important role in engineering calculations in the future. Evans (2009) suggests that non-graphical computation on GPUs could be the most important computing trend over the next 10 years.

GPU operation

To carry out stream processing, it is important to understand how the host (CPU) directs the GPU to perform calculations, and how the GPU itself operates. The process is graphically illustrated in Figure 2.

The host is responsible for managing the computer's resources, interacting with the outside world and sending commands to the GPU (Pharr & Fernando 2005). Communication between the host and the GPU is performed through a bus, which is simply a set of wires that carry data (Dokken *et al*. 2007). The capacity of the bus is limited, but the need to send data between the host and GPU can be reduced by accessing local memory on the GPU in the form of caches (vertex buffer or framebuffer) and texture memory (Owens *et al*. 2005). The most common bus used in modern graphics cards is the PCI (peripheral component interconnect) Express bus.

The use of the framebuffer minimizes the latency and communication required to send the data back to the host and then sending it to the GPU at a later stage (Nvidia 2006). Therefore, the data is generally adapted so it can be stored on the framebuffer and used in all proceeding calculations.

The classic graphics pipeline is shown in Figure 2. This pipeline has been used in graphics cards for the last 20 years, with improvements emerging in the functionality and programmability within each of the stages. Data from memory is made available to the GPU through the use of textures, which consist of arrays of pixels (Harris 2005). Textures are processed by the shader programs and are output as fragments. Fragments contains pixels, as well as extended data like depth, normal, and texture coordinates (Dokken *et al*. 2007). Before fragments become pixels, they still undergo many different operations (Nvidia 2006). Each pixel typically consists of four channels of data, three for the primary colors and the fourth for transparency. Thus, in a

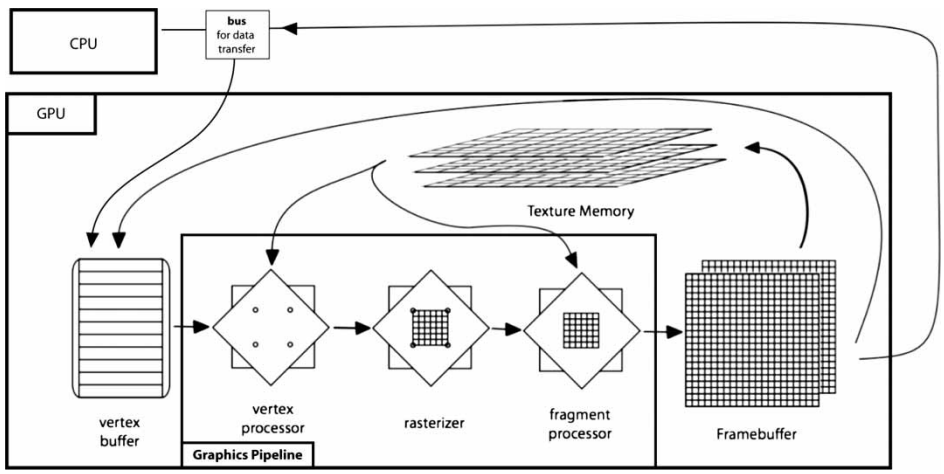

Figure 2 | Diagrammatical view of the classical CPU and GPU setup (adapted from Owens *et al*. 2007).

non-graphical context a pixel can be viewed as a one-, two-, three-, or four-dimensional vector-array of numbers (Patin 2003).

The first stage of the graphics pipeline (the vertex processor) transforms each of the vertices from world space to screen space. Textures are only processed once they are mapped onto these vertices. The second stage, called the rasterization stage, fills each of the triangles produced in the first stage with a color. The output of this stage is a fragment for every place where a pixel is situated. This stage is performed off screen (Möller & Haines 2002).

The third and final stage, called the fragment stage, determines what color each fragment must be. The fragment program reads from any global memory address, through the use of textures in a process called texture mapping (Dokken *et al.* 2007), but only writes to the framebuffer.

The output from the pipeline is a texture, which is stored in the framebuffer (Owens *et al.* 2005). The output from the framebuffer can either be sent back to the CPU or reused within the graphics pipeline.

