CUDA Solutions for the SSSP Problem*

Pedro J. Martín, Roberto Torres, and Antonio Gavilanes

Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain
{pjmartin@sip,r.torres@fdi,agav@sip}.ucm.es

Abstract. We present several algorithms that solve the single-source shortest-path problem using CUDA. We have run them on a database, composed of hundreds of large graphs represented by adjacency lists and adjacency matrices, achieving high speedups regarding a CPU implementation based on Fibonacci heaps. Concerning correctness, we outline why our solutions work, and show that a previous approach [10] is incorrect.

Keywords: Shortest path algorithms, GPU, CUDA.

1 Introduction

Computing shortest paths in a graph is one of the most fundamental problems in computer science and network optimization. In particular, the *Single-Source Shortest-Paths* (in the sequel, SSSP) problem, which computes the weight of the shortest path from a specific vertex (source) to all other vertices, in a weighted directed graph, is a heavily studied problem in graph theory.

Probably, the most well-known algorithm solving this problem for the case of graphs with nonnegative edges was given by Dijkstra in 1959 [1], and nearly all the subsequent proposals are based on it. In spite of its early formulation, this classic solution is still presented in almost every textbook on algorithms [2]. After the simplest Dijkstra´s implementation, which uses arrays to represent min-priority queues and runs in $O(n^2)$ time, where n is the number of vertices, many authors have designed different data structures to implement these queues and achieve better and better asymptotic running times. In particular, Fibonacci heaps [3] can be used to get $O(m + n \log n)$, where m is the number of edges.

As [4] points out, Dijkstra's algorithm is inherently sequential since its efficiency depends on a fixed ordering of the vertices. The Bellman-Ford algorithm allows all vertices to be considered in parallel but at the cost of being not efficient. Different formulations of parallel algorithms for the SSSP problem are reviewed in detail in [5]. In particular, a specific proposal for incorporating parallelism into Dijkstra's algorithm has been the introduction of parallel priority queues [6]. However, the literature contains few experimental studies on parallel algorithms of the nonnegative SSSP problem. Some of the more recent works study the use of supercomputers for solving large graphs. [7] reports performance results on the multithread parallel computer Cray MTA-2, using the Δ-stepping parallel algorithm of [5]. [7] exhibits remarkable parallel speedup when compared to competitive sequential algorithms, for

* Research supported by the Project CCG08-UCM/TIC-4252.

low-diameter sparse graphs of 100 million vertices and 1 billion edges. On the other hand, [8] contains an experimental evaluation of [6] on the APEmille supercomputer, but restricted to graphs with no more than thousands of vertices.

However, some modern applications, such as data mining, network organization, etc. require large graphs with millions of vertices, and some of the previous algorithms become impractical, when we do not have a very expensive hardware at our disposal. Fortunately, Graphics Processing Units (GPUs) supply a high parallel computation power at a low price. Moreover, they have become very popular since the languages involved in their programming have evolved from graphics APIs to general purpose languages. One of the best examples is the CUDA API [9] of NVIDIA. As a consequence of this evolution, the so called General Purpose Computing on GPU (GPGPU) [11] has consolidated as a very active research area, where many problems that are not directly related to computer graphics are solved using GPUs. The aim of all these GPU-based implementations is to achieve better running times than their CPU-based counterparts.

With this new technology available, the natural challenge is: “how can GPUs be used to solve the SSSP problem?”. Unfortunately programming with CUDA must be carefully taken, basically because CUDA programming model is very restricted concerning synchronization, and the unique proposal we are aware of ([10]) is not correct. Apart from giving a counterexample, in this paper we present different correct solutions, based on Dijkstra´s algorithm, that are experimentally compared using a database of hundreds of randomly generated large graphs.

2 Dijkstra´s Algorithm Overview

Dijkstra´s algorithm solves the SSSP problem for directed graphs $G = (V, E)$ in which every edge $(v, v') \in E$ has a positive weight $\omega(v, v') > 0$. Let n and m be the number or vertices and edges respectively. We assume that vertices are numbered from 0 to $n - 1$, and that 0 is the source vertex. The algorithm splits the set of vertices in two parts: the set R of *resolved vertices* (R-vertices) and the set U of *unresolved vertices* (U-vertices), and it keeps a shortest-path estimate $c[i]$ for each vertex i, which actually coincides with the shortest path weight for R-vertices. For U-vertices, $c[i]$ holds the weight of the *shortest special path* (SSP) to i w.r.t. R, that is, the shortest path among the paths to i that exclusively traverses R-vertices before reaching i.

