# Using high performance techniques to accelerate demand-driven hydraulic solvers

Michele Guidolin, Zoran Kapelan and Dragan Savić

## ABSTRACT

Computer models of water distribution networks are commonly used to simulate large systems under complex dynamic scenarios. These models normally use so-called demand-driven solvers, which determine the nodal pressures and pipe flow rates that correspond to specified nodal demands. This paper investigates the use of data parallel high performance computing (HPC) techniques to accelerate demand-driven hydraulic solvers. The sequential code of the solver implemented in the CWSNet library is analysed to understand which computational blocks contribute the most to the total computation time of a hydraulic simulation. The results obtained show that, contrary to popular belief, the linear solver is not the code block with the highest impact on the simulation time, but the pipe head loss computation. Two data parallel HPC techniques, single instruction multiple data (SIMD) operations and general purpose computation on graphics processing units (GPGPU), are used to accelerate the pipe head loss computation and linear algebra operations in new implementations of the hydraulic solver of CWSNet library. The results obtained on different network models show that the use of this techniques can improve significantly the performance of a demand-driven hydraulic solver.

**Key words** | computational performance, CWSNet, demand-driven hydraulic solver, general purpose computation of graphics processing units, single instruction multiple data

**Michele Guidolin** (corresponding author)
**Zoran Kapelan**
**Dragan Savić**
Centre for Water Systems,
College of Engineering,
Mathematics and Physical Sciences,
University of Exeter,
North Park Road,
Exeter EX4 4QF,
UK
E-mail: *M.Guidolin@exeter.ac.uk*

## INTRODUCTION

Demand-driven hydraulic solvers are extensively used in the water engineering field to solve a large array of water distribution system (WDS) problems. The solution to many of these problems requires the execution of a large number of extended period simulations (EPS) to achieve the desired results. This is particularly true for the following type of problems: optimisation of a hydraulic network (e.g. design or model calibration), application of various uncertainty quantification techniques, sensitivity analysis modelling, and risk analyses, etc. Thus, the computational performance of the hydraulic solver used has a large impact on the total execution time.

The most commonly used method to improve the performance of these applications is to execute multiple different simulations in parallel through the use of specific high performance computing (HPC) techniques that perform a task level parallelism. Task parallel techniques compute independent tasks, which have minimal data communication between them, in parallel. They differ from data parallel techniques which compute highly interlinked data in parallel. Two examples of task parallel HPC techniques, which have been implemented in existing hydraulic solvers to achieve parallel tasks execution, are: the message passing interface (MPI) library (Morley *et al.* 2006) and multi-threading (Lopez-Ibanez *et al.* 2008). The former distributes multiple simulations over a cluster of computers/super-computer, or runs simulations in parallel on a multi-core central processing unit (CPU), whereas the latter runs simulations in parallel only on multi-core CPU.

The parallel execution of multiple hydraulic simulations using HPC techniques can improve the performance of WDS applications that require a large number of EPS

doi: 10.2166/hydro.2012.198

runs. However, this increase in speed is directly proportional to the number of different independent simulations that can run in parallel. The number of independent simulations can change greatly depending on the algorithm used and the type of problem to be solved. Thus, the computational performance of the solver in performing a single hydraulic network simulation can still have a significant impact on the total execution time.

The computational performance of a solver can be improved by developing more efficient algorithms, such as the Enhanced Global Gradient Algorithm – EGGA (Giustolisi *et al.* 2011a), or by accelerating an existing algorithm using HPC techniques that execute a data level parallelism. The advantage of using data parallel techniques is that they usually can be combined together with task parallel techniques, thus executing many different simulations in parallel with each simulation computing its data in parallel, in order to improve performance. However, the disadvantage of this type of technique is that not all algorithms employ a significant amount of computation that can be parallelised. Furthermore, these techniques usually sacrifice some of the numerical precision to achieve improved performance.

Since WDS applications are typically executed on a personal computer (PC), it is important that the HPC techniques can be implemented on commonly available hardware. Two significant data parallel HPC techniques, which can be applied to hydraulic solvers and can be used in any common modern PC, are: single instruction multiple data (SIMD) operations, which are available in a modern CPU; and general purpose computation on graphics processing units (GPGPU), which uses the powerful graphic card of a common desktop to execute parallel computation.

The aim of this study is to analyse if it is possible to accelerate (and if so, by how much) a single simulation of a demand-driven hydraulic solver using these two data parallel HPC techniques. The aim is also to understand where in the code it is more convenient to use these techniques in order to achieve high speed-up as not all the computations in a hydraulic solver are data parallel. This analysis can be used to improve the computational performance of existing and future hydraulic solvers, thus they can fully benefit from existing and future computational hardware.

In order to achieve this aim, the demand-driven hydraulic solver of the CWSNet library (Guidolin *et al.* 2010a) is analysed to find the various blocks of the code that are most computationally intensive. This analysis is performed using a low level profiling tool that measures the performance of the code while simulating various hydraulic network models of different sizes. A number of computationally intensive code blocks, which are identified are then analysed to establish which one can take advantage of the two HPC techniques. This is done by checking whether they have a data computation that can be parallelised and whether the improvement will have a significant impact on the total computation time. Then the SIMD and GPGPU techniques are implemented on the chosen blocks of the demand-driven hydraulic solver of the CWSNet library. Finally, the performance of these new implementations is tested on various hydraulic networks.

## DATA PARALLEL HIGH PERFORMANCE TECHNIQUES

### Simple instruction multiple data operations

SIMD (Flynn 1972) operations execute the same instruction on multiple data simultaneously and thus they can achieve high performance by exploiting the eventual data parallelism that is present in an algorithm. These operations are available directly in the CPU through the use of special instructions and registers. Today SIMD operations are available in almost every family of processors; each family has a different set of instructions which have similar functionalities but they are not compatible. Some examples are the SSE instructions set for the x86 processor family (Raman *et al.* 2000), AltiVec for the PowerPc processors family (Fuller 1998) and the NEON technology for the ARM processor family (Goodacre & Sloss 2005).

Typical SIMD operations available are: scalar and floating point arithmetic, logic, comparison, data movement, and load and store (Raman *et al.* 2000). The registers available are usually limited to 128-bit register, i.e. four 32-bit single precision floating point values (floats), or two 64-bit double precision floating point values (double) computed simultaneously. Future designs, like the Advanced Vector

Extensions (AVX) set for the x86 processors, already plan more computationally expensive operations and the use of larger registers (256-bit), thus larger data parallelism (Firasta *et al.* 2009).

SIMD operations are very useful when an algorithm executes a sequence of the same simple operations on a large amount of data. One advantage is that the SIMD registers can load and store data directly from and to the main memory of the computer. Furthermore, it can take advantage of the memory cache system as the normal load and store instructions of the processor. This differs from the GPGPU case as will be shown in the next section.

The main disadvantage of SIMD operations is that not all the algorithms have large sections of data parallel computations available. Furthermore, the SIMD operations are penalised by eventual conditional statements (branches) in the computations, such as an algorithm with a large amount of flow control, and by non-contiguous memory access of the data, i.e. the data are locally sparse in memory.

## General purpose computing on graphics processing units

Since visualisation/graphical problems are inherently parallel, modern graphics cards are composed of hundreds of parallel cores that can execute thousands of threads of computations and thus achieve massive parallelism. For example the Fermi GPU architecture from NVIDIA can have a configuration of 16 multiprocessors each with 32 cores giving a total of 512 cores. On this configuration, each multiprocessor can execute up to 1,536 concurrent threads (Nickolls & Dally 2010).

However, a thread computation on a GPU is not the same as a thread computation on the CPU. Since a GPU core needs to be able to execute many threads simultaneously, GPU threads are simpler, i.e. the number of transistors dedicated to control flow and data management is minimal. Given the large number of threads that need to be executed and the simplicity of their hardware, thread computations on a GPU are grouped together in blocks and there is a penalty for incoherent branching and incoherent data access between threads on the same block (Owens *et al.* 2008). Thus, a data parallel algorithm with many conditional statements (branches) and non-contiguous data access is penalised when executed on a GPU as in the case of SIMD operations.

