Computers & Geosciences 87 (2016) 91–100

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Case Study

A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications

Antonio J. Rueda *, José M. Noguera, Adrián Luque

Departamento de Informática, Escuela Politécnica Superior, Universidad of Jaén, Paraje Las Lagunillas s/n, 23071 Jaén, Spain

ARTICLE INFO

Article history:
Received 1 August 2015
Received in revised form
20 November 2015
Accepted 3 December 2015
Available online 8 December 2015

Keywords:
GPU computing
OpenACC
CUDA
Drainage network extraction

ABSTRACT

In recent years GPU computing has gained wide acceptance as a simple low-cost solution for speeding up computationally expensive processing in many scientific and engineering applications. However, in most cases accelerating a traditional CPU implementation for a GPU is a non-trivial task that requires a thorough refactorization of the code and specific optimizations that depend on the architecture of the device. OpenACC is a promising technology that aims at reducing the effort required to accelerate C/C++/Fortran code on an attached multicore device. Virtually with this technology the CPU code only has to be augmented with a few compiler directives to identify the areas to be accelerated and the way in which data has to be moved between the CPU and GPU. Its potential benefits are multiple: better code readability, less development time, lower risk of errors and less dependency on the underlying architecture and future evolution of the GPU technology. Our aim with this work is to evaluate the pros and cons of using OpenACC against native GPU implementations in computationally expensive hydrological applications, using the classic D8 algorithm of O'Callaghan and Mark for river network extraction as case-study. We implemented the flow accumulation step of this algorithm in CPU, using OpenACC and two different CUDA versions, comparing the length and complexity of the code and its performance with different datasets. We advance that although OpenACC can not match the performance of a CUDA optimized implementation ($\times$3.5 slower in average), it provides a significant performance improvement against a CPU implementation ($\times$2–6) with by far a simpler code and less implementation effort.

1. Introduction

The efficient management of the vast volume of data generated by earth observation sensors, and the execution of complex numerical simulations that allows us to understand and predict the processes within the Earth are major challenges for the geoscience disciplines. Since the early 80s, high-performance computing techniques have been extensively used by geoscientist to address these important tasks. Concepts like distributed computing, clustering or parallel programming, and APIs like MPI or OpenMP have become popular among the community.

More recently GPU (Graphics Processing Unit) computing (Owens et al., 2008) has also been introduced as a simple low-cost solution for speeding up computationally expensive processing with successful applications in fields such as Hydrology, Atmospheric Physics or Seismology. GPU computing and traditional parallel distributed systems can be complementary and work together: coarse-grained tasks can be assigned to each computer, where they are decomposed into fine-grained subtasks that are executed in one or several GPUs.

Although CUDA and OpenCL technologies have greatly simplified GPU computing since the old GPGPU era (Owens et al., 2007), implementing and tuning an algorithm for the GPU remains being a non-trivial task with sometimes a good dose of trial and error. Errors are common and occur even in applications that have been extensively tested or reviewed by third-party experts (see Mielikainen et al., 2014 for an illustrative example). But probably the main problem is that a particular implementation can become outdated very quickly when a new generation of GPUs comes out. For instance the release of CUDA in 2007 was a revolution that led hundreds of existing GPGPU

* Corresponding author.
E-mail addresses: ajrueda@ujaen.es (A.J. Rueda), jnoguera@ujaen.es (J.M. Noguera), alluque@ujaen.es (A. Luque).

http://dx.doi.org/10.1016/j.cageo.2015.12.004
0098-3004/

applications and libraries based on shader programming to obsolescence. In a smaller scale, each new generation of CUDA-enabled GPUs brings new features that can simplify or provide more efficient solutions than those previously existing. For instance, the support of atomic operations, a basic feature to guarantee a safe access to memory from several concurrent threads, was not available until the second generation of CUDA (compute capability 1.1) was released. Before this many algorithms could only be implemented in an inefficient or cumbersome way.

For all the above ideas an ideal GPU programming interface for developing applications in Hydrology and other geosciences should have the following two basic characteristics:

1. Should be minimally intrusive: programming for the GPU should be close to programming to the CPU. This would reduce the implementation effort and accelerate the entire development cycle: implementing different GPU-accelerated approaches, testing, comparison against CPU versions, etc. Use of specific GPU-oriented languages (e.g. Cg and GLSL) is discouraged.
2. Should be a high-level interface providing an abstraction of the underlying GPU hardware. Ideally it should also take advantage of the new features of upcoming GPU generations in a transparent way. Focus should be on task decomposition and parallelism issues, not on subtle technical details.

OpenACC (OpenACC.org, 2013) is an API for GPU computing that appears to meet the previous requirements. It defines a set of compiler directives that allows the execution of certain parts of a C/C++/Fortran program on a GPU in a straightforward way. OpenACC has a high-level interface that automatically handles tasks such as GPU initialization and shutdown, data transfer between the host and the GPU or thread execution. Despite being a relatively recent specification, there are several commercial and free implementations already available.

Our aim with this work is to contrast the benefits and drawbacks of using OpenACC versus native GPU programming with CUDA in hydrological applications. Although development using OpenACC should be apparently simpler, faster and less error-prone than using CUDA, as any high-level API it incurs a performance loss. Our case-study for this comparison is the classic algorithm of O'Callaghan and Mark (1984) for the extraction of drainage networks from digital elevation models that has already been partially (Ortega and Rueda, 2010) or fully implemented in GPU (Eränen et al., 2014) using CUDA. We conclude that the benefits of using OpenACC in this kind of applications offset the moderate loss of performance.