The algorithm implements a loop. Each iteration is composed of three steps: (1) the estimates for U-vertices are relaxed using the last vertex added to R, which we will call the *frontier vertex*, (2) the minimum estimate for U-vertices is computed, and (3) a U-vertex with the minimum estimate is promoted to R, and becomes the new frontier. Figure 1 presents a typical Dijkstra’s algorithm implementation that includes the variable `f` to hold the current frontier vertex. Regardless of the graph representation we chose, it runs in $O(n^2)$.

The soundness of the algorithm is based on two fundamental properties that can be proved. First, anytime a new frontier arises in the third step, its estimate actually coincides to the weight of its shortest path, thus it can be safely promoted to R. Second, in order to relax the estimates of a U-vertex j using the current frontier vertex f, the SSP to j w.r.t. R cannot traverse more R-vertices after visiting f, hence we only consider the previous estimate and $c[f] + \omega(f, j)$ when updating $c[j]$.

```
void Dijkstra (c) {
  forall vertex i { c[i]=INFINITY; u[i]=true;}
  c[0]=0; u[0]=false;
  f=0; mssp=0;
  while (mssp!=INFINITY) {
    forall unresolved vertex j {
         c[j]= min(c[j], c[f]+w[f,j]);}
    mssp= INFINITY;
    forall unresolved vertex j
        if(c[j]<mssp){ mssp=c[j]; f=j;}
    u[f]=false;
  }//while
}
```

Fig. 1. Dijkstra's algorithm implementation

3 Parallelizing Dijkstra's Algorithm

Dijkstra´s algorithm handles a unique frontier vertex even when the estimates of several U-vertices coincide with the minimum computed in the second step. In these cases, the algorithm simply chooses one of them to compose the new frontier. In consequence, it requires a different iteration to promote each of them to R. Fortunately, this set of U-vertices, which we will call F, can be processed at once because the previous two properties remain:

1. Their estimates actually coincide with the weight of their shortest paths.
2. In order to relax the estimate of a remaining U-vertex j, the SSP to j w.r.t. R cannot traverse more R-vertices after visiting one F-vertex, hence only the previous estimate and $\min_{f \in F}\{c[f] + \omega(f, j)\}$ must be considered when updating $c[j]$. In particular, note that only one F-vertex can belong to the SSP to j w.r.t. R.

```
void DA2CF(c) {
  initialize(c, f, u);
  mssp = 0;
  while (mssp != INFINITY) {
    relax(c, f, u);
    mssp = minimum(c, u);
    update(c, f, u, mssp);
  }//while
}
```

```
void initialize(c, f, u) {
  forall vertex i {
    c[i] = INFINITY;
    f[i] = false;
    u[i] = true;
  }//for
  c[0] = 0;
  f[0] = true;  u[0] = false;
}
```

Fig. 2. Dijkstra's algorithm adapted to compound frontiers

Therefore the notion of *compound frontier* can be used to design the *Dijkstra's algorithm Adapted to Compound Frontiers* (DA2CF) presented in Fig. 2. Although the algorithm is composed of the same three basic operations, their implementations must suitably handle compound frontiers:

1. `relax(c, f, u)` must relax the shortest path estimate for every U-vertex using F-vertices. Hence it must compute $c[j] = min\{c[j], c[f] + \omega(f,j)\}$ for every pair of vertices $j \in U$ and $f \in F$.
2. `minimum(c, u)` must find the minimum estimate of the U-vertices, called `mssp`.
3. `update(c, f, u, mssp)` must update the set of U-vertices by removing those vertices whose estimate is equal to `mssp`, which will compose the new set of F-vertices.

There are many ways to implement these operations. Although sequential solutions could be easily written by means of the obvious single loop (two nested loops for the `relax` procedure), the operations can be performed in parallel, by launching a thread for each iteration of the loop (the main loop for `relax`).