Modern graphics cards are usually separate devices which are connected to the motherboard through a communication bus. It takes too long for a GPU to compute the data on the main memory directly through this bus; thus, the GPU uses an on-board internal memory. An algorithm that uses the GPU has to move the data from the main memory on the motherboard to the internal memory of the graphics card and vice versa. This data movement can have an impact on the total computation time of a data parallel algorithm that uses a GPU. To hide this delay a technique is to overlap data movement with data computation. However, in order to use this technique efficiently, the amount of data to compute must be very large.

Since data movement probably has the largest impact on the performance of GPU computing, much future development is concerned with removing this bottleneck by integrating the CPU and GPU in a single die/chip on the motherboard that share the same main memory (Brookwood 2010). This development, together with the expected implementation of large SIMD operations with an AVX instructions set, shows how in the future the ability to accelerate data parallel algorithms will be a primary objective of common PC processors.

In order to freely program modern graphics cards, new extensions of programming languages have been developed such us CUDA (NVIDIA Corporation 2011) and OpenCL (Khronos OpenCL Working Group 2011). Both extensions use the idea of an external kernel to indicate a computation on the GPU. A kernel is a function that is executed by a larger number of threads in parallel on the GPU (NVIDIA Corporation 2011). The kernels are launched from the main program; however this is not immediate since a kernel needs to be loaded on the GPU and the threads initialised.

Modern graphics cards can achieve very large speed-ups for specific data parallel algorithms (Nickolls & Dally 2010) which can be of the third order of magnitude (100x). Unfortunately, not every type of algorithm has the data parallel computation necessary to gain these large speed-ups (Lee *et al.* 2010). The specialised architecture of GPUs has some disadvantages that limit the type of computations that can be gained from it. These disadvantages are: (1) the data have to be moved from the main memory of the

motherboard to the on-board memory of the graphic card in order to be computed; (2) the massive data-parallelism can be taken advantage of only if there is a large amount of data to compute and minimal data synchronisation; (3) starting the GPU computation when a kernel is launched incurs in a small overhead; and (4) any change in the flow of execution or any non-contiguous data access between different threads/streams is costly in term of performance.

## CWSNET

CWSNet is a new open source library for hydraulic simulation of pressurised pipe networks. The main aim of the project was to develop a library that was EPANET2 (Rossman 2000) compatible and that takes advantage of an object oriented programming (OOP) model. CWSNet is written in C++ and all the elements of the network, the computations on their attributes and the various computations of the hydraulic modelling are represented as independent objects in the code. Each one of these objects has its own well-defined interface that simplifies the reuse, modification, swapping and upgrade of the code.

The internal structure of CWSNet is logically divided into three layers (see Figure 1): the network layer, the hydraulic solver layer and the mathematical layer. Each layer is composed of many different objects. In order to execute an EPS, an end user interacts mainly with the objects of the first layer (network layer), ignoring the objects in the other layers. However, during an EPS run, the objects of the last two layers will execute the large bulk of the computations, while the objects of the first layer will be mainly used for storing and retrieving network data and for checking the flow of execution (Guidolin *et al.* 2010a).

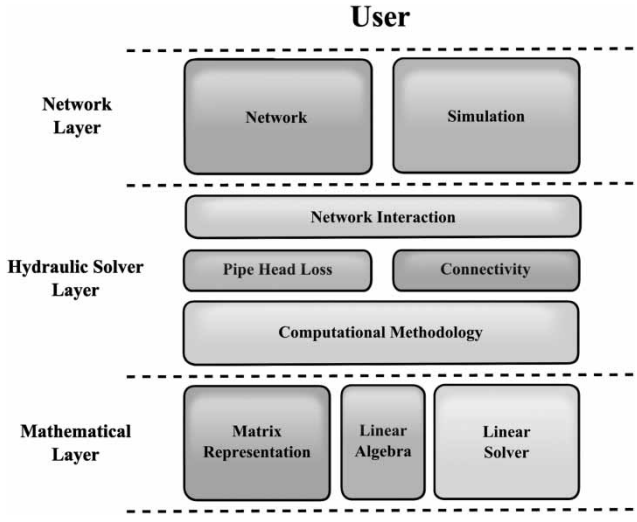


**Figure 1** | CWSNet internal structure.

These features (the use of an OOP model and a well defined structure) make the library easier to extend, either in terms of adding the functionality that currently does not exist in EPANET2, such as the possibility to add new different hydraulic solvers and/or methods for storing/manipulating relevant solver data in the matrix format, or in terms of improving some of the existing EPANET2 functionality, such as the ability in CWSNet to have additional devices (such as variable speed pumps, multiple emitters per junctions, etc.).

CWSNet can read and load hydraulic networks from EPANET2 files. The current version of CWSNet contains two hydraulic solvers: a demand-driven solver which implements the global gradient algorithm (GGA) (Todini & Pilati 1988) and produces numerical results comparable to EPANET2, and a pressure-driven solver which extends the GGA using nodes with demand-pressure dependency (Morley & Tricarico 2008; Rossman 2007). Given the ability to read EPANET2 files and produce numerically comparable results, the users can reuse their existing hydraulic networks with CWSNet.

Another important aim of the CWSNet library is to achieve high computational performance. Thus, CWSNet is thread-safe by design. It is possible in CWSNet to run different simulations of the same network or different networks in parallel. Furthermore, thanks to its extendibility by design, the code is ready to be integrated with new data parallel HPC techniques that can improve the computation performance of the library.

An additional characteristic of CWSNet is that the linear algebra formulae of a hydraulic solver algorithm can be directly represented in the code by using various sparse matrices and vector objects together with multiple vector-to-vector and matrix-vector functions (see Figure 2). While, this direct representation has a small impact on performance, it simplifies the understanding of the methodologies used in existing hydraulic solvers, the implementation of

```
DiagonalMatrix   _d11_inv; // Inverse derivative of the headloss.
EllColumnMatrix  _a21;     // Incidence sub-matrices ...
EllRowMatrix     _a12;     // ... in GGA.
CscMatrix        _lhs;     // Left hand side of the linear system.
...
// lhs = A_{21}*D^{-1}*A_{12}+W+E
_maths->computeLhs(_lhs,_a21, _d11_inv, _a12, _valve_setting_lhs, _emitter_lhs);
```

**Figure 2** | Code example that uses matrices/vectors objects and matrix-vector functions to compute the left hand side matrix of the linear system generated by the GGA model.

new extensions, and the creation of new hydraulic solvers with different methodologies.

The CWSNet code is published using the MIT License (Open Source Initiative OSI 2012). This is an open source and permissive licence; it grants any developer the right to modify the code or reuse it for another open source project as well for commercial software projects. Thus, anyone can inspect the code and contribute to it by fixing bugs or adding new features. Furthermore, the code is cross-platform; thus the library can be used in different operating systems and on different hardware. The code of CWSNet and the documentation of the library are freely available on the website of the Centre for Water Systems (CWS 2012).

## Demand-driven hydraulic solver code analysis

The demand-driven hydraulic solver of CWSNet is represented in the code by the object called HydrauliSolverDdEpanet2. Figure 3 shows a simplified version of the code (pseudo code) used by this hydraulic solver object to compute a steady-state step of the GGA algorithm (Todini & Pilati 1988).

Each steady-state step is composed of a loop of various iterations. This loop is executed until the flows in the links converge to a solution. It is possible to identify in the code various computational blocks that contribute significantly to the total computation time of the hydraulic solver, since they are computed during each iteration of the steady-state step. These blocks are as follows:

- For each node populate the demand and fixed-head vectors;
- For each link initialise status and populate the flow vector;
- Loop until the maximum number of available iterations is reached or converged to solution:
  - For each link prepare the vectors and matrices coefficients used by the GGA model by computing the head loss using the flow vector **(Pipe head loss) (Power/Log functions)**;
  - Prepare the right hand side vector and the left hand side matrix of the linear system using the demand, fixed-head and flow vectors **(Linear algebra)**;
  - Solve the linear system of equations and find the new nodal-head vector **(Linear solver)**;
  - Update the flow vector using the new nodal-head vector **(Linear algebra)**;
  - For each link and node update status using the new flow and nodal-head vectors;
  - If the correction of the flow vector is less than a tolerance, convergence is reached thus exit the loop;
- For each node and link update network elements using results found;
- Check which network elements are not connected to a source node **(Connectivity)**.

