Implementing and Optimizing Washall's Algorithm using CUDA

Zhao Tuowen
StdID:5110309458

February 20, 2014

Abstract

This the experiments report for the course of High Performance Computing. The main purpose is to realize **Washall's Algorithm** under the **GPU Architecture** using **CUDA**, and to optimize the implementation, as well as compare the execution time of different approaches, which also includes a sequential program running on the CPU. In the end of this report also provide some thoughts gathered through this experiment on how to effectively enhance the performance of a CUDA program.

1 Introduction

Washall's Algorithm, also known as the **Floyd-Washall's Algortims**, **Floyd's algorithm**, **RoyWarshall algorithm**, **RoyFloyd algorithm**, or the **WFI algorithm**, is a fast method to compute the minimum distance between any two arbitrary vertices, and can also compute the transitive closure of a given graph.

The idea behind the **Washall's Algorithm** is dynamic programming: if the minimal distance between any two vertices is known with the path only consist of a subset of the vertices, V' , then we can add a new vertex v, and obtain the minimal distance with path containing the vertices in $V' + v$, by "relaxing" the distance of paths between all pairs of $x, y \in V'$ with the newly added vertex v through this formula, $d_{xy} = \min\{d_{xv} + d_{vy}, d_{xy}\}$.

By closely analyze the algorithm, we find that the correctness of it holds as long as there doesn't exist an negative circle, that is, for every $v \in V$, $d_{vv}^{min} \geq 0$.

```
for k:=0 to N-1
for i:=0 to N-1
for j:=0 to N-1
  if d[i][k]+d[k][j]<d[i][j] then
    d[i][j]=d[i][k]+d[k][j];
```

Listing 1: Pseudo-code for the Basic Washall's Algorithm

Notice that the ordering of the (i,j) pair is inconsequential. In a single iteration of k, d_{ij} is only dependent on d_{ik} and d_{kj}; d_{ik},d_{kj} are independent:

We can assume that d_{ik}^{new} is dependent on some value, and prove it by contradiction. Then we have $d_{ik}^{old} < d_{ik}^{old} + d_{kk}$. Then we must have $d_{kk} < 0$ which contradicts with the assumption that the graph contains no negative circle. Thus proved.

1.1 The GPU Architecture

Originally designed for computer graphics, the GPU follows a different line of evolution from CPU. The graphics processor was designed to work at the pixel level; massive parallelism is adopted to provide more processing power.

I used Nvidia 770 GTX during the programming and testing phase of this experiments. This graphics card has 1536 CUDA cores, with memory bandwidth 224.3 GB/sec, and support compute capability 3.0 [3].

At hardware level, the device consist of 8 **SIMD Multiprocessors** , **SM** ; each multiprocessor contains 196 CUDA cores and can to compute the same instruction on multiple data, hence SIMD. The execution is synchronized by warps; each warp consists of 32 different execution threads. Some of it can be masked to execute nothing if the threads diverge[1].

Each SM have constant & texture caches, and a shared memory block. Every thread registered to the same SM can access the shared memory, acting like cache in the CPU, a staging area for the data in the global memory space. However, in CUDA, the shared memory is managed explicitly. Namely, it is manipulated using specific instruction for loading and storing. Also, it is mapped into 16 different banks to improve access speed for multidimensional data. A store/load can be completed in a single instruction if the threads in a half-warp access elements in different banks or the same one[2].

At software level, The threads are specified by **kernels** in the source, and are mapped to a maximum two three-dimensional space. The first three-dimensional space is the blocks. Each block is accessed through ***threadIdx*** and ***blockDim*** in the kernel. The second three-dimensional space is the grids. Each grid is accessed through ***blockIdx*** and ***gridDim*** in the kernel.The host can invoke a kernel using this piece of code[2]:

```
KernelName<<<GridDimension , BlockDimension>>>
            ( ... Params ... ) ;
```

1.1.1 Naive CUDA Implementation

This implementation is the most straight forward way. Every time invoke the kernel that compute all d_{ij} for a specific k, as shown in the pseudo-code. The value of k is enumerated in a loop in the host code.

```
for (int k=0;k<dim.num;++k)
{
   cuFloydKnl<<<dim3((int)ceil(dim.num/double(BLKSIZE)),dim.
    num,1),BLKSIZE>>>(cuMat,dim,k);
   cudaThreadSynchronize();
}
```