Figure 3 shows two versions of the `relax` procedure. On the left, `relax_F` processes F-vertices: "for each F-vertex we visit all of its successors, relaxing c for those vertices that are still unresolved". Observe that the sentence `c[j]=min(c[j], c[i]+ω(i,j))` could produce concurrency inconsistencies if two F-vertices `i` and `i'` accessed the same U-vertex `j` and the worst value `c[i]+ω(i,j)` were finally left. In order to prevent such inconsistencies, we use the atomic instruction `atomicMin(x,y)` that allows only one thread to store the minimum of `x` and `y` in the variable `x`.

CUDA devices of compute capability 1.0 do not support atomic functions, thus we propose another approach that does not use them. Figure 3 on the right presents the `relax_U` procedure which focuses on U-vertices instead of F-vertices: "for each U-vertex, we visit all of its predecessors, relaxing its c-value when a F-vertex is found". Notice that this approach requires predecessors instead of successors.

A parallel version of the `minimum` function is a more difficult task, because of its sequential nature. Fortunately, different reduction procedures have been already adapted to the stream model [12, 13, 14]. In this paper we have adapted the `reduce3` method included in the CUDA SDK 1.1 [15] to obtain the `minimum1` procedure of Fig. 4 on the right.

Finally, we parallelize the `update` procedure as Fig. 4 on the left shows. In the sequel, DA2CF_F and DA2CF_U will denote the algorithms that use `relax_F` and `relax_U`, respectively.

Regarding asymptotic complexity, let us compare the Dijkstra´s algorithm of Fig. 1, that runs in $O(n^2)$, to the sequential versions of the DA2CF algorithm that result

```
void relax_F(c, f, u) {
  forall i in parallel do {
    if (f[i]) {
      forall j successor of i do {
        if (u[j])
          atomicMin(c[j],c[i]+w[i,j]);
      }//for
    }//if
  }//for
}
```

```
void relax_U(c, f, u) {
  forall i in parallel do {
    if (u[i]) {
      forall j predecessor of i do {
        if (f[j])
          c[i]= min(c[i],c[j]+w[j,i]);
      }//for
    }//if
  }//for
}
```

Fig. 3. Processing frontier (left) or unresolved (right) vertices within the `relax` operation

```
void update(c, f, u, mssp) {
  forall i in parallel do {
    f[i] = false;
    if (c[i] == mssp) {
      u[i] = false;
      f[i] = true;
    }//if
  }//for
}
```

```
void minimum1(u, c, minimums) {
  forall i in parallel do {
    thid = threadIdx.x;
    i = blockIdx.x*(2*blockDim.x)+threadIdx.x;
    j = i + blockDim.x;
    data1 = u[i] ? c[i] : INFINITY;
    data2 = u[j] ? c[j] : INFINITY;
    sdata[thid] = min(data1, data2);
    __syncthreads();
    for (s = blockDim.x/2; s>0; s>>=1) {
      if (thid<s) {
        sdata[thid]=min(sdata[thid],sdata[thid+s]);
      }// if
      __syncthreads();
    }// for
    if (thid==0) minimums[blockIdx.x]= sdata[0];
  }// forall
}
```

Fig. 4. Updating the frontier (left), and computing the minimum sssp with CUDA (right)

when the "in parallel" qualifier is erased. Firstly, notice that the number of iterations required for the main DA2CF-loop depends more heavily on the given graph; since the size of the arising compound frontiers influences its termination. Hence, we analyze its worst case. We focus on adjacency lists since they fit better to large graphs and they provide the algorithm of Fig. 1 with smaller running times. The worst case corresponds to a complete graph requiring n iterations (the frontier size is always 1), DA2CF_F also takes a time in $O(n^2)$, but with a greater constant due to the management of the `f` array. However, DA2CF_U takes a time in $O(n^3)$, since the edges arriving at an unresolved vertex are repeatedly processed while it remains unresolved. In order to evaluate the general case, we experimentally run CUDA implementations on randomly generated graphs.