**Figure 3** | Pseudo code of a steady-state step of the demand-driven hydraulic solver.

- The linear solver is used to solve the linear system of steady-state equations (**Linear Solver**). Here, the sparse direct Cholesky LL Decomposition Method (CDM) is used in conjunction with a minimum degree reordering algorithm. This method is the same as the linear solver implemented in EPANET2 (Rossman 1999) which is based on (George & Liu 1981). Furthermore, the linear system of equations used in CWSNet is stored in a sparse matrix of compressed sparse column CSC type format (George & Liu 1981; Saad 1990).
- The computation of the head loss for each pipe in the network (**Pipe head loss**). In the CWSNet implementation, the head loss can be computed using three different equations, Hazen-Williams (H-W), Darcy-Weisbach (D-W) and Chezy-Manning (C-M). The characteristics of these equations are that they use operations that are computationally expensive, such as power and logarithm functions (**Power/Log functions**).
- The linear algebra operations used in the GGA method (**Linear Algebra**). In this hydraulic solver, the dense vectors, sparse vectors and diagonal matrix of the GGA are represented by simple arrays while the sparse matrices are of ELL (Ellpack/Itpack) type format (Saad 1990). These matrix-vector operations are not all executed at the same time during a steady-state, but they are separated by other various computations.

Figure 3 also shows that a connectivity of the elements is performed at the end of each steady-state step (**Connectivity**). The connectivity computation is used to check which network elements are connected to at least one water source node. This can be computationally intensive even if it is executed outside the iterations loop. This could happen since the implementation of the connectivity in this hydraulic solver performs a breadth-first search

algorithm (BFS) directly to the network for each source node each time an element of the network has a new status.

One of the main characteristic of the class that defines hydraulic solver objects is that it uses a template to specify the types of the four previous main computations. Thus, it is possible for different hydraulic solvers to change not only the main algorithm but also the type of linear solver, the linear algebra operations, the connectivity algorithm and the way to compute pipe head loss.

## PERFORMANCE ANALYSIS

The impact of various computational blocks of a program can be analysed using a profiling tool. This type of tool, called a profiler, measures various quantities/events during the normal execution of the program. Some of the more common quantities/events measured are the amount of memory used during the execution, the number and type of assembler instructions executed, the amount of cache load and misses and the frequency and duration of function calls. These measurements are then used together with the original code of the program to show directly which computational blocks of the program use more memory or use more CPU time. Thus, profilers are commonly used to aid program performance optimisation.

The performance impact on the total computation time of the computational-intensive blocks of CWSNet (previously described) is analysed using the OProfile application (Levon & Elie 2011). This software is a system-wide profiler for Linux systems, i.e. it can measure all the running programs of the systems as well as the impact of the operating system calls. To be used by OProfile, the CWSNet code does not need to be recompiled with special function calls; since OProfile uses the hardware performance counters of the modern CPU to analyse statistically the performance of an application. Thanks to the use of these special hardware counters, OProfile has a very low overhead.

The performance analysis carried out in this work utilises seven different hydraulic network models of varying sizes; each one is executed using an EPS. Table 1 presents the characteristics of the network models used in the tests: the name of the network, the number of links and nodes, the number of reservoirs (R), tanks (T), pumps (P) and valves (V), the number or operational control rules, the duration of the EPS, the time-step length, and the type of head loss formula used.

The models used in this comparison are: two networks used in the Battle of the Water Sensor Networks (BWSN) (Ostfeld *et al.* 2008); four real-life hydraulic networks from the UK which are commercially confidential; and a schematic representation of the Richmond WDS in the UK (van Zyl *et al.* 2004). All these network models utilise the Hazen-Williams equation to compute the head loss of the pipes, apart one of the real life network (N5) which uses the Darcy-Weisbach equation. The tests are executed on hardware with an Intel Core 2 Quad Q8300 which runs an Ubuntu 10.04 64bit Linux system. The CWSNet library is compiled using the 4.4.3 version of the GNU Compiler Collection.

Figure 4 shows the magnitude of the computation time that the computational blocks (previous introduced) have on the total execution time of the CWSNet demand-driven

**Table 1** | Characteristics of network models used in the performance tests

| ID | Network model | Nodes | Links | R/T/P/V | Controls/Rules | Duration/Step | Head-loss |
|---|---|---|---|---|---|---|---|
| N1 | Richmond skeleton | 48 | 51 | 1/6/7/0 | 14 | 24 h/1 h | H-W |
| N2 | BWSN1 | 129 | 178 | 1/2/2/8 | 5 | 96 h/30 min | H-W |
| N3 | Real life | 1,091 | 1,149 | 1/0/0/0 | 14 | 13 h/1 h | H-W |
| N4 | Real life | 2,182 | 2,266 | 1/0/0/26 | 96 | 24 h/15 min | H-W |
| N5 | Real life | 3,383 | 3,534 | 176/9/19/481 | 167 | 24 h/15 min | D-W |
| N6 | Real life | 8,749 | 9,260 | 4/1/2/175 | 50 | 24 h/1 h | H-W |
| N7 | BWSN2 | 12,527 | 14,831 | 2/2/4/5 | 1,067 | 36 h/1 h | H-W |

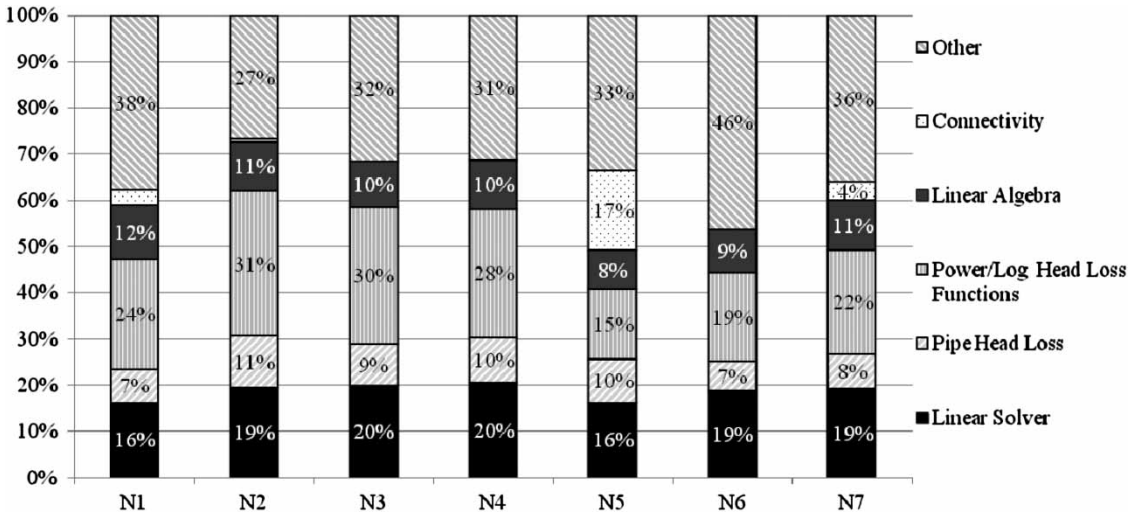


**Figure 4** | Proportions of the total computation time spent on different computations in the seven networks tested as retrieved by using OProfile.

hydraulic solver. The total time includes the initialisation of the solver, the various steady-state steps and the update of the results in the network object. It does not include the loading and initialisation of the network object from the input file, and the output of the results. Furthermore, it does not include the saving of the results at each steady-state step. The results presented in this work are different from those presented in (Guidolin *et al.* 2011), because the version of CWSNet used in the test is a more recent one (version 1.1.0) and the internal data structure has been optimised to minimise the amount of data used.

