CUDA Single Source Shortest Path

Madhu Sudhanan
dept. Electrical and Computer Engineering
Stony Brook University
New York, USA

***Abstract*—This report explores and summarizes on the algorithm, problems tackled and implementation of the Single Source Shortest Path algorithm between two points in a matrix array using CUDA parallel computing.**

***Index Terms*—SSSP, CUDA, parallel computing algorithms**

I. Introduction

There are two parts to this problem. First, given two square matrices A and B, we find the product matrix along with the first two minimum values and their respective indices. Second, we use the first minimum as the source and the second minimum as the destination to find the shortest path between the two. Finally, we integrate OpenCV to show an intuitive image of the computed path.

II. Parallel matrix multiplication and minima

A. Parallel matrix multiplication

We use the tiled matrix multiplication to better utilize CUDA shared memory which greatly improves performance. The shared memory within each CUDA block stores the sub-matrix A and B each of the size of a specific tile.

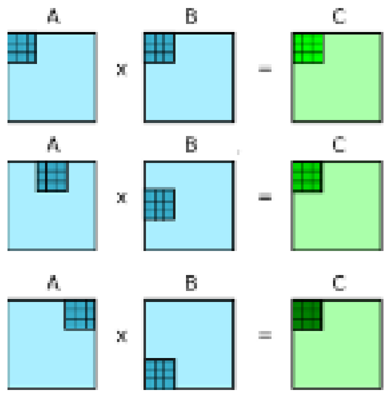

Fig. 1. Tiled matrix multiplication

B. Block reduction

We perform block reduction to find the first minimum value. Each block has 1024 threads, out of which half of the threads compare it's value with a value of another thread that is of ”stride” units more where stride is constant offset per reduction iteration. For example, if stride is 512, thread 0 compares it's value with thread 512, thread 1 compares it with thread 513, thread 2 with thread 514 and so on until thread 511 with thread 1023.

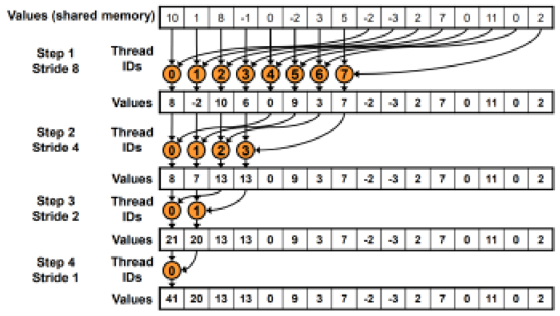

Fig. 2. Here, all 16 threads have their own allocated space in shared memory. So, thread 0 would compare the value at index 0, which is 10, with the value at index 8, which is -2. This means in step 1, only 8 threads would be involved in the reduction

As we find the minimum value, we also keep track of it's corresponding row and column by utilizing a user defined structure as shown below.

```
typedef struct
{
    float value;
    int16_t row, col;

}matElement;
```

C. Finding second minimum

We first replace the value at the index of the first minimum with the maximum float value and run the block reduction kernel to find the minimum. This would now be our second minimum value who's row and column is also stored. After this, we replace back our first minimum value in the result matrix. We now have our two minima and our result matrix and can proceed to find the shortest path between them.

III. Single Source Shortest Weighted Path

A. Graph representation

We first represent the matrix values and their corresponding neighbours using a few graph representations as shown in figure 3.

Vertex array Va is of the same size as matrix size where the index of Va array represents the index of the matrix array. The values stored in Va contains the starting index of it's corresponding neighbours (referred henceforth as edges, Ea).