Listing 2: Host Code for Naive Washall's Algorithms

```
template<typename T>
__global__ void cuFloydKnl(T *adjmat,const MatrixDim dim,
    const unsigned int k)
{
unsigned int row=blockIdx.y;
unsigned int col=BLKSIZE*blockIdx.x+threadIdx.x;
T rowdat;
T len;
T nlen;
rowdat=adjmat[row*dim.stride+k];
nlen=rowdat+adjmat[k*dim.stride+col];
len=adjmat[row*dim.stride+col];
adjmat[row*dim.stride+col]=nlen>len?len:nlen;
}
```

Listing 3: Kernel Definition for Naive Washall's Algorithms

This program creates a thread for every element to be updated; every block is bounded by the number of threads it can hold; the map is padded so that no divergence of control flow is needed. The thread per block number is fixed at 256 to keep the kernel occupancy high.

1.1.2 Reusing Data

This implementation is a slightly improved one of the previous version, in that every thread computes multiple elements instead of one, marginally reduced the total reads from the global memory. Each threads access elements in strides so that the read and write from the global memory space is coalesced:The m^{th} element thread accesses is with index start+i+m*stride. Each thread in this implementation compute 4 elements.

1.2 Result and Comparison

These two kernels are memory intensive in that they only perform limited operation on every elements to be transmitted through the bandwidth. As pictured in the following graph, the second is not much better than the first in that the data reuse is limited: every elements had to read and write at least once for each value of k to do the comparison.

Both of the CUDA version achieves 100% occupancy, but the performance is reduced by insufficient use of global bandwidth and the extremely imbalanced use of function unit. For a element to be updated, 3 reads and a write though the global memory bus is executed, but only compute one addition and a compare. The second version improved but a little: only 3/4 read is reduced from the equation.

Due to the memory intensive nature of this algorithm. We can use the following formula to measure the bandwidth usage in order to quickly compare different method(the lower the better):

$$\mu = \frac{Number\ of\ read\ \&\ write}{Block\ width^{1}}$$

When the memory bandwidth is the most influential problem, the speed-up can be roughly calculated as:

$$SpeedUp = \frac{\mu_{old}}{\mu_{new}}$$

[1]The block width is 1 in these two programs.

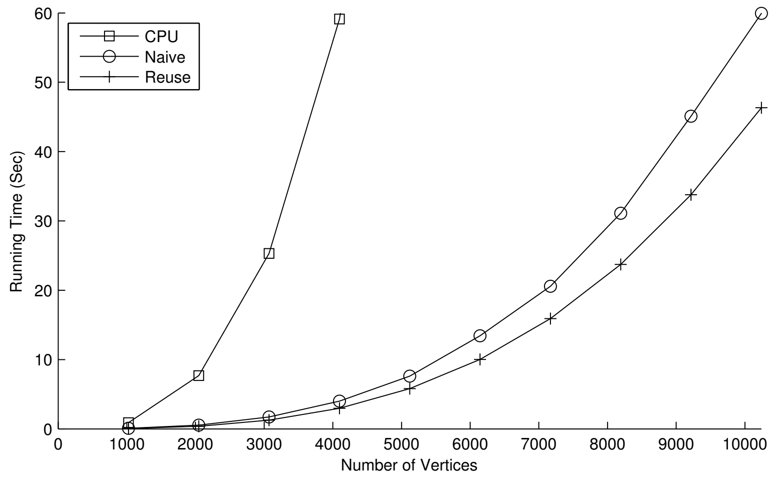

Figure 1: Comparison between CPU, Naive CUDA, and CUDA with Reusing

The speed up of the second version of the CUDA program is estimated as 1.200. Using the execution time obtained with 11264 vertices, the actual speed up is 1.282. The estimation is not far off from the real one.

2 Blocked Approach

Han et al.[4] suggested the blocked approach to allow loop unrolling and cache utilization under the CPU architecture. However, it also greatly improve data reuse, and shared memory usage, through calculating multiple instances of k in one go.