Until recently, it was commonly thought that the linear solver was the computational block with the highest impact on the total computation time of a demand-driven hydraulic solver. Thus, a large amount of research has been carried out in order to optimise the performance of the linear solver using HPC techniques (Crous *et al.* 2008; Crous 2009; Guidolin *et al.* 2010b; Wu & Lee 2011) or in order to find the best type of linear solver to achieve a fast and robust solution (Giustolisi *et al.* 2011b). However, as shown in Figure 4 by the analysis in this work and by computational analysis of a previous version of CWSNet (Guidolin *et al.* 2011), the linear solver impact on the total computation time is never over 20%. These results agree with the findings of Wu & Lee (2011) who analysed the computational time of the demand-driven hydraulic solver implemented in the EPANET2 software. They found that in EPANET2 'only 13% of the total run time is used by the linear solver' in the case of a network of 250,000 pipes. The total run time included the loading of the input file and the output of the report.

It is possible to see in Figure 4 that the power and log function computations have a significant impact on the total computation time. These functions are part of the pipe head loss computations and together they represent 25–42% of the total computation time. Thus, the various optimisation efforts should be concentrated on the pipe head loss computation. The network where the pipe head loss computation has the lowest impact is N5 which uses the Darcy-Weisbach formula. This formula has different execution paths depending on the Reynolds number; thus the amount of computation performed depends on the type of flow on the pipe. Only when there is a turbulent flow in rough conduits (Reynolds number greater than 4,000), both power and log functions are used. While turbulent flow can be very frequent in WDSs, in the case of network N5 this is happening around half of the time during the EPS. During the other half of the time, the flow in the pipes of network N5 is laminar. Thus these two computational expensive functions are used less frequently in network N5 than in the other networks that use the Hazen-Williams formula.

After the pipe head loss formula and the linear solver computations, it is the various linear algebra operations that have a large impact on the total run time; as a group

they represent from 8 to 12% of the hydraulic solver's computational time. However, taken independently, each linear algebra operation represents less than 2% of the computational time. Of the previously mentioned computational blocks, connectivity computation has the lower impact on the total computation time. This is mainly due to fact that this computation is only executed when there is a status change between links and nodes, and in these networks it does not happen in every steady-state step. The N5 network is an example where the status of the links and nodes change very frequently, thus it is possible to see that the connectivity has a large impact in the total computation time, i.e. 17%.

These performance tests also show that collectively all the other computations have a large impact on the total computation time of the hydraulic solver. This is particularly true for the N6 network where it accounts for more than 40% of the computation time. Some of these computations are: the iteration through the elements for a status check, the updating of the network results, the creation of the linear system of equations to solve, the reordering of nodes, etc. Furthermore, the amount of data used can have an impact on the number of cache misses during the iterations on the elements (i.e. failure to locate requested data in cache memory). Since these computations are very small taken singularly and they are very different and sparse in the code, the only way to improve their performance is to optimise the code of the algorithm by changing the internal data structures or by changing the algorithm itself.

## Applicability of the data parallel high performance techniques on computational blocks

As already stated, data parallel HPC techniques have some disadvantages; thus not every type of computation can benefit from implementing them. In this section, the computational blocks are analysed to establish which ones can take advantage from the two HPC techniques and which ones will benefit from using the two techniques to achieve a higher impact on the total computation time. Then using this analysis, new improved versions of the demand-driven hydraulic solver will be implemented using the SIMD operations and the GPU computation techniques applied to suitable computational blocks.

### Connectivity

While this type of computation is composed of many conditional statements, the implementation of BFS algorithm on GPU has been successful (Harish & Narayanan 2007) using one thread per node. However, these results were obtained using a large number of nodes (i.e. millions). Given that a typical hydraulic network is composed of several hundred/thousand nodes and that the connectivity impacts on the total computation time only in particular cases, the new versions of the hydraulic solver will use the original code for the connectivity computation.

### Linear algebra

The operations between vectors and matrices represented by a single array are well suited to gain from SIMD operations and GPU computation since an operation can be computed independently for each index. Unfortunately, the linear algebra operations that have an impact on the computation time are the ones performed on the sparse matrices. Given that in the sparse matrices the access on the non-zeros values is not contiguous, the gains obtained by using SIMD operations could be minimal. Furthermore, the time consumed on computing linear-algebra operations is only around 10% of the total computation time, i.e. any gain from the use of SIMD operations on this block of the code could probably have a very small impact on the total computation time. However, linear algebra operations using SIMD operations will be implemented and tested in the new improved hydraulic solvers since they are trivial to implement and it is interesting to see their impact on the computational time.

In the case of GPU computation, there is also a problem in that the majority of the linear algebra operations are separated from each other by some local computations on the vectors. Thus for each iteration, the code would need to execute multiple movements of data between the main memory and the GPU memory. Since these movements limit the gain of performance achieved by the GPU computation, the GPU version of the improved hydraulic solver will not use GPU computation for these operations.

## Linear solver

The existing techniques used to implement the parallel version of the sparse CDM are mainly based on elimination trees (Heath *et al.* 1991) which are built using graph theory methods on the sparse matrix of the linear problem. The speed up which can be obtained using these techniques depends on the size and sparsity of the matrix. Thus, a parallel version of the sparse CDM is not easy to implement and the performance gains could be null or minimal considering the small size of the linear system of equations to solve produced by the GGA method. For example Vuduc *et al.* (2010) show that the performance results from a preliminary Cholesky factorisation that uses GPU computation are less than a 2X speed-up over a CPU implementation, in the case of the smaller sparse matrix. The number of non-zeros of this matrix is two orders of magnitude larger (Guney 2010) than the number of non-zeros of the sparse matrix generated by the larger network tested in this work.

An alternative solution is to implement a different linear solver which is executed more suitably using SIMD operations or GPU computation, such as the conjugate gradient method (CGM) or any other iterative method. The iterative method type solvers can be implemented using matrix-vector operations which are well suited to gain from these two techniques and simpler to implement. A recent study shows that a CGM implemented on GPU outperforms a CGM implemented on CPU, using dense matrices that are 'equivalent to the coefficient matrix used in the GGA' (Crous 2009), only in exceptionally large networks. A different study, performed using sparse matrices generated by EPANET2 from real life and synthetic networks, shows as well that a CGM with a Jacobi preconditioner implemented on GPU is faster than the CPU version when the sparse matrices are sufficiently large (Guidolin *et al.* 2010b).

Neither of these studies compare the performance of the CGM to the CDM with minimum degree reordering implemented in CWSNet and EPANET2. However, (Wu & Lee 2011) show that a parallel CGM solver without preconditioner applied to EPANET2 'comes nowhere close to the performance obtained using the linear EPANET [sic] solver'. Giustolisi *et al.* (2011b) also show that a CDM with LDL decomposition and an approximate minimum degree permutation (AMD) ordering algorithm 'appear the best performing method for WDN [water distribution network] analysis'. In their tests the Cholesky LDL decomposition method with different ordering strategies outperforms any other iterative methods on their Matlab implementation of a GGA hydraulic solver. However, the authors suggest that the possibility to parallelise the CGM is attractive, considering that selecting a proper pre-conditioner has an impact on the reliability and parallelisation of the algorithm, and thus they are planning to test a GPU implementation of CGM with ILUPT as pre-conditioner (Incomplete LU factorisation with Threshold and Pivoting) on a very large network (more than 100,000 nodes).

The above studies show that the CDM linear solver appears to be the most suitable method to solve the linear system of equations generated by a GGA. However, a parallel CDM using SIMD operation and GPU computation is not easy to implement and does not guarantee high performance for the relatively small sparse matrices generated by the GGA. Given these problems, and the fact that the existing CDM linear solver of CWSNet contributes only around 20% of the total computation time; the new versions of the hydraulic solver will use the original code for the linear solver computation. The assumption that it is preferable to parallelise other operations instead of the linear solver was also one of the conclusions of the work of (Giustolisi *et al.* 2011b).