4 CUDA Implementations

The adjacency list representation of a graph is made up of three arrays: v for vertices, e for edges and ω for weights. Array v is used to access the adjacency list of each vertex. Specifically, the adjacency list of the vertex i appears in e and ω from index $v[i]$ to index $v[i+1]-1$ (Fig. 5 on the left). In order to deal with the last vertex in the same way, an extra component is added at the end of v such that $v[n]=m$. In consequence, array v is of size $n+1$ and both e and ω are of size m.

There are two possible interpretations for the data occurring in e. Vertices belonging to the adjacency list of vertex i can be understood as successors or predecessors. Formally, in the predecessor interpretation, there is an edge to i from each adjacent vertex, whereas in the successor interpretation the edge goes from i to each adjacent vertex. Graphs must be represented in the proper interpretation before execution, since the `relax` procedure requires either successors (`relax_F`) or predecessors (`relax_U`), but not both.

4.1 Implementations

We have sequential C implementations corresponding to the sequential versions of DA2CF_F and DA2CF_U, that we respectively call F^{CPU} and U^{CPU}. Before presenting the pure CUDA implementations, we have tried some hybrid systems running on both, CPU and GPU. Since the `minimum` function is inherently sequential, we have restricted this function to run on CPU. Moreover, in order to fit the requirements of any CUDA device, we have focused on the `relax_U` procedure. Hence, we have designed three hybrid implementations based on the DA2CF_U algorithm: U^{H1}, U^{H2} and U^{H3} which respectively run the `update` procedure, the `relax_U` procedure, and both `update` and `relax_U` on the GPU.

In order to run the complete algorithm on GPU, we must run additional passes of the `minimum` function, since the `minimum1` kernel of Fig. 4 only reduces each block to a single value. Thus, we have implemented another kernel, called `minimum2`, to execute a second pass on GPU. The obtained values are finally minimized on CPU in a sequential manner, because the number of these values is too small. Hence, we have two fully GPU-implemented solutions based on the DA2CF_U algorithm, U^{GPU} and $U^{GPU+2min}$ that apply one and two minimization passes, respectively. Based on the DA2CF_F algorithm, we also have two fully-GPU solutions, but this time they have been designed to analyze the cost due to simultaneous accesses to the `c` array. Thus, apart from the F^{GPU} solution, we have another one, called $F^{GPU\_no\_Atomic}$, that does not use the atomic function `atomicMin` but a non-atomic function `min`. We introduce the latter solution only for measuring purposes, since it is not correct in a parallel environment. Anyway, both solutions apply a single `minimum` pass.

4.2 Exploiting CUDA Resources

It is possible to accelerate the U^{GPU} solution by using some CUDA features. Concretely, in this subsection we exploit texture cache and shared memory to improve the implementation of the `relax_U` kernel. Let us call the corresponding solution U^{GPU_PLUS}.

In order to retrieve the boundaries of the adjacency list, the i–th thread must access $v[i]$ and $v[i+1]$, whereas the $(i+1)$–th thread must access $v[i+1]$ and $v[i+2]$. Thus, the value $v[i+1]$ is shared by the two threads, and can be brought only once if shared memory is used. Then, each thread i reads $v[i]$ from global memory, writes it to shared memory, and after that, it reads $v[i+1]$ from shared memory directly. A special case is the last thread of a block, since it will bring both $v[i]$ and $v[i+1]$.

Notice that two threads can access the array `f` for the same vertex `j`. To accelerate the corresponding readings, the array can be accessed through a texture, taking advantage of the texture cache. Thus, it is possible for threads of the same block to read `f[j]` from the cache and not from global memory.

4.3 A Bugged Implementation

Let us explain the problem we have found in the solution presented in [10]. The authors propose an implementation of Dijkstra´s algorithm which relaxes using the frontier, in a similar way to our `relax_F` procedure. However, instead of the atomic

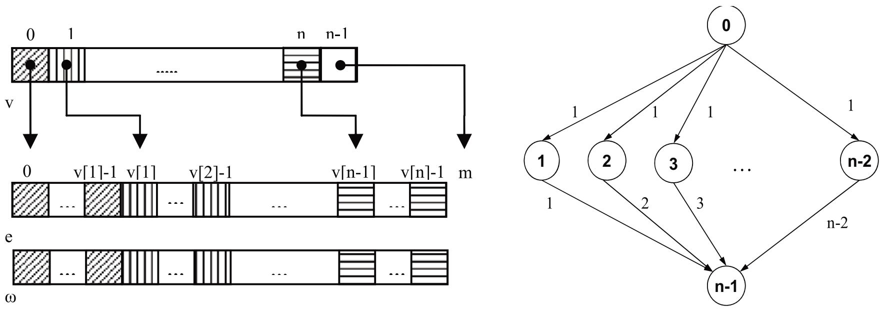

Fig. 5. Left: The adjacency list representation. Right: Counterexample to [10].