By closely analysis the data dependency of the algorithms between different iterations of k. During several consecutive loops of k, from a to b. $I = \{d_{ij} | i, j \in [a, b]\}$ only depends on the values in itself, which we call it the independent block. $S = \{d_{ij} | iorj \in [a, b], butnotboth\}$ only depends on the values of itself and that of the independent block, which we termed the singly-dependent block. All other elements d_{ij} depends on the values in S, which we call it the doubly-dependent block. This three set constitute all of the data elements in the graph. And we can compute according to the orders above, as shown in the following code(N is divisible by bw, block width):

```
for s:=0 to N/bw−1 do
begin
  −− For Indepenent Block
  for k:=0 to bw−1
  for i:=0 to bw−1
  for j:=0 to bw−1
     if d[s*bw+i][s*bw+k]+d[s*bw+k][s*bw+j]
        <d[s*bw+i][s*bw+j] then
           d[s*bw+i][s*bw+j]
              =d[s*bw+i][s*bw+k]+d[s*bw+k][s*bw+j];
  −− For Singly−dependent Block
  for k:=0 to bw−1
  for i:=0 to bw−1
  for j:=0 to N−1
  begin
        if d[s*bw+j][s*bw+k]+d[s*bw+k][i]
          <d[s*bw+j][i] then
            d[s*bw+j][i]
              =d[s*bw+j][s*bw+k]+d[s*bw+k][i];
        if d[i][s*bw+k]+d[s*bw+k][s*bw+j]
          <d[i][s*bw+j] then
            d[i][s*bw+j]
              =d[i][s*bw+k]+d[s*bw+k][s*bw+j];
  end;
  −− For Doubly−dependent Block
  for k:=0 to bw−1
  for i:=0 to N−1
  for j:=0 to N−1
        if d[i][s*bw+k]+d[s*bw+k][j]
          <d[i][j] then
            d[i][j]
                  =d[i][s*bw+k]+d[s*bw+k][j];
end;
```

Listing 4: Block Washall's Algorithm

Lund and Smith[5] proposed **Staged blocked Method** to improve the performance by allowing more thread blocks resident on a SM through reducing usage of shared memory and utilizing the register space. However, for device with compute capability 3.0, this seems hardly necessary unless larger blocks(> 32) are used.

2.1 Result and Comparison

Using the formula suggested in the previous section we can estimate the expected speed up. However, the complexity of the program is to increase using the blocked method and using staged-load. The result may deviate by a constant factor when the programming technique change.

For the naive method:

$$\mu = \frac{3+1}{1}$$

For the block method with block width 32:

$$\mu = \frac{3+1}{32}$$

For block width 32, the memory bandwidth is no longer the most prominent problem. However, to further reduce the problem, we can use a block width of 64, using similar tweaks in the Staged-blocked Method.

$$\mu = \frac{3+1}{64}$$

Also, to avoid overuse of registers causing the kernel occupancy to drop, we can add a directive to the definition[2].