## Pipe head loss and Power/Log functions

The block of the code which contributes the most to the total computation time of the hydraulic solver is the pipe head loss computation together with the power and logarithm functions, which are part of it. The head loss of a pipe can be computed using three different equations; in this work only two are analysed: Hazen-Williams (H-W) and Darcy-Weisbach (D-W). Both equations can achieve higher performance using SIMD operations and GPU computing since they are highly data parallel, i.e. the head loss is calculated independently between pipes. The Chezy-Manning equation is not analysed since it is more commonly used for open channel flow and thus it is not as frequently used as the other two equations.

The H-W and D-W equations have different computational characteristics. The latter is more complex to code using the data parallel HPC techniques since it is composed of different execution paths depending on the Reynolds number. The former is simpler but more computationally intensive since the very expensive power function is executed each time. Figure 4 shows that power and logarithm functions are computations that contribute greatly to the total computation time thus it is important to be able to parallelise them.

Both equations are very simple to parallelise using GPU computing, since power and logarithm functions are directly available on CUDA and OpenCL. The execution path in the GPU code can be easily programmed using conditional statements as in the case of a normal sequential code, and the data management is simple since the data needed for each pipe is taken from the same index in different arrays (the data are contiguous). However, the Darcy-Weisbach implementation could incur a performance penalty since there could be incoherent branching between threads of the same block. Furthermore, for each iteration the data have to be moved from the CPU main memory to the on board memory of the graphic cards and then back. Therefore, the gain obtained by the GPU computation could be limited.

In the case of SIMD operations, power and logarithm functions are not directly available since the set of instructions include only basic floating point arithmetical operations. It is possible to approximate these functions using polynomial with basic operations (Davidson *et al.* 1999); however it is not simple to achieve the performance and accuracy desired using this technique. Fortunately, it is possible to access optimised SIMD version of the power and logarithm function using the Intel C++ Compiler version 12.0 (Intel Corporation 2011), the Accelerate Framework for Mac OS X (Apple Inc. 2011) or the AMD libM library (Advanced Micro Devices, Inc. 2011).

Considering the characteristics of the pipe head loss computation and the availability of power and logarithm functions that use SIMD operations and GPU computation, the new improved hydraulic solvers will implement the pipe head loss computation using these HPC techniques.

## IMPLEMENTATION

The modifications made to the original code in the improved hydraulic solvers, which use the data parallel HPC performance techniques, are: (1) the linear-algebra operations and the pipe head loss computation for the SIMD operations; (2) only the pipe head loss computation for the GPU computation.

In this section, the implementations of the pipe head loss computation using the SIMD operations and the GPU computation techniques are presented. In the case of the SIMD operations version, the implementation uses the SSE instructions set for the x86 processors family together with the AMD libM library for the power and logarithm functions. In the case of the GPU computation version, the implementation uses the C for CUDA programming language version 4.0.

The two pipe head loss equations need various values for each pipe to be computed, such as pipe flow, minor loss coefficient, resistance coefficient, roughness coefficient, and pipe diameter. These values are stored in different arrays where each index in the arrays represents the values of a specific pipe. The values that do not change between steady-state steps are initialised and computed only once during the initialisation of the hydraulic solver.

The object that computes the head loss of the pipes in the network is composed of two methods: *prepareHeadLoss()* and *computeHeadLoss()*. The first method is called before the code that loops through each link to compute the coefficient using the pipe head loss, see the first step after the loop in Figure 3. In the original sequential code, this method does not perform any action. The second method is called each time the head loss of a pipe is requested during the computation of the vectors and matrices for the GGA.

In the case of the new implementations that use HPC techniques, the main computation is performed in all the pipes by the *prepareHeadLoss()* method. The resulting values are stored in three arrays. Then, when the *computeHeadLoss()* method is called by a specific pipe, the result values are retrieved from these three arrays. This implementation has the advantage that the head loss is calculated in all the pipes at the same time without any other extra

```
for(i=0; i<n_pipes; i=i+2) {
 __m128d fl       = _mm_load_pd(&flow[i]);
 __m128d ml       = _mm_load_pd(&minor_loss[i]);
 __m128d res      = _mm_load_pd(&resistance[i]);

 __m128d res_term = _mm_mul_pd(res,__vrd2_exp(_mm_mul_pd(__vrd2_log(fl), hw_exp )));
 __m128d ml_term  = _mm_mul_pd(ml, _mm_mul_pd(fl, fl));
 __m128d hl_term  = _mm_add_pd(res_term, ml_term );
 __m128d grad_term= _mm_add_pd(_mm_mul_pd(hw_exp, res_term), _mm_mul_pd(two, ml_term));

 __m128d grad_inv = _mm_div_pd(fl, grad_term);
 __m128d pro      = _mm_div_pd(hl_term, grad_term);

 _mm_stream_pd(&head_loss[i], hl_term);
 _mm_stream_pd(&gradient_inverse[i], grad_inv);
 _mm_stream_pd(&product[i], pro);
}
```

**Figure 5** | SIMD operations version of the prepareHeadLoss() method which computes H-W equation.

execution between pipe computations. It has the disadvantage that the head loss is computed also for closed links.

Figure 5 shows the code of the SIMD operations version of the *prepareHeadLoss()* method that computes the H-W equation. The *__m128d* type represents a 128-bit register which contains two double precision floating point values. The methods starting with *_mm* prefix represent intrinsic functions that perform SIMD operations, while the *_pd* postfix indicates that the instructions use two double precision values.

The code contains a loop that iterates through each pipe by stepping two pipes at time ($i = i + 2$). The first three lines of code load for each input array provides the values from two pipes simultaneously. Then the various parts of the equation are computed using two values at a time. Finally the results are written back to each output array by two values at a time. The methods that have the *__vrd2* prefix are special methods that work on two double precision floating point values from the libM library. This library does not directly contain a power function; however it is possible to compute the power values using the exponent and logarithm functions.

Figure 6 shows the code of the GPU computation version of the *prepareHeadLoss()* method that computes the H-W equation. The first step in a GPU computation is to decide the number of threads needed, which in this case is one for each pipe in the network, and to partition the threads into blocks that can be solved independently (NVIDIA Corporation 2011). The first two lines of code in Figure 6 set the number of threads per block and the total number of blocks, which depends on the number of pipes.

```
int threads = 128;
int blocks  = (n_pipes + threads - 1) / threads;

copyHostToDevice(flow);

deviceKernelComputeHW<<<blocks,threads>>>(n_pipes, EXPONENT,
                                          flow, minor_loss,
                                          resistance, head_loss,
                                          gradient_inverse, product);
copyDeviceToHost(head_loss);
copyDeviceToHost(gradient_inverse);
copyDeviceToHost(product);
```

**Figure 6** | GPU computation version of the *prepareHeadLoss()* method which computes the H-W equation.

In this case, the number of threads per block is fixed at 128 since this is the value that gave better performance in the majority of the tests performed.

In the third line of code of Figure 6, the values of array containing the flow of each pipe are copied into the memory of the GPU. The *copyHostToDevice()* and *copyDeviceToHost()* are internal methods that, respectively, move an array of values from the main memory to the on board memory of the GPU and vice versa. Then, the H-W equation is computed on the GPU using the *deviceKernelComputeHW()* method which launch the kernel that calculates the H-W equation for each pipe. This method uses specific CUDA language extensions (<<< and >>> tokens) to define the number of threads per block and total number of blocks used. Finally, in the three last lines of code, the values of the arrays containing the results are copied back from the GPU memory into the main memory.