2. Background

2.1. GPU computing technologies

NVIDIA released CUDA in 2007 as a new technology that enabled programming GPUs for general purpose problems without any knowledge of the rendering pipeline (NVIDIA Corporation, 2015). Therefore, in CUDA the GPU is not seen as a 3D graphics processor but as a shared-memory parallel multiprocessor. Each multiprocessor is, in turn, composed of several SIMD (Single Instruction Multiple Data) streaming processors. A program has to define the thread code that is going to be executed by the processors and launch a computation with a given number of these threads, depending on the nature of the problem to solve. CUDA groups the threads into warps that are executed in parallel in a multiprocessor. During the computation the threads can access the device memory for the input data and also for storing the results.

OpenCL (Stone et al., 2010; Khronos OpenCL Working Group, 2014) aims at establishing a portable vendor-independent platform for implementing algorithms in any heterogeneous multi-core system, including CPUs, GPUs, and FPGAs. A key advantage of OpenCL is its abstract memory and execution model that avoids the need to know or use any device specific low-level feature. But like CUDA, translating a CPU implementation of an algorithm to OpenCL requires a thorough rewrite, including coding the threads in the OpenCL C language. The performance compared to a native API like CUDA is also lower.

OpenACC (OpenACC.org, 2013) is an API developed by Cray, CAPS, Nvidia and PGI targeted at programming CPU/GPU systems. OpenACC uses a simple method based on compiler directives to specify which parts of the C/C++/Fortran code should be GPU-accelerated. Therefore, compared to CUDA or OpenCL, the effort required to translate serial to parallel CPU/GPU code is much smaller, and in most cases original code statements does not need to be modified. Moreover a single code can be used for serial/parallel implementations, as ordinary C++ compilers ignore OpenACC pragma directives. OpenACC is inspired in a OpenMP, a previous programming standard for parallel shared-memory multiprocessing (OpenMP Architecture Review Board, 2013). Although OpenMP was designed for parallel programming in systems with multicore CPUs, beginning with version 4.0 it has started to support GPU accelerators. Therefore, it is likely that both standards will converge into a single/combined specification in the near future.

The OpenACC standard is currently supported by three commercial compilers from PGI, CAPS and Cray. These three companies, along with nVidia, were the initial developers of the standard. The support from the open source community has recently begun with the accULL compiler (Reyes et al., 2012), OpenUH (Tian et al., 2014), and OpenARC (Lee and Vetter, 2014). The 5.x.x series of the GNU GCC compiler has also started to support OpenACC as an experimental feature. Unfortunately these open source implementations are research compilers and/or are still at an early development stage, therefore only the commercial alternatives can be considered production ready.

2.2. GPU computing in geoscience applications

Parallel and distributed computing have been extensively used in the geosciences due to the need to manipulate large-scale data and the complexity of the numerical models used in the analysis, simulation and prediction of the Earth processes. However, GPU computing has been introduced only recently, when modern general purpose GPU programming frameworks have been made available. Prior to this,

some related work as implementing lattice-Boltzmann methods (Li et al., 2003) or solving the Navier–Stokes equations for incompressible fluid flow (Krüger and Westermann, 2003; Harris et al., 2003; Goodnight et al., 2003; Bolz et al., 2003) was done using GPGPU based on shader programming. However, the limitations and complexities of this programming model prevented the widespread adoption of GPU computing for geoscience applications. The survey of Owens et al. (2007) provides an excellent overview of the first GPGPU era.

Michalakes and Vachharajani (2008) were probably the first to show the possibilities of modern GPUs in geoscience applications, using CUDA for accelerating weather numerical algorithms. Shortly later Walsh et al. (2009) used the same technology for simulating geofluid dynamics and seismic wave propagation. Since then many more applications have been reported, as 3D visualization of geomorphologic processes (Clark et al., 2012; Mateo-Lázaro et al., 2014), ice-flow modeling (Braedstrup et al., 2014), calculation of surface roughness parameters (Li et al., 2015), solar irradiance estimation (Lukač and Žalik, 2013), and further work on seismic wave modeling (Abdelkhalek et al., 2009; Okamoto et al., 2010; Komatitsch et al., 2010) and acceleration of weather numerical algorithms (Zhang and Garcia, 2012; Mielikainen et al., 2012).

For the particular problem of drainage network extraction from digital elevation models (DEMs), Ortega and Rueda (2010) were the first to propose a CPU/GPU implementation of the classic algorithm of O'Callaghan and Mark (1984). In this work they used CUDA to accelerate the flow accumulation phase, which is by far the most computationally expensive. Later, Qin and Zhan (2012) accelerated the DEM preprocessing (removing depressions and flat areas) and flow accumulation considering multiple instead of single flow directions. Finally Eränen et al. (2014) successfully implemented in the GPU all the steps of the drainage network extraction algorithm for single flow directions considering also uncertainty in the input elevation model.

2.3. Drainage network extraction

The extraction of the river networks from a DEM can be done using different strategies but the most widely used is based on the work of O'Callaghan and Mark (1984), that simulates the water flow over the terrain to extract drainage information. The basic implementation of this approach has three stages. In the first stage, one or more drainage directions are assigned from each DEM cell to its eight neighbours following the line of steepest slope. In the second stage, a unit of flow is set to each cell, and the DEM is repeatedly scanned row by. During each iteration, the flow of the cells containing a non-zero flow is transferred to their neighbours following the drainage directions computed before. In the final stage a threshold is chosen and all cells with an accumulated transferred flow greater than this threshold are identified as part of the drainage network.

In the first stage the original work of O'Callaghan and Mark (1984) assigned only one flow direction to each cell chosen from 8 possibilities, but since then more sophisticated multiple flow models have been proposed (Freeman, 1991). Ideally the algorithm should always find the corresponding drainage directions for each cell but several factors including measure errors or lack of accuracy in the remote sensors lead to DEMs with pits or flat areas that cannot be processed. This problem is solved by fixing the DEM using different strategies (Rieger, 1998; Jones, 2002). For a general introduction on the vast existing literature on drainage network extraction, including descriptions of the different flow models and pit and flat removing approaches we refer the reader to the excellent survey of Barták (2009).