Important GPU hardware limitations are the amount of memory allocated to a single texture, and the total amount of data that can be stored on the GPU's onboard memory. Once the total onboard memory limit is reached on a Windows XP-based system, the GPU cannot process the data stream. Windows Vista and Windows 7 accommodate for the limit in onboard graphics memory by introducing the ability to use global memory for the GPU, and thus the total memory required to process the algorithm would not be a limiting factor (Microsoft 2006). The use of this virtualized memory, however, causes losses in processing power, because the GPU has to send and receive data from the CPU through the bus.

Unified GPUs were first introduced in the Nvidia GeForce 8800 in 2006, and have since become the standard for GPUs. Prior to unified processors, GPUs consisted of a fixed number of fragment and vertex processors. Unified processors consist of many stream processors which manage GPU processing resources more efficiently as stream processors dynamically switch between performing fragment or vertex processes. This eliminates the need for programmers to consider load balancing of the different processors. Thus, unified GPUs have become ideal general-purpose parallel floating-point processors (Nvidia 2006).

Programming for the GPU

The languages and API (application program interface) available to program for the GPU all exploit the GPU's graphical capabilities for non-graphical calculations (Owens *et al.* 2005). While there are small differences between the languages, they all follow the same basic framework. The main programming languages available for stream processing on the GPU include:

- Shader based API
  - CG (C for Graphics) used for Nvidia graphics cards
  - HLSL (High Level Shader Language) used by Microsoft through DirectX
  - GLSL (OpenGL Shader Language) used by OpenGL
- GPGPU API
  - CUDA (Compute Unified Device Architecture) a Nvidia GPGPU language
  - OpenCL (Open Computing Language) cross platform API
  - Direct Compute, a Microsoft GPGPU language used in Windows Vista and Windows 7

The OpenGL shading language was used in this study as it is very stable and has been in use for over 20 years, undergoing many revisions in this time (GLSL 4.10 was released in July 2010). Thus, OpenGL and GLSL continually stay on top of latest hardware developments and are trusted and stable languages. Although GLSL is not a direct GPGPU language, GLSL is used to program real-time graphics and thus is a highly optimized API. OpenGL is also manufacturer independent.

A typical OpenGL program consists of creating a window and rendering a scene to the framebuffer (Segal & Akeley 2008). OpenGL is only concerned with rendering data into and reading data from the framebuffer (Segal & Akeley 2008). The library does not provide any support for any other peripherals such as a mouse or a keyboard. However, the OpenGL Utility Toolkit (GLUT), a freely distributed, cross-platform windowing API used to interact between the operating system and OpenGL, can perform these tasks (Kilgard 1996). GLUT consists of a simple but useful interface and was designed for use with simple to moderately complex programs, which deal mainly with OpenGL rendering (Kilgard

1996). GLUT has support for C, C++, FORTRAN, and Ada; and is also platform independent (Woodhouse 2002).

The extent of parallelism used on a GPU depends on the ability of the programmer to exploit the instruction-level parallelism which is available through the typically four element vector structure within the GPU and ensuring that the numerous processors on the GPU are utilized for the program (Harris 2005). In order to take full advantage of the GPU's capabilities, it is important that shader programs are highly arithmetically intensive, i.e. has a large ratio of mathematical operations to memory accesses (Pharr & Fernando 2005). If it is not, the overheads added by the initialization will take longer and slow down the algorithm (Dokken *et al*. 2005).

Data parallelism is performed on the GPU by assigning a shader program to each processor on the graphics card. This allows each processor to perform the same set of instructions (or shader program) at the same time on different data streams. A data stream is made up of arrays of data stored in textures, and includes both the input to and output from the shader programs (Owens *et al*. 2007).