function `atomicMin`, they use the following code to relax a vertex `j` which is successor of a frontier vertex `i`:

```
if (uc[j] > c[i]+w[i,j])  uc[j] = c[i]+w[i,j];
```

where the array `uc`, called the *updating cost array*, holds a copy of the array `c` before relaxing. Indeed, as the authors explain, the new cost is not reflected in `c`, but is updated in `uc`, in order to avoid read-after-write inconsistencies. Later, they dump `uc` onto `c` in the `update` kernel.

Unfortunately, this technique is not enough to avoid write-after-write inconsistencies. Concretely, if two frontier vertices `i` and `i'`, whose related threads are simultaneously running, satisfy `uc[j]>c[i]+w[i,j]` and `uc[j]>c[i']+w[i',j]` at the same time for the same U-vertex `j`, then both threads will make the above *if*-condition true. Thus, there will be no control on the final value assigned to `uc[j]`. Since debugging concurrent programs is highly difficult, we have defined the graph of Fig. 5 on the right in order to increase the number of these critical situations.

We have run their implementation on a GeForce 8800 GTS, similar to the GeForce 8800 GTX used in [10], with n=1024 and 32 threads per block. Furthermore, we have also tested the undirected version of the graph, since the authors do not specify the kind of graph they manage. In any case, observe that vertices ranging from 1 to $n-2$ will compose the frontier after the first iteration. Actually, $c[i] = uc[i] = 1$, for $1 \leq i \leq n-2$, and $c[n-1] = uc[n-1] =$ INFINITY, after the first iteration. Hence, every vertex $(1 \leq i \leq n-2)$, tries to relax $uc[n-1]$ to a different value during the second iteration. In consequence $c[n-1]$ ends with a value that randomly changes from execution to execution, instead of computing the right solution $c[n-1] = 2$.

Since threads of different blocks cannot be synchronized in CUDA, solving this bug requires the use of atomic functions in the `relax_F` implementation. Unfortunately such functions are only available from compute capability 1.1, so solving this bug for the cards GeForce 8800 GTS and GTX demands a deeper modification of the algorithm. This is actually the aim of our `relax_U` kernel.

5 Adjacency Matrices

In the case of adjacency lists, it is difficult to conceive a method to allow threads to collaborate when reading from global memory. On the opposite, when adjacency matrices are used, threads must visit every element of each column or row, and so, threads can cooperate to bring elements of arrays `f`, `c` or `u` to shared memory.

As we did for the adjacency list representation, we can consider two kind of implementations: one that looks for predecessors (U^{CPU} and U^{GPU}), and another one that looks for successors (F^{CPU} and F^{GPU}). In U^{GPU}, each thread t must look for its predecessors by visiting the t–th column. In order to make threads collaborate, the exploration is divided in chunks of b elements, where b is the number of threads in a block. The arrays `f` and `c` are also divided in chunks of b elements. Before visiting the chunk of the column, each thread brings a component of the chunk of `f` and `c` into shared memory. That way, the information of the arrays `f` and `c` is already available when each predecessor within the chunk is processed. Once a chunk is dispatched, the next one is processed identically.

On the other hand, F^{GPU} processes frontier vertices, so each thread explores a row. Threads can also collaborate similarly, but this time they bring elements of `u`.