3. OpenACC

While relatively simple compared with CUDA or OpenCL, a full description of OpenACC is beyond the scope of this paper. Therefore, we only outline here its main features and most important directives, inviting the reader to look further in the OpenACC specification (OpenACC.org, 2013). In the next section, during the discussion of the algorithm implementation we present a few more advanced directives. We focus on C/C++ code although the OpenACC directives for Fortran are basically the same.

OpenACC follows a host-directed execution where some parts of the program are redirected to an accelerator device, usually a GPU, looking for a performance gain. In order to execute the assigned task the accelerator launches several threads grouped into three levels of parallelism granularity: gangs, workers and vectors. Gangs correspond to independent execution units (e.g. different multiprocessors). There is limited support for communication or synchronization between threads from different gangs. A gang has a number of workers that implement fine-grained parallelism with efficient communication mechanisms. Finally a worker can have a number of vectors that correspond with fast SIMD or vector operations. These vectors are roughly equivalent to the threads running in a SIMD processor in a CUDA-enabled GPU. In the memory model of OpenACC the accelerator has its own memory, and data has to be transferred from host to device before starting any computation. In most cases OpenACC transfers data to and from the device transparently.

All OpenACC directives have the form:

```
# pragma acc directive-name [additional clauses]
```

The effect of an OpenACC directive applies to the following statement, structured block or loop. There are two directives to indicate to OpenACC that a portion of code should be executed in the accelerator: `parallel` and `kernels`. Both are essentially equivalent; the main difference is that in a parallel region the programmer explicitly indicates that it must be run in the accelerator while with kernels the compilers decide whether it will be executed in the accelerator and how (e.g. dividing the code into several different executions). We focus here on the parallel directive although what follows is also valid for kernels.

Listing 1 shows a simple C++ code that implements a parallel vector sum accelerated with OpenACC directives. The loop directive instructs the OpenACC compiler to execute the loop iterations in parallel in the GPU. The OpenACC compiler automatically generates the required code to copy the vectors `v1` and `v2` to the device memory and the `res` vector back to the host memory. However, the

programmer can explicitly indicate the way in which data moves to and from the device to avoid any undesired transfer with the resulting performance penalty. This can be done with the `data` directive, specifying the operations to perform with the following clauses:

- `copyin(var)`: Creates a variable in the device memory copying its value from the host.
- `copyout(var)`: Creates a variable in the device memory which value is copied to the host after the computation in the parallel region has finished.
- `copy(var)`: Equivalent to using both previous clauses on a variable.
- `create(var)`: Creates a temporal variable in the device memory which is not required in the host.
- `present(var)`: Indicates that a variable is already on the device memory after being created in a previous parallel region.

Listing 1. Sample OpenACC-accelerated code.

```
void vectorSum(float *v1, float *v2, float *res, int size)
{
  #pragma acc data copyin(v1[0:size], v2[0:size])
  #pragma acc data copyout(res[0:size])
  #pragma acc parallel
  #pragma loop
  for (int i = 0; i < size; ++i) {
    res[i] = v1[i] + v2[i];
  }
}
```

It is possible to transfer only part of an array indicating the start and the size of the interval using brackets. When working with dynamically allocated arrays, this notation must also be used to tell the OpenACC compiler the dimensions of the array, as it cannot be inferred during the code compilation. This is shown in Listing 1: the dimensions of the arrays pointed by `v1`, `v2` and `res` are given; the two first are copied to the device and the former is only copied back to the host at the end of the loop.

Finally it is very common to combine several directives into one. For instance parallel and loop directives are usually combined when they annotate the same fragment of code:

```
# pragma acc parallel loop
```

4. Flow accumulation implementation

As we explained in Sections 2.2 and 2.3, the flow accumulation (or flow routing) is the most time consuming step of the drainage delineation process, particularly if a multiple direction flow model is chosen. Therefore, most efforts to accelerate drainage network identification have focused on here, using traditional parallel hardware (Mower, 1994) or modern GPUs (Ortega and Rueda, 2010; Qin and Zhan, 2012). Flow accumulation exhibits a fine-grained parallelism that can be accelerated by GPUs in a straightforward way. Unfortunately, as the flow is being accumulated in certain cells and is drained away from the DEM, more and more cells get empty and each subsequent iteration requires less processing. Anyway, GPU implementations usually run 6–8 times faster than a single core CPU implementation using a single direction flow and 10 times faster for a multiple direction flow model (Qin and Zhan, 2012).

4.1. CPU implementation

Listing 2 shows a simple non-optimized single core CPU implementation of the flow accumulation (single flow direction model). The OpenACC pragma directives should be ignored by now; we describe their meaning in the next section. The function `flowRoutingStep` implements a single flow transfer step on the entire DEM. The main function `flowRouting` iteratively calls `flowRoutingStep` until there is

Table 1
Execution times (in seconds) of the single-core CPU, OpenACC and the naive and optimized CUDA versions for several datasets.