Figure 7 shows the code of the kernel which computes the H-W equation. This code is very similar to the sequential code. The main difference is that the index of the pipe ($i$) is retrieved using the built-in variables *blockDim*, *blockIdx*,

```
__global__ void deviceKernelComputeHW(const size_t N, const double exp,
                                      const double* flow, const double* minor_loss,
                                      const double* resistance, double* head_loss,
                                      double* gradient_inverse, double* product)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i<N) {
    double resistance_term = resistance[i] * pow(flow[i], exp);
    double minor_loss_term = minor_loss*flow[i]*flow[i];
    double head_loss_term  = (resistance_term + minor_loss_term);
    double gradient_term   = (exp*resistance_term + 2.0*minor_loss_term);

    head_loss[i]         =  head_loss_term;
    gradient_inverse[i]  =  flow[i] / gradient_term;
    product[i]           =  head_loss_term / gradient_term;
  }
}
```

**Figure 7** | Kernel that computes the H-W equation for each pipe in the GPU.

and *threadIdx* which return the number of threads in a block, the number of the blocks being computed, and number of threads inside the block that is executing the kernel respectively. Thus, these three variables are used to uniquely identify a thread. Another characteristic of this code is that the number of threads executed is probably more than the number of pipes in the network, since the threads per block is a multiple of 32. Thus, the code contains a conditional statement which checks that threads executing the kernel are the ones with a unique number lower than the number of pipes to compute.

The code examples shown in Figures 5 and 7 are a simplification of the original CWSNet code. These examples do not show, for length reasons, the conditional statement that protects against the division by zero and the computation of the absolute value of the flow when there is respectively no flow or negative flow in the pipes. The conditional statement and the absolute value computation are executed in parallel in both SIMD and GPU cases.

## PERFORMANCE RESULTS OF IMPROVED VERSION AND DISCUSSION

The performance gains achieved by the use of SIMD operations and GPU computation are evaluated by comparing the average computational times obtained by the original demand-driven hydraulic solver of CWSNet to the ones obtained by the new improved versions of the demand-driven hydraulic solver. The four improved versions tested in this work are: the hydraulic solver that uses SIMD operations only for the linear algebra operations (SIMD-LA); the one that uses SIMD operations only for the pipe head loss computation (SIMD-HL); the hydraulic solver that uses SIMD operations for both previous computational blocks (called SIMD-LA-HL); and the hydraulic solver that uses GPU computation (GPU-HL) only for the pipe head loss computation.

The networks used in the tests are those from Table 1 and the hardware is the same one used in the previous performance analysis tests with the addition of an NVIDIA GeForce GTX 285 GPU for the GPU computation. The improved versions of the hydraulic solver use double precision floating point values in the computations as the original code; however both techniques sacrifice some numerical precision in order to achieve high performance. Therefore in the comparisons, the maximum relative error ($E$) in the nodal heads between the original hydraulic solver and the improved one is evaluated for each network using Equation (1), where $n$ is the number of nodes in the network, *Ho(i)* is the nodal head computed by the original hydraulic solver for node $i$ and *Hn(i)* is the nodal head computed by the new improved hydraulic solver tested at the same node $i$.

$$E = \max_{i=1,n} \frac{|Ho(i) - Hn(i)|}{\left|\frac{Ho(i) + Hn(i)}{2}\right|} \quad (1)$$

Tables 2 and 3 present the results of the tests with the hydraulic solvers using SIMD and GPU computations,

**Table 2** | Computational time and relative error obtained by the hydraulic solvers using SIMD operations

| | Original CWSNet | | SIMD-LA | | | | SIMD-HL | | | | SIMD-LA-HL | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | T (ms) | S | T | Imp (%) | E | S | T | Imp (%) | E | S | T | Imp (%) | E | S |
| N1 | 16.2 | 91 | 15.8 | 2.5 | $2.36\times10^{-8}$ | 91 | 13.7 | 15.4 | $2.83\times10^{-8}$ | 91 | 13.8 | 14.8 | $2.71\times10^{-8}$ | 91 |
| N2 | 51.3 | 204 | 51.1 | 0.4 | $1.14\times10^{-8}$ | 204 | 38.1 | 25.7 | $1.27\times10^{-9}$ | 204 | 38.0 | 25.9 | $8.24\times10^{-9}$ | 204 |
| N3 | 19.4 | 14 | 18.7 | 3.6 | $5.82\times10^{-8}$ | 14 | 14.6 | 24.7 | $2.44\times10^{-6}$ | 14 | 14.2 | 26.8 | $7.45\times10^{-8}$ | 14 |
| N4 | 393.8 | 97 | 381.2 | 3.2 | $9.27\times10^{-7}$ | 97 | 289.7 | 26.4 | $3.93\times10^{-7}$ | 97 | 283.1 | 28.1 | $9.48\times10^{-7}$ | 97 |
| N5 | 934.3 | 155 | 915.3 | 2.0 | $1.94\times10^{-7}$ | 158 | 848.7 | 9.2 | $3.63\times10^{-7}$ | 156 | 829.1 | 11.3 | $2.75\times10^{-7}$ | 155 |
| N6 | 1,409.8 | 122 | 1,356.4 | 3.8 | $2.13\times10^{-6}$ | 122 | 1,227.3 | 12.9 | $2.02\times10^{-6}$ | 122 | 1,212.8 | 14.0 | $2.28\times10^{-6}$ | 122 |
| N7 | 1,459.6 | 44 | 1,452.9 | 0.5 | $9.40\times10^{-6}$ | 44 | 1,248.6 | 14.5 | $5.14\times10^{-8}$ | 44 | – | – | – | – |

**Table 3** | Computational time and relative error obtained by the hydraulic solvers using GPU computation

| | Original CWSNet | | GPU-HL | | | |
|---|---|---|---|---|---|---|
| ID | T (ms) | S | T | Imp (%) | E | S |
| N1 | 16.2 | 91 | 105.7 | −552.5 | $3.40\times10^{-8}$ | 91 |
| N2 | 51.3 | 204 | 140.7 | −174.3 | $1.63\times10^{-9}$ | 204 |
| N3 | 19.4 | 14 | 60.1 | −209.8 | $5.01\times10^{-8}$ | 14 |
| N4 | 393.8 | 97 | 340.5 | 13.5 | $4.02\times10^{-7}$ | 97 |
| N5 | 934.3 | 155 | 850.6 | 9.0 | $1.29\times10^{-7}$ | 158 |
| N6 | 1,409.8 | 122 | 1,210.7 | 14.1 | $1.18\times10^{-6}$ | 122 |
| N7 | 1,459.6 | 44 | 1,184.8 | 18.8 | $1.24\times10^{-7}$ | 44 |

respectively. In these tables, *T* is the average mean execution time in milliseconds (ms) between 10 executions of the hydraulic solver analysed; *Imp* is the percentage of computational time improvement obtained by the improved hydraulic solver as compared to the original one; and S is the number of steady-state steps performed. As in the previous tests, the execution time includes only the initialisation of the solver, the various steady-state steps and the update of the results in the network object.

Tables 2 and 3 show that the maximum relative error between the improved versions and the original hydraulic solver is never higher than $1.0\times10^{-6}$. The numerical precision lost by the HPC techniques could be considered insignificant. However, this small difference between the results of the original computation and the results of the computation using HPC techniques could change the outcome in the case of networks that are not well balanced. This is the case when valves, pumps, or pipes keep switching their status from one iteration to the next and thus the hydraulic solver might not converge to a solution within the maximum number of iterations allowed. For example, in Table 2, the results of the hydraulic solver that uses SIMD operations for both linear algebra operations and pipe head loss computation are not shown for the network N7, since the hydraulic solver does not converge, i.e. SIMD-LA-HL reach the maximum number of iterations allowed without converging while the other versions of the solvers converge just before the maximum number of iterations. Furthermore, Table 2 shows that the HPC techniques can have an impact on the total number of steady-state steps performed (see e.g. steady-state steps for N5). This is due to a small difference in numerical computations that activate rules and controls at different times during the EPS.