6 Results and Discussion

We have tested all the implementations using an Intel CORE 2 QUAD Q6600 2.40 GHz 2GB DDR memory, and a NVIDIA GEFORCE GTX 280, which has 30 multiprocessors and 1 GB of GDDR3 memory, using 256 threads per block. The database is composed of randomly generated graphs with a number of vertices that ranges from 1 to 11 M for adjacency lists, and from 1 to 15 K for adjacency matrices. The database includes 25 graphs for each of these sizes. The degree of each graph is fixed, so every vertex has the same number of adjacent vertices. The chosen degree is 7 for adjacency lists, while $n/5$ for matrices. Concerning lists, graphs have been generated using the predecessor interpretation, so we have also turned each graph into

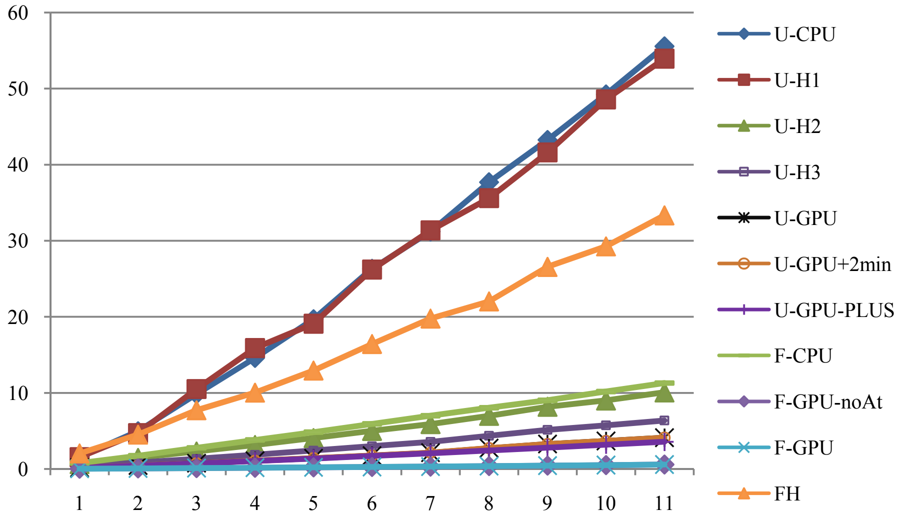

Fig. 6. Results for adjacency lists. Units: seconds for the y-axis and 2^{20} vertices for the x-axis.

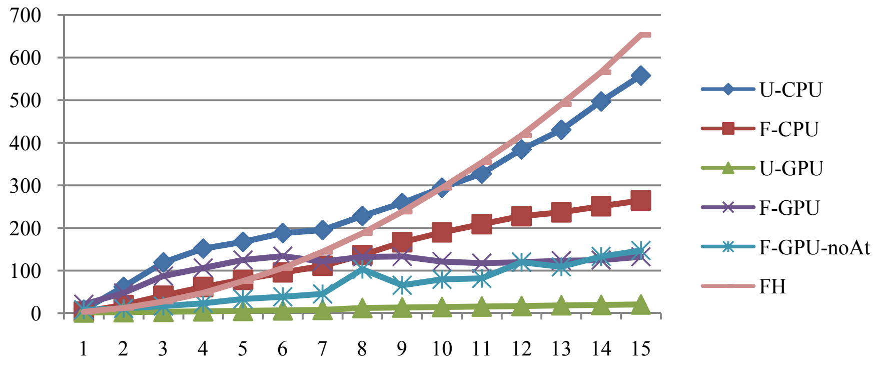

Fig. 7. Results for adjacency matrices. Units: ms. for the y-axis and 2^{10} vertices for the x-axis.

its successor interpretation. Notice that the degree of the graph can not to be kept after this operation. Edge weights are integers that randomly range from 1 to 10.

Figures 6 and 7 show the results we have obtained comparing the average times for each solution to a CPU-solution, called FH, implemented using Fibonacci Heaps and based on the SPLIB library [16]. Most of our solutions, including some CPU ones, run faster on these graphs since the arising frontiers are large. Thus, our solutions only require a few iterations to solve the problem. Concretely, around 45 are enough to solve the largest graphs represented with adjacency lists.