Dataset	Size	CPU	OACC	CUDA (naive)	CUDA (opt.)
N36W004	1201^2	1.55	0.79	0.42	0.23
N36W005	1201^2	1.78	0.82	0.50	0.25
N37W004	1201^2	5.14	1.28	0.93	0.46
N37W005	1201^2	6.97	1.52	1.31	0.64
Granada	2401^2	38.66	6.36	5.68	1.29
Africa	4801^2	233.69	44.99	46.07	10.55

no flow transfer on the DEM. The inputs are the `accumFlow` array, that stores the number of flow units that have been drained through each cell and the `lowestNeighbour` array, which value for a single cell represents the position of the lowest of its eight adjacent neighbours (i.e. the neighbour that drains the flow of the cell). A special position `OUTSIDE_DEM` is used when the cell is at the border of the DEM and all their neighbours have a higher elevation, that is, we can assume that the cell flow is drained outside the DEM. A third array `flow` is created in `flowRouting` and initialized with a unit of flow per cell. In each iteration in the `flowRoutingStep` function, the current flow of each cell is examined; if this is greater than 0 it is transferred to its lowest neighbour, also updating its accumulated flow.

Listing 2. CPU/OpenACC implementation of flow accumulation.

```
int flowRoutingStep(int flow[], int accumFlow[], int lowestNeighbour[],
  int dimX, int dimY)
{
  int dim = dimX * dimY, transfer = 0;

  #pragma acc data present(flow[0:dim], accumFlow[0:dim])
  #pragma acc data present(lowestNeighbour[0:dim])
  #pragma acc parallel loop reduction(+:transfer)
  for (int i = 0; i < dim; ++i) {
    int cellFlow = flow[i];

    // Proceed only if there is flow in the cell
    if (cellFlow > 0.0f) {
      #pragma acc atomic
      flow[i] -= cellFlow;
      transfer += cellFlow;

      int lowest = lowestNeighbour[i];

      // Skip draining to a neighbour if the cell drains
      // outside the DEM or the cell is at sea
      if (lowest != OUTSIDE_DEM) {
        #pragma acc atomic
        flow[lowest] += cellFlow;
        #pragma acc atomic
        accumFlow[lowest] += cellFlow;
      }
    }
  }
  return transfer;
}

void flowRouting(int accumFlow[], int lowestNeighbour[],
  int dimX, int dimY)
{
  int transfer, dim = dimX * dimY;
  int *flow = new int[dim];

  #pragma acc data create(flow[0:dim])
  #pragma acc data copyin(lowestNeighbour[0:dim])
  #pragma acc data copyout(accumFlow[0:dim])
  {
    // Fill cells with 1 flow unit and set accumulated flow to 0
    #pragma acc parallel loop
    for (int i = 0; i < dim; ++i) {
      flow[i] = 1;
      accumFlow[i] = 0;
    }
    // Do a routing step until there is no flow transfer in the previous iteration
    do {
      transfer = flowRoutingStep(
        flow, accumFlow, lowestNeighbour, dimX, dimY);
    } while (transfer > 0);
  }
  delete[] flow;
}
```

4.2. OpenACC implementation

The `pragma` directives in Listing 2 illustrate the simple changes needed to enable GPU computing for accelerating the implementation.

The data directives in `flowRouting` instruct OpenACC how to proceed with the data arrays. The flow array is directly created in the GPU memory and filled with one flow unit per cell. The `accumFlow` array is also created in the GPU and filled with zero values but it is copied to the host memory when the computation finishes. Finally `lowestNeighbour` array is a read-only array that is required during the

computation, and therefore is annotated with the `copyin` clause. All these data directives are optional: if not given, the arrays are copied to the GPU memory before starting the computation and back to the host at the end. However, they make the implementation faster by avoiding multiple or undesired data movements.

The `flowRoutingStep` function starts with another data directive indicating OpenACC that all the needed data are present in the GPU memory, and therefore everything is ready for launching the kernels. This is done with the next parallel loop directive. The OpenACC compiler runs this loop in parallel launching several gangs of vector computations. The number of gangs and vectors is decided by the compiler after presumably analyzing the code and considering the number of processing units available in the GPU. However, it can also be given by the programmer using the clauses gang(⟨num⟩) and vector(⟨num⟩) to fine-tune the implementation for a particular platform (at the cost of making it less portable). For instance `gang(64)` and `vector(192)` gives the best performance for our testing GPU. Arrays `flow` and `accumFlow` are now updated in parallel by several threads, and therefore these accesses have to be atomic to avoid a corrupted result after a simultaneous flow update in the same cell from two or more threads. Annotating the three writing accesses with atomic solves the problem. The transfer variable seems to be a similar case but as it accumulates a global result from all the threads it can be solved much more efficiently by using a parallel reduction operation (Martin et al., 2012). The reduction directive can be used for this purpose, that takes as arguments the operation to perform during the reduction (+) and the variable to get the result.

Apart from adding an extra block for the data directives, the original C++ code does not need any modification to be GPU-accelerated with OpenACC. The only required directive is `parallel loop` before the for-loop in `flowRoutingStep`. The rest, particularly the data directives, are advisable for improving the performance.

4.3. CUDA implementation

Listings 3 and 4 show a naive CUDA reimplementation of the code in Listing 2. Even a simple example like this requires a thorough refactorization. Most of the code in the counterpart of the `flowRouting` function in Listing 4 is now dedicated to creating and copying the data arrays to the GPU memory. For the sake of simplicity we omitted creating and initializing the flow array in the GPU as it would require defining and calling a new kernel. A key aspect in a CUDA is setting the number of blocks and threads per block that are launched in the computation, which has a direct impact on its performance. In a CUDA-enabled system the OpenACC compiler usually maps gangs and vectors in blocks and threads per block respectively.