A stream can be simple or complex, for instance it can consist of a stream of floating-point numbers or a stream of transformation matrices. A single stream can also be of any length, although longer streams are more efficient than shorter ones. A shader program works on an entire stream and cannot perform different tasks on individual elements in the stream (Owens *et al*. 2005). Therefore, the elements within the stream must be independent of other elements within the same stream (Owens *et al*. 2005). This allows shader programs to be highly efficient if the input data and the computed data are either stored locally or through carefully controlled global references. The same shader program can run on different GPU processors, each simultaneously manipulating a different data stream (Owens *et al*. 2005).

APPLICATION OF STREAM PROCESSING TO HYDRAULIC NETWORK SOLVERS

Approach

Hydraulic modeling is an indispensible tool for the design of a water distribution system due to the size and complexity of many networks, and the non-linear nature of the governing hydraulic equations. The most common method used to solve the hydraulic network equations is known as the Global Gradient Algorithm (Todini & Pilati 1988). This method uses mass and energy balance to set up the network equations under the assumption of incompressible flow. The application of a Newton–Raphson based method yields a set of linearized equations that are solved simultaneously for the link flow rates, and then repeated until convergence is achieved. The Jacobian matrix in the set of linear equations is symmetrical, positive-definite, and highly sparse, and the equations are solved with the Cholesky method (Rossman 2000).

The Global Gradient Algorithm is implemented in the public domain software EPANET, developed by the US Environmental Protection Agency in the 1990s (Rossman 2000). EPANET has been adopted as the standard for hydraulic modeling in research and commercial software.

Solving the set of linearized equations in the Global Gradient Algorithm is computationally intensive, and thus has potential for application to stream processing. However, the sequential Cholesky method does not calculate inverse matrix coefficients independently of each other. Since these internal dependences in the data stream are not allowed in stream processing, the sequential Cholesky method could not be used, and an alternative linear equation solver had to be found.

The hydraulic network equations have weak convergence characteristics, and thus neither the Jacobi nor Gauss–Seidel methods could be used in the Global Gradient Algorithm as both methods require excessive numbers of iterations to converge (Todini & Pilati 1988). However, the Conjugate Gradient method is not affected by this weak convergence and is comparable to the number of operations required by both the Jacobi or Gauss–Seidel methods (Todini 1979; Todini & Pilati 1988). Importantly, the Conjugate Gradient method calculates inverse matrix coefficients independent of each other, and was thus considered a suitable alternative to the Cholesky method for stream processing application.

While the matrix inversion is only one part of the method to solve hydraulic networks, it is done iteratively and is the main contributor to the computational cost of the solver. Since the aim of this study was to explore the implementation

and potential benefits of stream processing for hydraulic network solvers, only the matrix solution process was implemented on the GPU. The standard Conjugate Gradient method was implemented on both the CPU and GPU and the computational times compared. No measures, such as preconditioning, were implemented to optimize the efficiency of the Conjugate Gradient method, since these methods are likely to show similar improvements on the CPU and GPU. For the same reason, sparse matrix techniques were not implemented (the Jacobian matrix of a water distribution network is highly sparse), but the methods were applied to dense matrices representing highly connected networks. The implications for sparse network matrixes were interpreted from the results of the dense matrixes.

The dense matrixes were constructed with diagonal elements that are equal to the sum of the absolute values of all of the off-diagonal elements in the row, making the system weakly diagonally dominant. This is equivalent to the coefficient matrix used in the Global Gradient Algorithm. The computation time of these matrices can then be used to assume the computational time required for a water distribution system, in which each of the pipes are connected to the next two pipes in the system. The dense system consists of n^2 elements, while the sparse system would have $5n$ elements, with n being the size of the system. It is understood that there are extra computation required for the sparse implementations, like memory handling. The size of the linear system was increased starting with a 2×2 matrix (size 4), and increasing the numbers of rows and columns in powers of two.

Implementation

In this study the C++ programming language was used for the main host program and GLSL for the GPU shader programs. The OpenGL 2.1 library was used to link the C++ and the GLSL code, initialize and store the textures in the framebuffer, and initialize all the shader programs. GLUT was used to interact between the window system and OpenGL. The fragment shader programs contain the algorithms that were used to perform the Conjugate Gradient method. As the program loads all of the data onto the GPU memory through the use of textures, the CPU was not used for any of the calculations performed in the stream processing implementation of the Conjugate Gradient method.