As can be seen in Table 2, the hydraulic solver that uses SIMD operations for the linear algebra operations only, is only slightly faster than the original one. The improvements obtained by SIMD operations should be directly related to the number of values that are computed simultaneously by the SIMD operations (two 128-bit registers equivalent to four double precision values). Given that the linear algebra operations block represents between 8% (N5) and 12% (N1) of the total computational time (see Figure 4), the improved hydraulic solver should achieve between 4 and 6% faster computations than the original one. However, as previously stated, the non-contiguous access of data of the linear algebra operations is not ideal for the SIMD operations.

The hydraulic solver that uses SIMD operations only for the pipe head loss computation is significantly faster than the original one, from 9.2% (N5) to 26% (N4) faster. This is due to the head loss computational block, which includes power and logarithm functions, representing between 25% (N5) and 42% (N2) of the total computational time. This confirms that the head loss computations for each pipe are highly data parallel; thus can achieve good results using SIMD operations.

The lowest time improvement is obtained on network N5; this network uses the D-W equations for the pipe head loss computation, which follows different execution paths depending on the Reynolds number. This code with many conditional statements did not gain much from the use of SIMD operations. In the case of large networks, the improvements are smaller, since the head loss computation and the linear algebra operations represent a small amount of the total computation time. Furthermore, the large amount of data in the larger network increases the cache miss in the computations and these misses limit the effectiveness of the SIMD operations.

The improved demand-driven hydraulic solver that uses SIMD operations for both computational blocks achieves between 11% (N5) and 28% (N4) faster computation than the original hydraulic solver. The improvements obtained using SIMD operations for both computational blocks are not a direct sum of the improvements obtained by using SIMD operations in the two blocks separately. This is more evident in the results obtained by the smallest network. A possible explanation is that the use of SIMD operations in different blocks of code changes the data access and thus the management of the cache between implementations; this could result in a different percentage of improvements.

In the results shown in Table 3 for the GPU version of the improved hydraulic solver, it is possible to see that the computational times for smaller networks are higher than the ones of the original version. This is happening because the amount of data is so small that the performance gains obtained by the GPU computation are not big enough to compensate for the performance lost due to data movements and the launch of the kernels. By increasing the amount of data required to compute with larger network, the GPU version achieves faster computations than the original version and the SIMD one. However, these speed-ups are not as high as the difference in theoretical performance between the GPU and CPU would indicate. Unfortunately, the data movements during each iteration have a high impact on the total computational performance even in the case of large networks.

Table 4 presents the comparisons between the execution times obtained with EPANET2 and those obtained with CWSNet hydraulic solvers (the original and the best timings from the improved hydraulic solvers). The comparison with EPANET2 is intended to show how the original and the improved CWSNet fare in comparison with the state of the art demand-driven hydraulic solver in the field. In this table, *Imp* and the maximum relative error (*E*) use EPANET2 as base. It is shown that the execution time of the original CWSNet hydraulic solver is slower than EPANET2. This is due to EPANET2 being designed to be fast, while CWSNet was designed to be flexible and

**Table 4** | Comparison between EPANET2 and CWSNet demand-driven hydraulic solvers

| | EPANET2 | | Original CWSNet | | | | Improved CWSNet (best timings) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | *T (ms)* | *S* | *T* | *Imp (%)* | *E* | *S* | *T* | *Imp (%)* | *E* | *S* | *Version* |
| N1 | 14.1 | 91 | 16.2 | −15.1 | $7.24\times10^{-8}$ | 91 | 13.8 | 1.9 | $7.24\times10^{-8}$ | 91 | SIMD-LA-PHL |
| N2 | 47.2 | 207 | 51.3 | −8.7 | $4.12\times10^{-5}$ | 204 | 38.0 | 19.5 | $4.12\times10^{-5}$ | 204 | SIMD-LA-PHL |
| N3 | 16.2 | 14 | 19.4 | −19.7 | $9.26\times10^{-8}$ | 14 | 14.2 | 12.4 | $1.10\times10^{-7}$ | 14 | SIMD-LA-PHL |
| N4 | 220.9 | 97 | 393.8 | −78.2 | $8.34\times10^{-7}$ | 97 | 283.1 | −28.1 | $1.36\times10^{-6}$ | 97 | SIMD-LA-PHL |
| N5 | 665.6 | 164 | 934.3 | −40.4 | $1.12\times10^{-2}$ | 155 | 829.1 | −24.6 | $1.12\times10^{-2}$ | 155 | SIMD-LA-PHL |
| N6 | 523.6 | 40 | 1,409.8 | −169.3 | $2.52\times10^{-6}$ | 122 | 1,210.7 | −131.2 | $1.56\times10^{-6}$ | 122 | GPU-HL |
| N7 | 1,230.9 | 44 | 1,459.6 | −18.6 | $1.07\times10^{-3}$ | 44 | 1,184.8 | 3.7 | $1.07\times10^{-3}$ | 44 | GPU-HL |

extensible. This flexibility requires additional computations, such as the linear algebra operations, and the use of additional resources such as computer memory for intermediate vectors-matrices; thus it comes at a performance cost. This is illustrated in Table 4 on the cases of networks N4 and N5. In addition to this, EPANET2 solves network N6 with a decreased number of steady-state steps than the two CWSNet hydraulic solvers. This is due to the fact that the controls and rules of this network do not work in CWSNet the same way as in EPANET2. In the case of network N6, controls and rules in CWSNet are activated roughly every 6 minutes as opposed to every 30 minutes as in EPANET2, thus increasing significantly the number of steady-state steps. However, despite all of the above, due to the use of HPC technique, the improved CWSNet hydraulic solvers can still match and outperform EPANET2 in other cases for both small (N1-N3) and large networks (N7).

## CONCLUSIONS

This work shows that data parallel HPC techniques can be used successfully to accelerate demand-driven hydraulic solvers. However, these techniques need to be used for computations that are highly data parallel in order to gain computational performance. The HPC techniques studied are: single instruction multiple data (SIMD) operations and general purpose computing on graphics processing units (GPGPU).

While until now the typical chosen candidate for using data parallel HPC techniques was the linear solver, this work indicates that the ideal computation to gain from these techniques is the calculation of the head loss in each pipe. The pipe head loss computation is not only highly data parallel (i.e. the calculation is done independently in each pipe), but it has also the higher impact in the total computation time of the hydraulic simulation. In the case of the demand driven hydraulic solver of the CWSNet library, the pipe head loss computation represents from 23 to 42% of the execution time depending on the size of the network and type of head loss equation used.

The tests performed show that the new implementation of the CWSNet hydraulic solver using SIMD operations on the pipe head loss computation obtains from 9 to 26.5% faster execution than the original sequential hydraulic solver, depending on the size of the network and type of head loss equation used. Since SIMD operations are relatively simple to implement, they are a good technique to use in existing hydraulic solver applications in order to achieve higher performance.

Where GPU computation is used to calculate the head loss of the pipes, the new implementation of the hydraulic solver that uses GPGPU obtains faster executions only when the network computed is significantly large. For networks larger than 2,000 pipes, the executions are from 9.0 to 18.8% faster than the original sequential hydraulic solver. The possible performance gains that can be obtained by using the large computational power of the GPU are limited in this particular implementation by the need to move data from the main memory of the CPU to the on board memory of the GPU in each iteration of a hydraulic solver step.

In order to further increase the computational performance of future implementations of hydraulic solvers, GPU computation is the technique that has the higher potential for performance improvements. These further improvements could be obtained because of the increasing trend in CPU design to integrate the CPU and GPU in a single die/chip on the motherboard. This could remove the performance bottleneck of data movement between CPU and GPU.

Alternatively, a more elaborate re-design of a demand-driven hydraulic solver could achieve high computational performance by using GPU computation. This re-design could allow a larger number of possible computations to be carried out directly on the GPU. One example is to use the GPU for the computation performed in each element of the network and the preparation of vectors and matrices (coefficients) used by the GGA, with only the minimum of the necessary sequential code left to be executed on the CPU.

## REFERENCES

Advanced Micro Devices, Inc. 2011 *AMD LibM*. Available from: http://developer.amd.com/libM.

Apple Inc. 2011 *Taking Advantage of the Accelerate Framework*. Available from: https://developer.apple.com/performance/accelerateframework.html.