Listing 3. Kernel of the naive CUDA implementation of flow accumulation.

```
__global__ void flowRoutingStepKernel(int *flow, int *accumFlow,
  int *lowestNeighbour, int dimX, int dimY, int *transfer)
{
  // Compute the cell to process using the block and thread id
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < dimX * dimY && flow[i] > 0) {
    int cellFlow = flow[i];

    atomicAdd(&flow[i], -cellFlow);
    atomicAdd(transfer, cellFlow);

    int lowest = lowestNeighbour[i];

    if (lowest != OUTSIDE_DEM) {
      atomicAdd(&flow[lowest], cellFlow);
      atomicAdd(&accumFlow[lowest], cellFlow);
    }
  }
}
```

Table 2
Speedups in execution times of the OpenACC and CUDA (optimized) versions compared to the single-core CPU version, and CUDA compared to OpenACC.

Dataset	Size	OACC vs CPU	CUDA vs CPU	CUDA vs OACC
N36W004	1201^2	× 1.97	× 6.87	× 3.50
N36W005	1201^2	× 2.18	× 7.21	× 3.33
N37W004	1201^2	× 4.01	× 11.25	× 2.81
N37W005	1201^2	× 4.59	× 10.88	× 2.37
Granada	2401^2	× 6.08	× 29.88	× 4.92
Africa	4801^2	× 5.19	× 22.15	× 4.26

Listing 4. Naive CUDA implementation of flow accumulation.

```
void flowRoutingCUDA(int accumFlow[], int lowestNeighbour[],
  int dimX, int dimY)
{
  int transfer, dim = dimX * dimY, arraySize = sizeof(int) * dim;

  // Define number of blocks and threads per block
  dim3 threadsPerBlock(512);
  dim3 blocks(ceil((float) dim / threadsPerBlock.x));

  // Init flow and accumulated flow arrays
  int *flow = new int[dim];

  for (int i = 0; i < dim; ++i) {
    flow[i] = 1;
    accumFlow[i] = 0;
  }

  // Create arrays in GPU memory and transfer data from main memory
  int *flowGPU, *accumFlowGPU, *lowestNeighbourGPU, *transferGPU

  cudaMalloc(&flowGPU, arraySize);
  cudaMemcpy(flowGPU, flow, arraySize, cudaMemcpyHostToDevice);
  delete[] flow;

  cudaMalloc(&flowGPU, arraySize);
  cudaMemcpy(accumFlowGPU, accumFlow, arraySize, cudaMemcpyHostToDevice);

  cudaMalloc(&lowestNeighbourGPU, arraySize);
  cudaMemcpy(lowestNeighbourGPU, lowestNeighbour,
    arraySize, cudaMemcpyHostToDevice);

  cudaMalloc(&transferGPU, sizeof(int));

  // Do a routing step until there is no flow transfer in the previous iteration
  do {
    // Reset accumulated flow for the next iteration
    cudaMemset(transferGPU, 0, sizeof(int));

    // Launch computation
    flowRoutingStepKernel<<<blocks, threadsPerBlock>>>(
      flowGPU, accumFlowGPU, lowestNeighbourGPU,
      dimX, dimY, transferGPU);

      // Get accumulated flow in the last iteration
      cudaMemcpy(transfer, transferGPU, sizeof(int), cudaMemcpyDeviceToHost);
    } while (transfer > 0);

  // Transfer results to accumFlow array in main memory
  cudaMemcpy(accumFlow, accumFlowGPU, arraySize, cudaMemcpyDeviceToHost);

  // Deallocate GPU memory
  cudaFree(transferGPU);
  cudaFree(lowestNeighbourGPU);
  cudaFree(accumFlowGPU);
  cudaFree(flowGPU);
}
```

The `flowRoutingStepKernel` shown in Listing 3 is the kernel that is executed in parallel by the CUDA processors. It is quite similar to the code of the `flowRoutingStep` function, with a couple of main differences. First, there is no loop as each thread running a kernel processes a different cell. The index of the cell to process is computed in the usual way in CUDA, using the block size, block identifier and thread local identifier. And second, the global flow transfer is implemented as an array of size 1 in the GPU memory, and is not computed through a parallel reduction but using atomics sums.

In our previous work (Ortega and Rueda, 2010) we showed how in order to have a significant performance gain against a CPU implementation each thread has to compute the flow accumulation for several cells (e.g. 64 cells per thread). We have implemented an additional optimized version that works in this way by adding a loop in the kernel to process a different set of cells in each thread. The number of blocks defined in `flowRoutingCUDA` also needs to be recomputed, as less threads are required to process the DEM cells.

Our optimized CUDA version, apart from implementing the one-thread/multiple-cells scheme described above, also avoids the use of atomic operations. To achieve this goal, it uses two sets of the `flow` and `accumFlow` buffers, and uses alternatively one set for reading and another for writing. The details can be found in Ortega and Rueda (2010). Another improvement is the use of an array of per-block booleans encoding if all the cells assigned to the threads of each block are empty. A property verified during the flow accumulation is that if a cell does not receive flow from its neighbours, and therefore remains with zero flow during one iteration, then it has not to be reconsidered in the next iterations (O'Callaghan and Mark, 1984). Consequently, if a true value is found, all the threads of the block end

immediately as no further processing is required. Conversely if the value is false, it is set to true and the threads of the block proceed as usual. If any of the cells processed by the thread have a flow different from zero, the boolean is updated to false to ensure that the cell is processed again in the next iteration.

There is more room for optimization in CUDA: a careful choice of the number of blocks and threads per block to launch in the computation, a coalesced access to data in global memory and an effective use of shared memory in the SIMD processors can speed up a CPU sequential version several orders of magnitude. However this comes with the cost of a considerable refactorization of the original implementation and some trial and error with sometimes unexpected results. Simpler optimizations can also be implemented in OpenACC although our aim with this paper has always been to keep it as close to the original CPU version as possible.

Another advantage of CUDA, due to being a widely adopted technology, is the number of available high-quality APIs that can significantly accelerate the development of certain common tasks in many applications (e.g. cuBLAS for linear algebra or cuFFT for fast Fourier transform computation). It is reasonable to think that as OpenACC gains acceptance, more libraries will also be available to help developers.