Computational time was first estimated from the time it took to perform a single iteration of the Conjugate Gradient method for the various CPU and GPU implementations on a dense linear system. This provided a general comparison of the algorithm on the different architecture. However, since the number of iterations required to solve different networks vary, the total run times were also compared.

The Conjugate Gradient method used in both the CPU and GPU architecture is presented below in Figure 3, where $r^{(0)} = -b_i$ and the values of $\beta^{(0)}$ and $x_i^{(0)}$ are both equal to zero in the first iteration.

The reduction operation, used when finding a single value from a data stream, for example finding the minimum or maximum value or making a summation of all the elements in an array, is used extensively in each of the solution methods in the Conjugate Gradient method. The conventional, sequential reduction method is, however, inefficient in parallel on the GPU. The constructed parallel reduction algorithm creates a pyramid of textures which alternatively read-from and write-to textures, in a process referred to as 'ping-pong'. This process is graphically shown in Figure 4, where the size of the output texture is

for $\boldsymbol{i = n}$ **do**

$$x_i^{(k+1)} = x_i^{(k)} + \alpha^{(k)} p_i^{(k)}$$

$$r_i^{(k+1)} = r_i^{(k)} - \alpha^{(k)} z_i^{(k)}$$

$$\beta^{(k+1)} = \frac{r_i^{(k+1)T} r_i^{(k+1)}}{r_i^{(k)T} r_i^{(k)}}$$

$$p_i^{(k+1)} = -r_i^{(k+1)} + \beta^{(k+1)} p_i^{(k)}$$

for $\boldsymbol{j = 1}$ **do**

$$z_i^{(k+1)} = A_{ij} p_i^{(k+1)}$$

$$\alpha^{(k+1)} = \frac{r_i^{(k+1)T} r_i^{(k+1)}}{p_i^{(k+1)T} z_i^{(k+1)}}$$

Figure 3 | The Conjugate Gradient method.

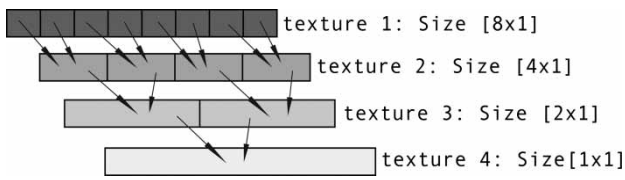

Figure 4 | Parallel reduction, used on the GPU.

recursively halved. Thus, textures of size $n = 2^p$ can generally be solved in O(log n) steps, which reduce the texture to a scalar value in fewer operations than the sequential reduction algorithm, which solves in O(n) steps (Owens *et al*. 2005). The reduction operation is performed in the calculation of α, β, where a vector is reduced to a single value, and $\boldsymbol{z}$, where a matrix is reduced to a vector.

The procedure for multiplying a matrix, $\boldsymbol{A}$, of size $[n \times n]$ with a vector, $\boldsymbol{x}$, of size $[n \times 1]$ is calculated on the GPU such that it first produces a new texture $\boldsymbol{z}$ of size $[n \times n]$, such that $\boldsymbol{z}_{ij} = \boldsymbol{A}_{ij} \times \boldsymbol{x}_j$, where i denotes the column and j the row in each variable. The procedure is complete once all of the rows in the texture are summed using the reduction algorithm creating a texture of size $[n \times 1]$.

The calculation of α and β were performed using three shader programs. The first program multiplies each of the vectors together. The second program reduces each vector to a single value. The final program divides these values.