Brookwood, N. 2010 AMD Fusion Family of APUs: enabling a superior, immersive PC experience. *Insight* **64**, 1–8.

CWS 2012 CWSNet Library. Available from: http://centres.exeter.ac.uk/cws/cwsnet.

Crous, P. A. 2009 Application of Stream Processing to Hydraulic Network Solvers. Masters Thesis, University of Johannesburg, Johannesburg, South Africa.

Crous, P. A., van Zyl, J. E. & Nel, A. 2008 Using stream processing to improve the speed of hydraulic network solvers. In: *ASCE Conf. Proc.* (K. Van Zyl, ed.). ASCE, Kruger National Park, South Africa, p. 71. Available from: http://link.aip.org/link/?ASC/340/71/1 (accessed December 14, 2010).

Davidson, J., Savic, D. A. & Walters, G. 1999 Method for the identification of explicit polynomial formulae for the friction in turbulent pipe flow. *Journal of Hydroinformatics* **1**, 115–126.

Firasta, N., Buxton, M., Jinbo, P., Nasri, K. & Kuo, S. 2009 Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel Software Network*. Available from: http://software.intel.com/en-us/articles/intel-avx-new-frontiers-in-performance-improvements-and-energy-efficiency/.

Flynn, M. J. 1972 Some computer organizations and their effectiveness. *IEEE Transactions on Computers* **C-21**, 948–960.

Fuller, S. 1998 Motorola's AltiVec™ Technology. *White Paper* May, 6. Available from: www.iele.polsl.pl/elenota/Motorola/altivecwp.pdf.

George, A. & Liu, J. W. H. 1981 *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, New Jersey, USA.

Giustolisi, O., Berardi, L., Laucelli, D. & Savić, D. A. 2011a A computationally efficient modeling method for large size water network analysis. *Journal of Hydraulic Engineering* **138**, 313–326.

Giustolisi, O., Savic, D. A., Laucelli, D. & Berardi, L. 2011b Testing linear solver for WDN models. In: *Urban Water Management: Challenges and Opportunities. Computing and Control for the Water Industry 2011 (CCWI 2011)*, 5–7 September 2011, Exeter, Vol. 3, pp. 817–822.

Goodacre, J. & Sloss, A. N. 2005 Parallelism and the ARM instruction set architecture. *Computer* **38**, 42–50.

Guidolin, M., Burovskiy, P., Kapelan, Z. & Savić, D. A. 2010a CWSnet: An object-oriented toolkit for water distribution system simulations. In: *11th International Water Distribution System Analysis conference*, WDSA 2010, Tucson, AZ, USA, pp. 1–13.

Guidolin, M., Kapelan, Z., Savić, D. A. & Giustolisi, O. 2010b High performance hydraulic simulations with epanet on graphics processing units. In: *Proc. 9th International Conference on Hydroinformatics*, Tianjin, China, pp. 897–904.

Guidolin, M., Savic, D. A. & Kapelan, Z. 2011 Computational performance analysis and improvement of the demand-driven hydraulic solver for the CWSNet library. In: *Urban Water Management: Challenges and Opportunities. Computing and Control for the Water Industry 2011 (CCWI 2011)*, 5–7 September 2011, Exeter, Vol. 1, pp. 45–50.

Guney, M. E. 2010 High-performance Direct Solution of Finite Element Problems on Multi-core Processors. PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia, USA.

Harish, P. & Narayanan, P. 2007 Accelerating large graph algorithms on the GPU using CUDA. High Performance Computing–HiPC 2007, pp. 197–208.

Heath, M. T., Ng, E. & Peyton, B. W. 1991. Parallel algorithms for sparse linear systems. *SIAM Review* **33**, 420–460.

Intel Corporation 2011 *Intel C++ Compiler 12.0 User and Reference Guides*. Available from: http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/index.htm.

Khronos OpenCL Working Group 2011 The OpenCL specification version 1.1. Available from: www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. & Dubey, P. 2010 Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: *ACM SIGARCH Computer Architecture News*, pp. 451–460.

Levon, J. & Elie, P. 2011 OProfile, a system-wide profiler for Linux systems. *Homepage*: http://oprofile.sourceforge.net.

Lopez-Ibanez, M., Prasad, D. T. & Paechter, B. 2008 Parallel optimisation of pump schedules with a thread-safe variant of EPANET toolkit. In: *ASCE Conf. Proc.* (K. Van Zyl, ed.) ASCE, Kruger National Park, South Africa, p. 40.

Morley, M. S., Kapelan, Z., Savić, D. A. & de Marinis, G. 2006 deEPANET: a distributed hydraulic-solver architecture for accelerating optimization applications working with conditions of uncertainty. In: *7th International Conference on Hydroinformatics*, Nice, France, pp. 2465–2472.

Morley, M. S. & Tricarico, C. 2008 *Pressure Driven Demand Extension for EPANET (EPANETpdd)*. Internal report, University of Exeter, UK. Available on request.

NVIDIA Corporation 2011 *NVIDIA CUDA C Programming Guide Version 4.0*. Available from: http://developer.nvidia.com/cuda.

Nickolls, J. & Dally, W. J. 2010 The GPU Computing Era. *IEEE Micro*, **30**, 56–69.

Open Source Initiative OSI 2012 MIT License. Available from: http://www.opensource.org/licenses/mit-license.php.

Ostfeld, A., Uber, J. G., Salomons, E., Berry, J. W., Hart, W. E., Phillips, C. A., Watson, J.-P., Dorini, G., Jonkergouw, P., Kapelan, Z., di Pierro, F., Khu, S.-T., Savic, D. A., Eliades, D., Polycarpou, M., Ghimire, S. R., Barkdoll, B. D., Gueli, R., Huang, J. J., McBean, E. A., James, W., Krause, A., Leskovec, J., Isovitsch, S., Xu, J., Guestrin, C., VanBriesen, J., Small, M., Fischbeck, P., Preis, A., Propato, M., Piller, O., Trachtman, G. B., Wu, Z. Y. & Walski, T. 2008 The Battle of the Water Sensor Networks (BWSN): a design challenge for engineers and algorithms. *Journal of Water Resources Planning and Management* **134**, 556–568.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. & Phillips, J. C. 2008 GPU Computing. *Proceedings of the IEEE* **96**, 879–899.

Raman, S. K., Pentkovski, V. & Keshava, J. 2000 Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro* **20**, 47–57.

Rossman, L. A. 1999 *Computer models/EPANET. Water Distribution Systems Handbook*. McGraw Hill, New York.

Rossman, L. A. 2000 *EPANET 2: Users Manual*. United States Environmental Protection Agency, Cincinnati, USA.

Rossman, L. A. 2007 Discussion of 'Solution for Water Distribution Systems under Pressure-Deficient Conditions' by Wah Khim Ang and Paul W. Jowitt. *Journal of Water Resources Planning and Management* **133**, 566–567.

Saad, Y. 1990 SPARSKIT: A basic toolkit for sparse matrix computations, Citeseer.

Todini, E. & Pilati, S. 1988 A gradient algorithm for the analysis of pipe networks. In: *Computer Applications in Water Supply, Vol. 1 – Systems Analysis and Simulation*, John Wiley & Sons, New York, pp. 1–20.

Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M. & Shringarpure, A. 2010 On the limits of GPU acceleration. In: *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pp. 13–19.

Wu, Z. Y. & Lee, I. 2011 Lesson for parallelizing linear equation solvers and water distribution analysis. In: *Urban Water Management: Challenges and Opportunities. Computing and Control for the Water Industry 2011 (CCWI 2011)*, 5–7 September 2011, Exeter, Vol. 1, 21–26.

van Zyl, J. E., Savic, D. A. & Walters, G. A. 2004 Operational Optimization of Water Distribution Systems Using a Hybrid Genetic Algorithm. *Journal of Water Resources Planning and Management* **130**, 160–170.

First received 21 December 2011; accepted in revised form 28 June 2012. Available online 24 September 2012