4.4. Performance evaluation

The proposed implementations were tested on several real life datasets to evaluate their efficiency depending on the DEM size. Our experimentation was carried out on a PC equipped with a dual Intel Xeon E5620 processor running at 2.4 GHz with 12 GB of RAM, an NVIDIA Tesla C2050 and Windows 7 64 bits. The algorithms were directly implemented in C++ and compiled with Visual Studio 2012. We used the PGI x64+Accelerator Compiler v. 14.7 for the OpenACC implementations and the NVIDIA CUDA Toolkit 6.

The source of the DEMs used in this study was the widespread 3 arcsec resolution (90 m) SRTM3 DEM v2.1 provided by the NASA Jet Propulsion Laboratory. We selected several regions with distinct terrain features and resolution size (ranging from 1021^2 to 4801^2 cells). These DEMs were preprocessed as described by Jenson and Domingue(1988) to fill the depressions that could hinder flow routing.

5. Results and discussion

Table 1 reports the time in seconds required to calculate the drainage network for the different datasets using the proposed implementations. From left to right, Table 1 provides the name and dimensions of the datasets and the total running time required by the single-core CPU implementation (baseline time), the OpenACC version (described in Section 4.2), the naive CUDA version and the optimized CUDA version (described in Section 4.3).

Table 2, on the other hand, shows the speedup achieved by our two GPU implementations over the single-core CPU version, and compares the OpenACC implementation with the optimized CUDA alternative.

Fig. 1 depicts the datasets used in our experimentation, as well as the drainage networks extracted by the evaluated implementations. Note that all versions obtained the same networks, as they were based on the same algorithm and were initialized with the same parameters (only the implementation varied).

When comparing the performance of the CPU and the OpenACC implementations (see Tables 1 and 2) we observe that the performance gain of OpenACC is somewhat limited (around $\times 2$) for the N36W004 and N36W005 datasets. This stems from the fact that these datasets were almost covered by sea, as shown in Figs. 1a and b. Therefore, the performance gain on processing a relatively small number of cells in parallel barely compensate for the cost of the data movements between CPU and GPU. However, as we increased the DEM size, the OpenACC implementation quickly leaves the CPU version behind. For example, OpenACC outperformed the CPU by a factor of around $\times 5$–6 for the two largest datasets evaluated. We consider that this performance improvement is very satisfactory considering the small investment in coding time we had to put on the OpenACC implementation.

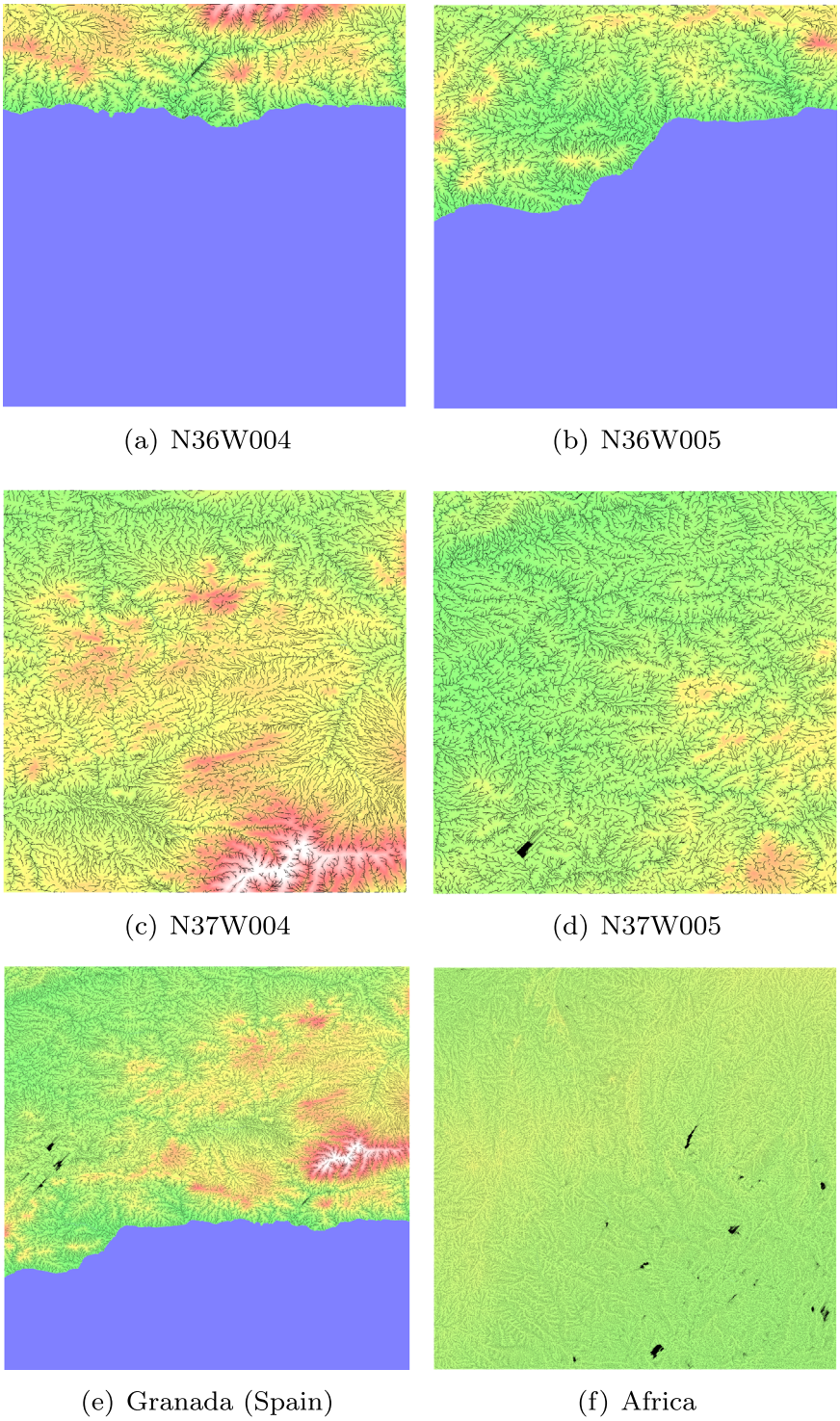
Regarding the CUDA implementations, our experiments showed that the performance of the naive version was roughly similar to the OpenACC alternative. Unsurprisingly, a simple porting from CPU to CUDA does not fully exploit the computing power of the GPU. A real performance gain of CUDA over OpenACC can only be achieved with more involved implementations including specific GPU optimizations (Ortega and Rueda, 2010) as we showed in Section 4.3. Conversely, this also suggests that the OpenACC compiler did a good job at parallelizing and optimizing the CPU code on its own.