As an example of the GLSL code used for general computation on the GPU, the code required for calculating matrix $\boldsymbol{z}$ is presented in Figure 5. This shader program is executed on the fragment processors and has two inputs (lines 2 and 3). The inputs are defined before the shader program is processed and they both remain constant throughout the duration of the shader program. Lines 9 and 11 are used to find the value of the texture at a specific point, which is equivalent to finding a value within an array. As the input texture *vector* consists of one column of values, the x-axis is only required at the midpoint and all the y-axis values are found. The shader program is processed simultaneously by all available fragment processors until all data streams are processed. If there are more data streams than available processors on the GPU, the data streams are processed in batches until all the elements in the data stream are processed. Further, this shader program can be used on a square matrix texture and a vector texture of any size, as the shader program itself does not define the size of the input textures. The output (line 13 in Figure 5) is placed into a new, but predefined texture in the framebuffer.

```
1 //input textures
2 uniform sampler2D matrix;
3 uniform sampler2D vector;
4
5 void main(void) {
6 float M_value,
7 V_value;
8 //look up the value at the texture coordinate in the matrix texture inA
9 M_value = texture2D(matrix,vec2(gl_TexCoord[0].x, gl_TexCoord[0].y)).x;
10 //look up the value at the texture coordinates in the vector texture in x
11 V_value = texture2D(vector,vec2(0.5, gl_TexCoord[0].y)).x;
12 //output the answer to a new texture in z
13 gl_FragColor = vec4(M_value*V_value);
14 }
```

Figure 5 | The matrix–vector multiplication shader program.

The calculations were performed using both the stream processing model on the GPU and conventional programming model on the CPU. This was repeated on two different computers: Computer 1 ran Windows XP (Service Pack 2). The 6.14.11.9038 driver, released on 14 July 2009, was installed for the graphics card in Computer 1. Computer 2 ran Windows 7 Enterprise 64-bit. The 8.17.12.6089 graphics driver, which was released on 10 August 2010, was used for this GPU, an Nvidia GTX 260. The specifications of each of the computers are presented in Table 1.

Both of the GPUs use the PCIe 2.0×16 bus to transfer data between the GPU and the CPU, which run at 8 GB/s. The onboard memory transfer for each of the GPUs is presented in Table 1.

Findings

The computation time results for both computers are shown in Figure 6 for a single iteration of the Conjugate Gradient method, and Figure 7 for full convergence of the Conjugate Gradient method.

Figure 6 highlights the differences between the CPU and GPU architectures. Initially the GPUs use considerably longer computation times, but the CPU computation times increase at a considerably higher rate The GPUs outperform the CPUs for linear systems larger than $1{,}161 \times 1{,}161$ on Computer 1 and 857×857 on Computer 2.

The trend lines are presented in Figure 6 in order to analytically compare the Conjugate Gradient method on the two different architectures. The computational time of the smallest (smaller than 10×10) matrixes are ignored, since additional computational overheads clearly affected their calculation speed significantly.

The reasons that the CPU outperforms the GPU for the smaller linear system sizes include the following:

- *Clock rate:* The clock frequency of a CPU core is significantly higher than that of the GPU cores. For example,

Table 1 | Specifications of the different computers

	Computer	Processor	Number of processors	Clock rate	Shader clock rate	RAM	Memory speed	GFLOPs
CPU	1	Intel Celeron	1	2.8 GHz	–	1 GB	400 MHz	
	2	AMD Athlon II X2 250	2	3.01 GHz	–	4 GB	1,333 MHz	
GPU	1	GeForce GTX 285	240	648 MHz	1.476 GHz	1 GB	1,242 MHz	1,062.7
	2	GeForce GTX 260	192	576 MHz	1.242 GHz	512 MB	999 MHz	715.4