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
   minBlocksPerMultiprocessor)
MyKernel(...)
{
      ...
}
```

Listing 5: Upper-bonding Register Usage

However, this is not done by some tricky manipulation on the register itself, such as register packing to pack several short operand in a single register. Rather it is done by simply spilling that extra space needed to the local storage, which may cause some performance lost.

All of the program is implemented using similar coding style and all achieved 100% kernel occupancy, theoretically. The performance of these methods is shown in the following graph.

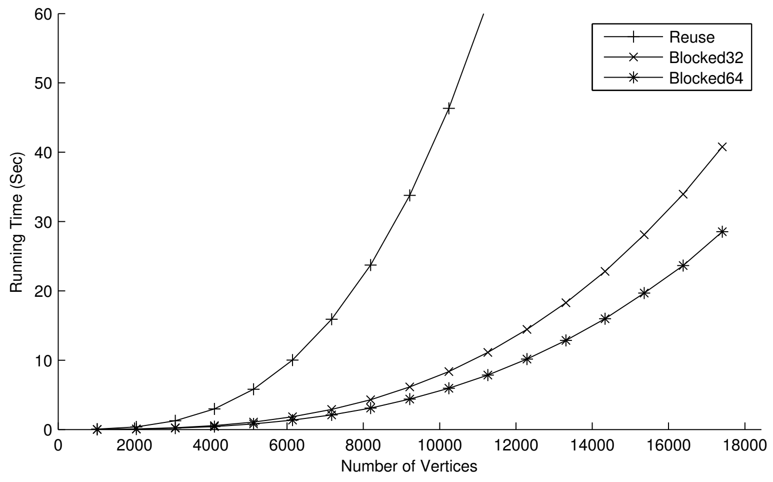

Figure 2: Comparison between CPU, Naive CUDA, and CUDA with Reusing

The prominent kernel (for doubly-dependent blocks) in the staged-blocked method with block width of 64 achieves balanced usage of function unit and memory, good utilization of shared memory, low local memory use, and theoretically full kernel occupancy. However, this algorithm still heavily relies on the load/store functions unit.

3 Conclusion

There are several notions of great importance in improving the performance of a CUDA program.

- Kernel occupancy
- Control flow divergence

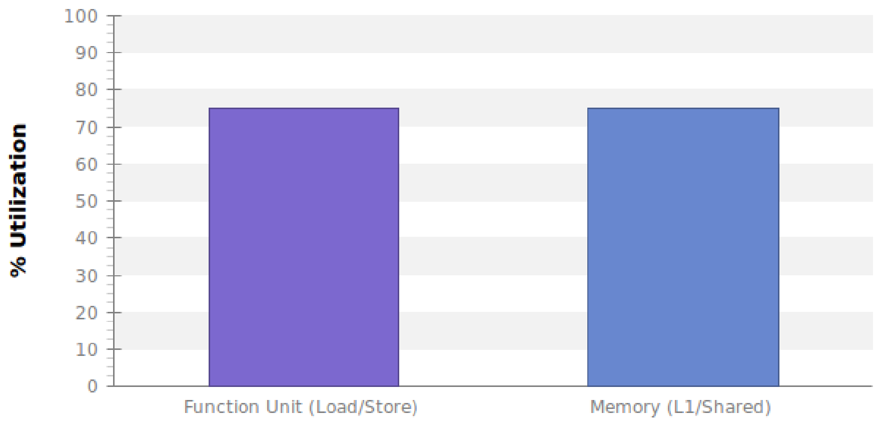

Figure 3: Comparison between Compute and Memory Utilization

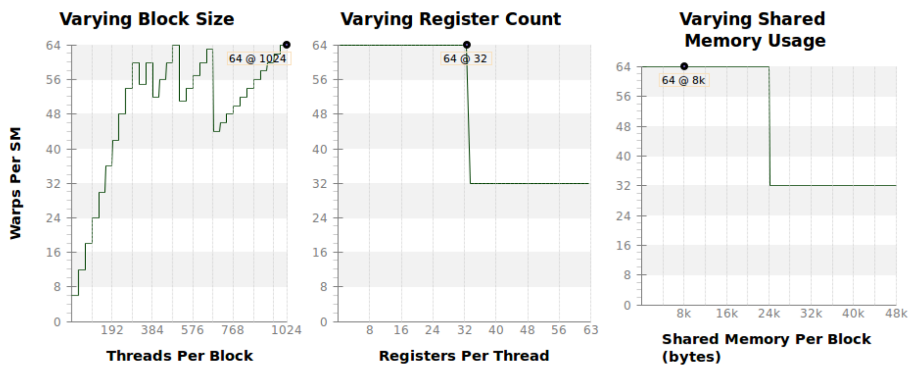

Figure 4: Achieved Occupany(theoretical) and Limiting Factors

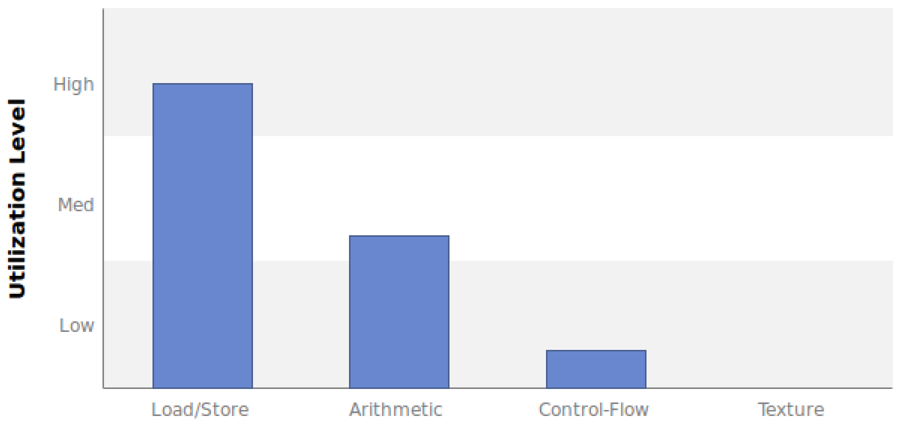

Figure 5: Load Comparison between Different Function Unit

- Function unit usage

Kernel occupancy can hide data latency. When one warp waiting for a memory transaction, a branch jump, or a synchronization point, another can resume its execution. It is limited by the resources available under the compute capability of the device, for example:

- Maximum threads per block
- Maximum threads per SM

- Maximum blocks per SM
- Maximum warp per SM
- Shared memory per SM
- Maximum Register per SM

Control flow divergence will cause threads in a warp to execute different instructions. Actually, they won't execute any differently, some threads actions are just masked by the SM, which means that their action become partly serialized. This can cause the most severe damage to the overall performance.

Function unit usage can improve parallelism in execution. The arithmetic unit can continue to work while a data transaction is occurring. A biased usage may cause the thread to be constantly waiting for one unit to free up, and the performance is thus limited by the throughput one type of unit.

Beyond these are more of improvements to the algorithm than that to the adaptation.

As evident in the analysis, CUDA provide an efficient way to program the GPU for general purpose calculations, which sometimes can be hundreds time faster.

However, the interface between the host and device is too complicated and exposed too much detail of the machine to the programmer. The difference between different device is huge[2]. As a result, to maximize performance, optimization has to be done under every specific environment.

The control flow is extremely simple, which on the one hand greatly simplified synchronization issues, while on the other hand hugely limited the complexity of the program. From my experience, instead of one bulky kernel to include everything, several lightweight kernels that operate in conjunction and each complete one simple step often prove to be more efficient. Also, this causes CUDA to be naturally data intensive. Complex calculations that calls for too much branching will be much less efficient.

In short, CUDA offers powerful alternative in tackling massively-parallel problems, but one has to be extra wary in designing the software to fully utilize its potential.

References

[1] CUDA C best practices guide. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, 2013.

[2] CUDA C programming guide 5.0. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2013.

[3] Nvidia GTX 770 specifications. `http://www.techpowerup.com/gpudb/1856/geforce-gtx-770.html`, 2013.

[4] S.-C. Han, F. Franchetti, and P. Markus. Program generation for the all-pairs shortest path problem. *Parallel Architecture and Compilation Techniques (PACT)*, pages 222–232, 2006.

[5] B. Lund and J. W. Smith. A multi-stage cuda kernel for floyd-warshall. *CoRR abs/1001.4108*, 2010.

Appendix A Program Time Chart

	Execution Time (millisecond)				
Vertices	CPU	Naive	Reuse	Blocked(32)	Blocked(64)[1]
1024	890	75.4846	48.6206	10.8614	10.5695
2048	7672	535.468	374.634	78.0474	65.5086
3072	25306	1722.28	1255.93	249.828	198.287
4096	59118	3999.22	2971.7	560.264	429.572
5120	115346	7620.96	5797.44	1074.35	802.604
6144		13430.3	10012.3	1837.27	1352.21
7168		20570.4	15903.4	2892.66	2103.58
8192		31098.2	23721.6	4288.22	3092.14
9216		45104.8	33785.8	6143.36	4384.66
10240		59932.8	46330.6	8359.62	5941.12
11264		79063.6	61671.4	11114.2	7856.6
12288				14432.1	10162.6
13312				18293.8	12847.6
14336				22818.8	15987.7
15360				28103.4	19689.8
16384				33943.4	23661.6
17408				40782	28533.8

Table 1: Run Time Comparison

The running time is obtained by averaging of 5 runs, excluding the time for I/O. Timing the CPU program uses ***clock()*** from ***ctime***, CUDA uses event(***cudaEventElapsedTime***).

Appendix B Final Program Source Code (Excerpts)