Nevertheless, the optimized CUDA implementation managed to outperform the OpenACC version with a factor ranging from $\times 2.4$ to $\times 4.9$ for all datasets. Therefore, we can affirm that CUDA is still the fastest alternative if the programmer has the technical skills and is willing to invest a certain amount of time on refactoring, parallelizing and tuning the algorithm.

If we compare the three implementations in terms of simplicity of code we observe that the OpenACC implementation has 42 lines of code, whereas the CUDA versions required 49 and 72 lines respectively, that is, 16% and 72% more. But it is even more important to highlight that the OpenACC version reuses 100% of the code of the CPU version adding a 40% extra for the directives. In contrast both CUDA versions require an implementation from scratch.

At the light of these results, when choosing between these two technologies we have to consider how critical the performance is, how much time we are willing to invest into the implementation, and the availability of existing GPU-accelerated libraries that can accelerate and simplify our development. Nevertheless, it is also possible that both approaches benefit from each other. For example, OpenACC could be helpful as a general solution when the developing times are important and there is only non-parallel code available. CUDA, on the other hand, could be used to fine-tune only the most specifically computer-intensive kernels.

We have chosen the D8 algorithm for this study because it can be parallelized in an illustrative, straightforward manner. However, we should keep in mind that there exists cases where an effective parallelization requires a complete redesign with a different structure than the sequential solution, e.g., when the response of a node depends on the response of neighbouring nodes in a branching network. Under these circumstances, OpenACC would lose some of its advantage, as a reimplementation of the algorithm is unavoidable. Still, if an OpenACC version is possible, it would probably be simpler and more legible than any CUDA equivalent version.



(a) N36W004 (b) N36W005

(c) N37W004 (d) N37W005

(e) Granada (Spain) (f) Africa

Fig. 1. Datasets used in our experiments.

6. Concluding remarks

This paper aims at geoscientists (particularly in the field of Hydrology) who are interested in developing computationally expensive applications but lack the time or technical knowledge for manually parallelizing the algorithms with CUDA or any other specific GPU-oriented technology. In this context, we have presented OpenACC as a valid and high level API that provides an easy access to the GPU power by delegating most of the subtle technical details on the compiler.

As a case study we have chosen the classic D8 algorithm for extracting drainage networks proposed by O'Callaghan and Mark (1984). Through our paper we have described the baseline CPU-version of this algorithm, and explained how it can be parallelized with OpenACC and CUDA. Both GPU-based implementations have been fully described and compared in terms of complexity and coding effort. Finally,

their performance has been evaluated with some real-world datasets.

Our paper has showed that only half a dozen of pragma directives were required to parallelize an existing C/C++ implementation of the D8 algorithm in a straightforward manner. This suggests that OpenACC is particularly interesting for accelerating legacy Fortran or C/C++ code or as a way to quickly have an accelerated implementation of an algorithm under development. Unfortunately, our comparisons have also proved that this flexibility comes with a performance cost. That is, the low-level programming model of CUDA allows the programmer to manually tune the implementation in many ways, which results in a superior performance to OpenACC. Nonetheless, it is expected that the natural evolution of the OpenACC compilers would reduce the performance gap between both technologies during the next years.

As future work we plan to do a comparison between different OpenACC implementations when more mature commercial and open source compilers will be available, using this or a similar case study. The OpenACC standard directives and each particular implementation have clauses and options for tuning and optimizing the GPU code generated; the question of how close we can get to a CUDA implementation could be answered in a future study. In addition, throughout this work we have used small to moderate sized DEMs, therefore we have planned to extend our evaluation to larger or even offline datasets. Finally we believe that our results can be extrapolated to other geosciences, but this assumption has to be confirmed with new case studies in other fields besides Hydrology.

Acknowledgements

This work has been partially funded by the Ministerio De Economía y Competitividad of Spain under the I+D+i research program TIN2014-58218-R.

References

Abdelkhalek, R., Calandra, H., Coulaud, O., Roman, J., Latu, G., 2009. Fast seismic modeling and reverse time migration on a GPU cluster. In: Smari, W.W., McIntire, J.P. (Eds.), HPCS. IEEE, Leipzig, Germany, pp. 36–43.

Barták, V., 2009. How to extract river networks and catchment boundaries from DEM: a review of digital terrain analysis techniques. J. Landsc. Stud. 2, 57–68.

Bolz, J., Farmer, I., Grinspun, E., Schröoder, P., 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. 22, 917–924.

Braedstrup, C.F., Damsgaard, A., Egholm, D.L., 2014. Ice-sheet modelling accelerated by graphics cards. Comput. Geosci. 72, 210–220.