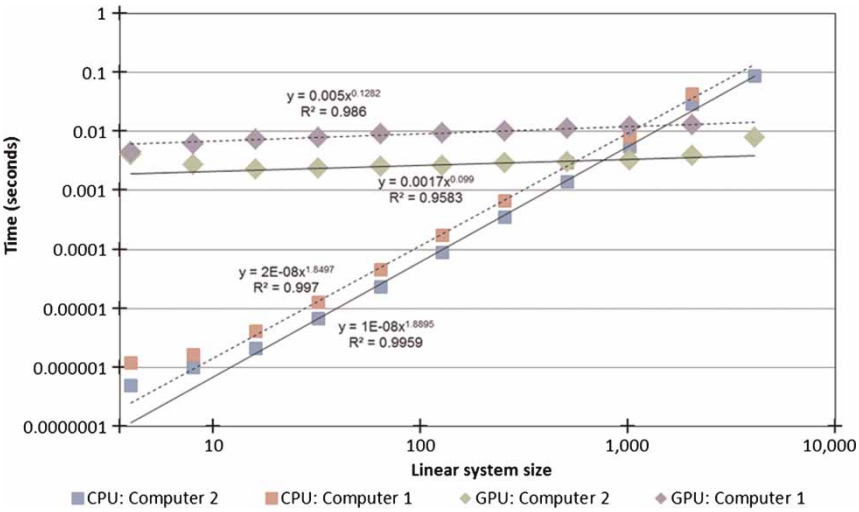

Figure 6 | Performance of a single iteration of the Conjugate Gradient.

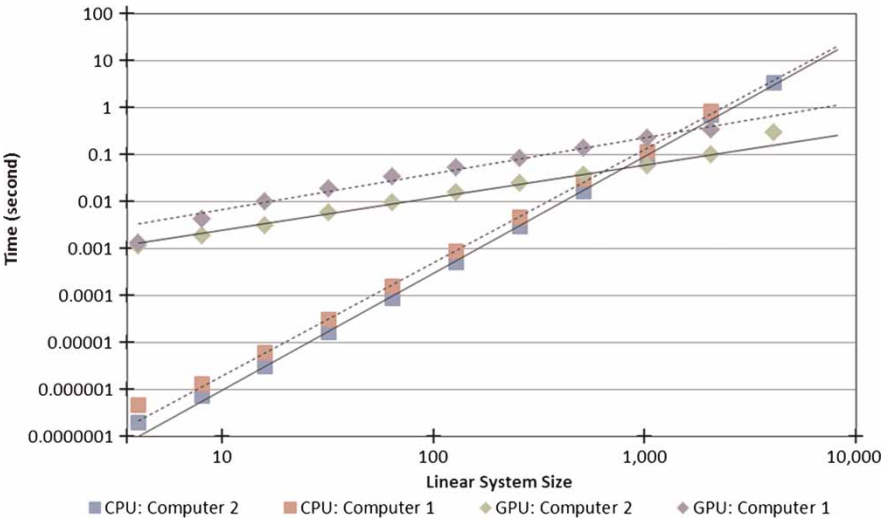

Figure 7 | Performance for the total convergence of the Conjugate Gradient.

the CPU in Computer 2 ran at a clock rate of 2.7 GHz while each of the processors on the GPU in Computer 2 ran at 1.2 GHz. This higher clock frequency allows the CPU to perform operations much faster on small data sets than the GPU can for the same small data set due to low parallelization. However, the advantage of

the GPU is evident as the number of elements in the data stream increases as there are 192 processors running the shader programs in parallel on the GPU compared to two processors on the CPU.

- *CPU bound:* Even though the GPU is capable of performing the same task on a data stream very efficiently, the rate at which the tasks are sent through to the GPU depend on how fast the CPU can upload shader programs to the GPU. For small networks the overheads to load these tasks are the same as for large networks, but the relative gain in computational time of the GPU calculations are small compared to these overheads.
- *Memory switching:* The CPU is more efficient at transferring data to and from global memory than the GPU.

For small data streams (less than 10) the time required to set up, handle and store the data becomes important and thus these points move away from the exponential trend line. Other data points for the analysis of the data provide good fits for experiential curves as shown. Memory handling becomes less pronounced as the linear system becomes larger. This is because the amount of arithmetic operations performed increases with an increase in system size.

The effect of CPU bound was seen in the comparison between the performance of the GPUs. The GPU of Computer 2 is significantly slower than that of Computer 1, which should result in the GPU of Computer 1 computing faster. However, the CPU in Computer 2 was superior to that of Computer 1 allowing the GPU in Computer 2 to outperform that in Computer 1. Thus, it is evident that the results using stream processing on the GPU are dependent on the capabilities of the CPU. This effect has been illustrated in a test in which a CPU processor was overclocked to different frequencies and the performance of the GPU was recorded (VR-Zone 2009).