Let us analyze the results for the adjacency list representation presented in Fig. 6. The figure shows that fully CUDA-implemented solutions (U^{GPU}, $U^{GPU+2min}$ and F^{GPU}) are more efficient than partially CUDA-implemented ones (U^{H1}, U^{H2} and U^{H3}), which is due to the overhead connected to the data movement between CPU and GPU. The figure also shows that a two-pass minimization behaves as a single one, since U^{GPU} and $U^{GPU+2min}$ overlap. This can be explained comparing the number of values provided by `minimum1` to those provided by `minimum2`. Notice that these numbers are $n/(2b)$ and $n/(2b)^2$, respectively, where b is the number of threads per block. Since n ranges from $1 * 2^{20}$ to $11 * 2^{20}$ and we have chosen $b = 256 = 2^8$, these numbers finally range from 2^{11} to $11 * 2^{11}$ for `minimum1` and from 2^2 to $11 * 2^2$ for `minimum2`. Therefore, the number of values that must be copied from GPU to CPU, in order to be minimized on CPU, is similar for U^{GPU} and $U^{GPU+2min}$; so there is no difference in time consumption. The figure also shows that exploiting CUDA resources leads to better results, since U^{GPU_PLUS} is slightly faster, obtaining a factor near 10X w.r.t. FH.

Concerning solutions based on the `relax_F` procedure, Fig. 6 shows that processing unresolved vertices is slower than processing the frontier, even for parallel implementations, since F^{GPU} is quite faster than U^{GPU}. Also notice that $F^{GPU\_no\_Atomic}$ behaves as F^{GPU} because the simultaneous accesses to the same `c`-component are rare when the degree is small. These solutions reach a factor around 60X w.r.t. FH.

Regarding adjacency matrices (Fig. 7), the more vertices the graph has, the higher is the degree. Thus, the frontier sets are huge, and `relax_F` based solutions are slower than `relax_U` based ones. To summarize, U^{GPU} is the fastest, achieving a factor of 32X w.r.t. FH. Finally, the figure gives more insight about how atomic operations affect the overall performance, since $F^{GPU\_no\_Atomic}$ is usually faster than F^{GPU}.

7 Conclusions

GPUs can be used to speed up solutions to many problems, including classic problems. Nevertheless, the CUDA programming model is very restricted concerning synchronization, so implementations must be carefully designed, and intuitions about their correctness should be given at least.

In the paper we have shown different CUDA solutions for the SSSP problem, considering adjacency lists and matrices. We have also explained the bug we found in [10], which is basically due to write-after-write inconsistencies. In order to solve this bug, two approaches have been shown. On the one hand, atomic functions can be used for devices of compute capability 1.1 and higher. On the other one, the usual `relax` procedure can be reversed in order to process unresolved vertices instead of frontier vertices. Although processing unresolved vertices is theoretically less efficient, the latter approach is the only applicable solution to any CUDA device.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Num. Math. 1, 269–271 (1959)
2. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms, 2nd edn. MIT Press, Cambridge (2001)
3. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34, 596–615 (1987)
4. Meyer, U., Sanders, P.: Δ-stepping: A parallel single source shortest path algorithm. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 393–404. Springer, Heidelberg (1998)
5. Meyer, U., Sanders, P.: Δ-stepping: a parallelizable shortest path algorithm. J. of Algorithms 49, 114–152 (2003)
6. Brodal, G., Träff, J., Zaroliagis, C.: A parallel priority queue with constant time operations. J. Parallel and Distributed Computing 49, 4–21 (1998)
7. Madduri, K., Bader, D., Berry, J., Crobak, J.: An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In: Proc. Workshop on Algorithm Engineering and Experiments (ALENEX 2007) (2006)
8. Di Stefano, G., Petricola, A., Zaroliagis, C.: On the implementation of parallel shortest path algorithms on a supercomputer. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) ISPA 2006. LNCS, vol. 4330, pp. 406–417. Springer, Heidelberg (2006)
9. `http://www.nvidia.com/object/cuda_home.html#`
10. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
11. `http://www.gpgpu.org/`
12. Buck, I., Purcell, T.: A toolkit for computation on GPUs. In: GPU Gems, ch. 37. Addison-Wesley, Reading (2004)
13. Harris, M., Sengupta, S., Owens, J.: Parallel prefix sum (Scan) with CUDA. In: GPU Gems, ch. 39, vol. 3. Addison-Wesley, Reading (2008)
14. Harris, M.: Parallel prefix sum (Scan) with CUDA (2007), `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf`
15. `http://www.nvidia.com/object/cuda_get.html`
16. `http://avglab.com/andrew/soft.html`