```
for (int k=0;k<dim.stride;k+=ITERSIZE)
{
  cuFloydIDKnl<<<1,dim3(ITERSIZE,16,1)>>>(cuMat,dim,k);
  cuFloydSDKnl<<<dim3(dim.stride/ITERSIZE,2,1),dim3(16,
   ITERSIZE,1)>>>(cuMat,dim,k);
  cuFloydDDKnl<<<dim3(dim.stride/ITERSIZE,dim.stride/
   ITERSIZE,1),dim3(16,ITERSIZE,1)>>>(cuMat,dim,k);
}
```

Listing 6: Host Invocation

[1]Also uses staged load. Every time puts 16*64 elements in the shared memory

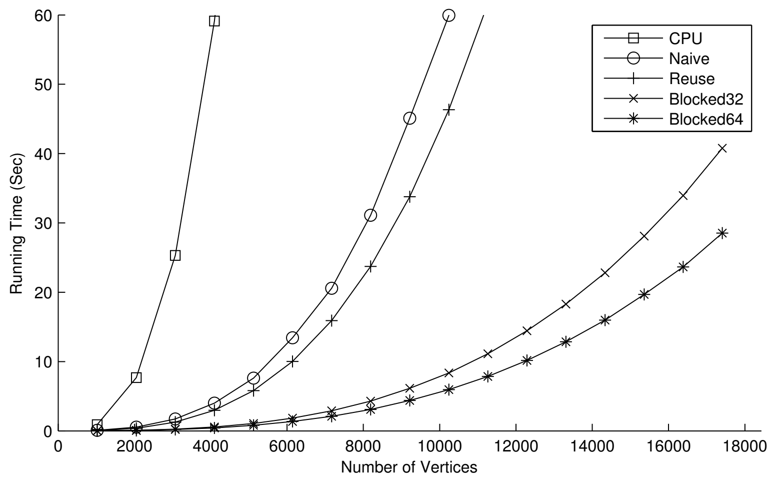

Figure 6: Run Time Comparison