The implication of the results for sparse systems is that the GPU would outperform the CPU for sparse matrices larger than about $270{,}000 \times 270{,}000$ on Computer 1 and $145{,}000 \times 145{,}000$ on Computer 2. These represent exceptionally large water distribution models, and thus it is unlikely that stream processing would find practical application to hydraulic network solvers using current technology and the Conjugate Gradient method. However, the study clearly showed the potential for substantial improvements in computational efficiency for engineering problems that are better suited to stream processing.

CONCLUSION

GPUs have advanced drastically in recent years and have opened up the possibility of applying them to problems outside of the graphics field. The implementation of non-graphical programming on the GPU through stream processing has great potential for speeding up certain types of engineering calculations.

Even though there is a strong movement in programming from conventional linear or parallel programming on the CPU to mass-parallel programming on the GPU, further advances in GPU architecture and software are likely to introduce the GPU to a much greater extent as a standard technology in the scientific and engineering fields.

In this study, the Conjugate Gradient method used in hydraulic network solvers was investigated on the GPU and the CPU. The implementation of the Conjugate Gradient method in the stream processing model on the GPUs proved to be both feasible and faster than the CPU implementations, but only for large linear systems. In large linear systems, the processing performance on the GPU was much greater than the CPU's performance due to the ability of utilizing multiple processors at the same time. However, the sparse nature of hydraulic networks means that GPUs will outperform CPUs only for exceptionally large networks. As a result, it is unlikely that stream processing will find practical application to hydraulic network solvers using current technology. However, the study clearly showed the potential for substantial improvements in computational efficiency for engineering problems that are better suited to stream processing.

Finally, although OpenCL could not be used in this study, the development of OpenCL could enable the next generation of parallel processing applications and should be considered in future engineering applications. It already has the support of many large corporations, such as Apple, AMD, Intel, Nvidia and IBM as it bridges the gap between the different architecture on the computer, namely the CPU and the GPU, allowing for both multiple task parallelization, which is most efficient on the CPU, and multiple data parallelization, which is most efficient on the GPU.

REFERENCES

AMD 2009 *ATI Radeon HD 5970 Graphics Feature Summary*. Retrieved 1 4, 2010, from AMD: http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5970/Pages/ati-radeon-hd-5970-specifications.aspx

Anandtech 2009 *Intel's Core i7-975 & 950: Preparing for Lynnfield*. Retrieved 01 04, 2010, from Anandtech: http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=3574

De Maesschalck, T. 2008 *University of Antwerp makes 4000EUR NVIDIA supercompute*. Retrieved October 18, 2009, from Dark Vision Hardware: http://www.dvhardware.net/article27538.html

Dokken, T., Hagen, T. R. & Hjelmervik, J. M. 2005 The GPU as a High Performance Computational Resource. *SCG 2005*. ACM Press, Budmerice, pp. 21–26.

Dokken, T., Hagen, T. R. & Mikkelsen, H. 2007 An introduction to general-purpose computing on programmable graphics hardware. In: *Geometric Modelling, Numerical Solutions, and Optimization* (G. Hasle, K. A. Lie & E. Quak, eds). Springer, Germany, pp. 123–161.

Ekman, M., Warg, F. & Nilsson, J. 2005 An in-depth look at computer performance growth. *ACM SIGARCH Comput. Architect. News* **33** (1), 144–147.

Evans, D. 2009 *Why your next CPU should be a GPU*. Retrieved May 5, 2009, from Tech Radar: http://www.techradar.com/news/computing/pc/why-your-next-cpu-could-be-a-gpu-585876

GPUReview.com 2010 *Video Card Comparison*. Retrieved 01 04, 2010, from GPU Review: http://www.gpureview.com/show_cards.php?card1=618&card2=603

Guidolin, M., Burovskiy, P., Kapelan, Z. & Savic, D. A. 2010 CWSNet: An object-oriented toolkit for water distribution system simulations. In *Proceedings of WDSA 2010* on September 12–15, Tucson, Arizona, USA.