Clark, S.R., Skogseid, J., Stensby, V., Smethurst, M.A., Tarrou, C., Bruaset, A.M., Thurmond, A.K., 2012. 4Dplates: on the fly visualization of multilayer geoscientific datasets in a plate tectonic environment. Comput. Geosci. 45, 46–51.

Eränen, D., Oksanen, J., Westerholm, J., Sarjakoski, T., 2014. A full graphics processing unit implementation of uncertainty-aware drainage basin delineation. Comput. Geosci. 73, 48–60.

Freeman, T.G., 1991. Calculating catchment area with divergent flow based on a regular grid. Comput. Geosci. 17, 413–422.

Goodnight, N., Woolley, C., Lewin, G., Luebke, D., Humphreys, G., 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 102–111.

Harris, M.J., Baxter, W.V., Scheuermann, T., Lastra, A., 2003. Simulation of cloud dynamics on graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 92–101.

Jenson, S.K., Domingue, J.O., 1988. Extracting topographic structure from digital elevation data for geographic information system analysis. photogrammetric engineering and remote. Photogramm. Eng. Remote Sens. 54, 1593–1600.

Jones, R., 2002. Algorithms for using a DEM for mapping catchment areas of stream sediment samples. Comput. Geosci. 28, 1051–1060.

Khronos OpenCL Working Group, 2014. The OpenCL Specification. Version: 2.0 document revision: 26.

Komatitsch, D., Erlebacher, G., Göddeke, D., Michéa, D., 2010. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. J. Comput. Phys. 229, 7692–7714.

Krüger, J., Westermann, R., 2003. Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graph. 22, 908–916.

Lee, S., Vetter, J.S., 2014. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. ACM, Vancouver, Canada, pp. 115–120.

Li, W., Wei, X., Kaufman, A., 2003. Implementing lattice Boltzmann computation on graphics hardware. Vis. Comput. 19, 444–456.

Li, X., Song, C., López, S., Li, Y., López, J.F., 2015. Fast computation of bare soil surface roughness on a fermi {GPU}. Comput. Geosci. 82, 38–44.

Lukač, N., Žalik, B., 2013. GPU-based roofs' solar potential estimation using LiDAR data. Comput. Geosci. 52, 34–41.

Martin, P., Ayuso, L., Torres, R., Gavilanes, A., 2012. Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In: 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 511–519.

Mateo-Lázaro, J., Sánchez-Navarro, J.A., García-Gil, A., Edo-Romero, V., 2014. 3D-geological structures with digital elevation models using GPU programming. Comput. Geosci. 70, 138–146.

Michalakes, J., Vachharajani, M., 2008. GPU acceleration of numerical weather prediction. in: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp. 1–7.

Mielikainen, J., Huang, B., Huang, H., Goldberg, M., 2012. Improved GPU/CUDA based parallel weather and research forecast (WRF) single moment 5-class (WSM5) cloud microphysics. IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. 5, 1256–1265.

Mielikainen, J., Huang, M., Huang, B., Huang, A.H.L., 2014. Comments on the paper by Huadong Xiao, Jing Sun, Xiaofeng Bian and Zhijun Dai, "GPU acceleration of the WSM6 cloud microphysics scheme in GRAPES model". Comput. Geosci. 72, 262–263.

Mower, J.E., 1994. Data-parallel procedures for drainage basin analysis. Comput. Geosci. 20, 1365–1378.

NVIDIA Corporation, 2015. CUDA C Programming Guide. version 7.0.

O'Callaghan, J.F., Mark, D.M., 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graph. Image Process. 5, 323–344.

Okamoto, T., Takenaka, H., Nakamura, T., Aoki, T., 2010. Accelerating large-scale simulation of seismic wave propagation by multi-GPUs and three-dimensional domain decomposition. Earth Planets Space 62, 939–942.

OpenACC.org, 2013. The OpenACC Application Programming Interface. version 2.0.

OpenMP Architecture Review Board, 2013. OpenMP Application Program Interface. version 4.0.

Ortega, L., Rueda, A.J., 2010. Parallel drainage network computation on CUDA. Comput. Geosci. 36, 171–178.

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J., 2008. GPU computing. Proc. IEEE 96, 879–899.

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

Qin, C.Z., Zhan, L., 2012. Parallelizing flow-accumulation calculations on graphics processing units from iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. Comput. Geosci. 43, 7–16.

Reyes, R., López-Rodríguez, I., Fumero, J.J., de Sande, F., 2012. accULL: an OpenACC implementation with CUDA and OpenCL support. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P. (Eds.), Euro-Par 2012 Parallel Processing. Lecture Notes in Computer Science vol. 7484. Springer, Berlin, Heidelberg, pp. 871–882.

Rieger, W., 1998. A phenomenon-based approach to upslope contributing area and depressions in DEMs. Hydrol. Processes 12, 857–872.

Stone, J.E., Gohara, D., Shi, G., 2010. OpenCL: a parallel programming standard for heterogeneous computing systems. IEEE Des. Test 12, 66–73.

Tian, X., Xu, R., Chapman, B., 2014. OpenUH: Open Source OpenACC Compiler. URL: ⟨http://on-demand.gputechconf.com/gtc/2014/presentations/S4343-openuh-open-source-openacc-compiler.pdf⟩ (retrieved 11/15/2015).

Walsh, S.D., Saar, M.O., Bailey, P., Lilja, D.J., 2009. Accelerating geoscience and engineering system simulations on graphics hardware. Comput. Geosci. 35, 2353–2364.

Zhang, H., Garcia, J., 2012. GPU acceleration of a cloud resolving model using CUDA. In: 12th Annual International Conference on Computational Science, pp. 1030–1038.