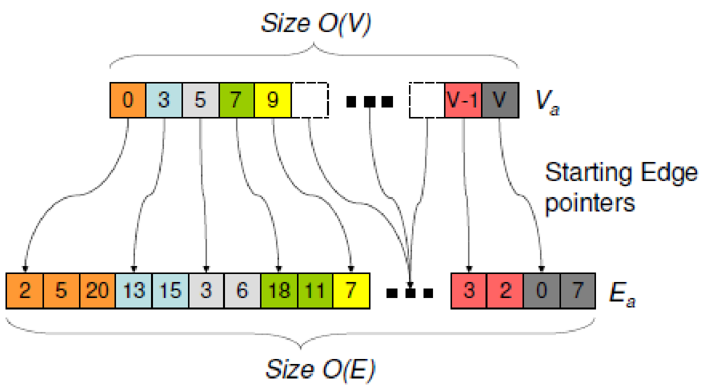

Fig. 3. Graph representation

In this implementation for matrices, we consider all points to have fixed number of edges (4 inside the matrix and 2/3 on its boundaries). For each point, the weight **(W)** of the edges in each direction is the float value of the point.

We also use an array of type **pathElement** called **intermediateCostAndPath (Ci)** with size that of matrix size to store intermediate cost values and the index that the current value points to as the path in the current iteration. This is mainly used to resolve read after write inconsistencies in the global memory. Updates to the intermediateCostAndPath array should lock the memory location before modifying it as many threads may write different values at the same location concurrently. For this purpose we implement and atomic min function discussed later.

```
typedef pathElement
{
    float value;    //intermediateCost
    int pathIndex;
}
```

The value **pathIndex** stores the index of the node with the least intermediateCost which would give the path.

We define a cost array **(C)** with size that of matrix size which stores the final cost values.

We define a threadMask array with size that of matrix size which stores a boolean 1 or a 0. It is used to specify which thread needs to compute the intermediate cost values and path in any iteration.

We use a global variable "terminate" to indicate weather all threads are done computing their cost values and that no more computations are remaining.

B. SSSP algorithm

We use two CUDA kernels to perform the SSSP algorithm. In kernel 1, initially, the threadMask of the source is 1, hence that thread would run and compute the intermediate values and path of it's corresponding neighbours. Ci, C and W represents intermedateCost, cost and weight respectively.

```
//Kernel 1 sudo code
tid = getThreadId()
if(threadMask[tid] is 1) then
    threadMask[tid] = 0
    for all neighbours nid of tid do
        atomic min start
        if(Ci[nid] > (C[tid] + W[nid]) then
            Ci[nid] = C[tid] + W[nid]
        end if
        atomic min done
    end for
end if
```

Now kernel 2 computes the final costs of each of the values in the matrix array. It will compute if the already existing cost of a thread is more that the computed intermediate cost. If it is, then that means we have a better path from another node specified by the intermediate cost array. The thread will update its final cost and flip it's threadMask to 1 so in the next iteration it can run to update it's own neighbours. This continues until all the threads do not flip the threadMask which means the shortest path has been computed.

```
//Kernel 2 sudo code
tid = getThreadId()
if(Ci[tid] < C[tid]) then
    C[tid] = Ci[tid]
    threadMask[tid] = 1
    terminate = false;
end if
Ci[tid] = C[tid]
```

C. Atomic minimum

As mentioned previously, the memory operation on intermediateCost must be done atomically as many threads may modify the same location concurrently. Since CUDA doesn't provide any built-in library function for atomic min operations, we can implement it using atomicCAS (atomic Compare And Swap) function which is supported by CUDA.

```
    atomicCAS(addr, val, swapVal);
```

The **atomicCAS()** function takes in three arguments and compares the value at address **addr** with **val**. If they are equal then it updates the memory location **addr** with the **swapVal** and returns **val** else it does not do anything and just returns whatever value is located at address **addr**.

Consider the following order in which the atomic operations are performed as shown in figure 4: **Thread 1, Thread 3, Thread 5, Thread 7**

- **Thread 1** checks and and doesn't modify ImCst.
- **Thread 3** checks and sees there is a better path and modifies ImCst to **40**.
- **Thread 5** compares with the new ImCst, **40**, and finds an even better path and modifies ImCst to **37**.
- **Thread 7** checks with the latest ImCst, **37**, and ignores as we already have a bette path.