Harris, M. 2005 Mapping Computational Concepts to GPUs. In: *GPU Gems 2* (M. Pharr, ed.). Addison-Wesley, MA, USA.

Intel 2009 *Intel Core i7 Desktop processor*. Retrieved 1 4, 2010, from Intel: http://www.intel.com/support/processors/sb/cs-023143.htm#1

Karsten, O. & Trier, P. 2004 *Implementing Rapid, Stable Fluid Dynamics on the GPU*. Retrieved January 23, 2008, from projects.n-o-e.dk/GPU_water_simulation/gpu-water.pdf

Kilgard, M. J. 1996 *The OpenGL Utility Toolkit (GLUT) Programming Interface*. Retrieved October 18, 2008, from OpenGL: http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf

Microsoft 2006 *Graphics Memory Reporting through WDDM*. Retrieved October 23, 2009, from Microsoft: http://www.microsoft.com/whdc/device/display/graphicsmemory.mspx

Möller, T. A. & Haines, E. 2002 *Real-Time Rendering*, 2nd edition. A K Peters, Ltd, USA.

Monti, G., Lindner, C., Puente, L. F. & Koch, A. W. 2005 Consumer Graphics Cards for Fast Image Processing based on the Pixel Shader 3.0 Standard. In *EOS Conference on Industrial Imaging and Machine Vision*, pp. 15–21.

Nvidia 2006 *GeForce 8800 GPU Architecture Overview*. Technical Brief TB-02787-001_V01, California.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. & Purcell, T. J. 2005 A Survey of General-Purpose Computation on Graphics Hardware. *Eurographics 2005, State of the Art Report*, pp. 21–51.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. & Purcell, T. J. 2007 A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum* **26** (1), 80–113.

Patin, F. 2003 *An Introduction to Digital Image Processing*. Retrieved May 2, 2011, from ProgrammersHeaven.com: http://www.programmersheaven.com/articles/patin/ImageProc.pdf

Pharr, M. & Fernando, R. 2005 *GPU Gems 2*. Addison-Wesley, MA, USA.

Rossman, L. A. 2000 *EPANET 2 User's Manual*. US Environmental Protection Agency, National Risk Management Research Laboratory, Cincinnati.

Segal, M. & Akeley, K. 2008 *The OpenGL Graphics System: A Specification*. Retrieved November 3, 2008, from OpenGL: http://www.opengl.org/registry/doc/glspec30.20080811.pdf

St-Laurent, S. 2004 *Shaders for Game Programmers and Artists*. Thomson Course Technology, MA, USA.

Todini, E. 1979 Un metodo del gradiente per la verifica delle reti idrauliche. *Boll. Ingegneri Toscana* **11**, 11–14.

Todini, E. & Pilati, S. 1988 A Gradient Algorithm for the Analysis of Pipe Networks. In: *Computer Applications in Water Supply*, Vol. 1 (System analysis and simulation) (B. Coulbeck & C. H. Orr, eds). John Wiley & Sons, London, pp. 1–20.

University of Antwerp 2009 *Fastra 2: The world's most powerful desktop supercomputer*. Retrieved February 12, 2010, from Fastra II: http://fastra2.ua.ac.be/

VR-Zone 2009 *Effects of CPU frequency on FPS (Single GTX 280)*. Retrieved October 28, 2009, from VR-Zone: http://vr-zone.com/articles/effects-of-cpu-frequency-on-fps-single-gtx-280-/7160-1.html

Woodhouse, B. 2002 *Basics of GLUT*. Retrieved July 1, 2009, from GameDev.net: http://www.gamedev.net/reference/articles/article1680.asp

Wu, Z. Y. & Eftekharian, A. A. 2011 Parallel Artificial Neural Network Using CUDA Enabled GPU for Extracting Hydraulic Domain Knowledge of Large Water Distribution Systems. In *Proceedings of World Environmental and Water Resources Congress* on May 22–26, Palm Springs, California, USA.

First received 22 February 2011; accepted in revised form 30 August 2011. Available online 22 November 2011