However, a problem that arises here is that we need to perform the atomic swap on the array of type **pathElement** but CUDA

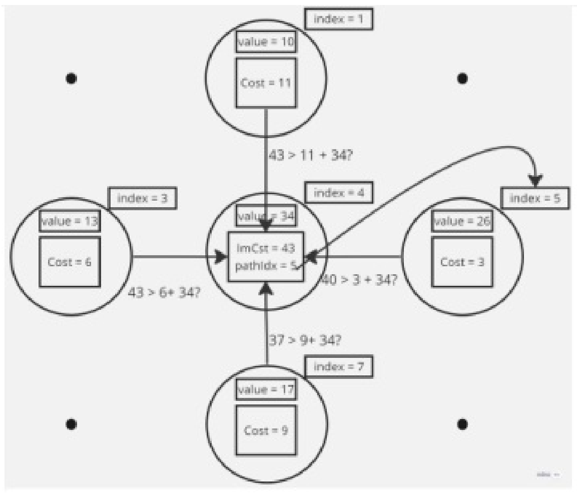

Fig. 4. An example 4x4 matrix where dots represent matrix values/thread not of interest. Circles represent the matrix values/threads where index 1 would represent CUDA thread 1, big rectangular box inside circle labeled with index 4 represents the **pathElement** type in intermediateCostAndPath array for that specefic index.

atomicCAS() only supports a few primitive datatypes such as int, unsigned long long int, etc. To overcome this, we first covnert the 8-byte **pathElement (pE)** pointer to a pointer of type **unsigned long long int (ulli)** which is also 8-bytes using the built-in compiler directive **reinterpret_cast**<>(). Once we do this, we can then perform our atomic operation on the address of type **ulli**.

```
atomicMin(pE* addr, pE* path)
{
ulli currPth = reinterpret_to_ulli(addr)
while(path.val
        < reinterpret_to_pE(currPth).val)
    {
        ulli old = currPth
        currPth = atomicCAS((ulli*) addr,
        old, reinterpret_to_ulli(path))
        if(crrPth == old) break;
    }
    return reinterpret_to_pE(currPth)
}
```

IV. Visualization of Path

We use the open source library OpenCV to visually show the path. The final paths are stored in the reverse order, that is, the destination node points to it's next node, which in turn points to it's next node until we reach the source (shown in figure 5). We recursively traverse this path from the destination node and display the path from source to destination. A sample is shows in figure 6.

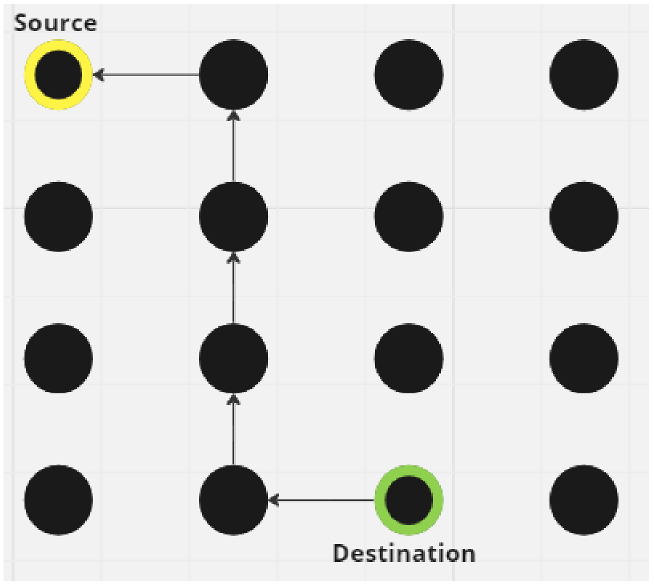

Fig. 5. Path computed in reverse order

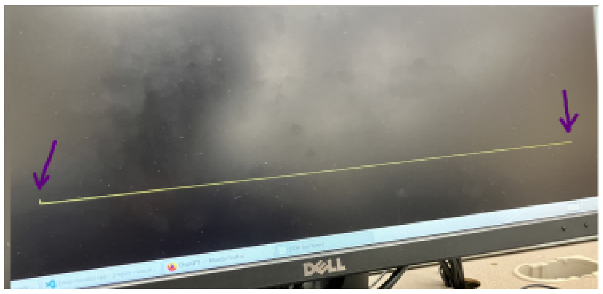

Fig. 6. An sample path where the arrow on the right is the source and the one on the left is the destination