```
template<typename T>
__global__ void cuFloydIDKnl(T *adjmat,const MatrixDim dim,
   const unsigned int node)
{
  unsigned int x=threadIdx.x,y=threadIdx.y;
  unsigned int pos=(node+y)*dim.stride+node+x;
  __shared__ T tmpL[4][16][16],tmpR[16][64];
  T len[4];
  T nlen;
  for (int i=0;i<4;++i)
    len[i]=adjmat[pos+((i*dim.stride)<<4)];
  for (int i=0;i<4;++i)
  for (int k=0;k<16;++k)
  {
    tmpR[y][x]=len[i];
    if ((x>>4)==i)
      for (int j=0;j<4;++j)
        tmpL[j][y][x&15]=len[j];
    __syncthreads();
    for (int j=0;j<4;++j)
    {
      nlen=tmpL[j][y][k]+tmpR[k][x];
      len[j]=len[j]>nlen?nlen:len[j];
    }
    __syncthreads();
  }
  for (int i=0;i<4;++i)
    adjmat[pos+((i*dim.stride)<<4)]=len[i];
}
```

Listing 7: Kernel for Independent Block

```
template<typename T>
__global__ void cuFloydSDKnl(T *adjmat,const MatrixDim dim,
   const unsigned int node)
{
  unsigned int x=threadIdx.x,y=threadIdx.y,ry,rx;
  ry=y&15;
  rx=(y&0xFFF0)+x;
  unsigned int bx=blockIdx.x,by=blockIdx.y;
  unsigned int posn,pos;
  __shared__ T tmpL[64][16],tmpR[16][64];
  T len[4];
  T nlen;
  if (by==0)
  {
    posn=(node+y)*dim.stride+node+x;
    pos=(node+ry)*dim.stride+bx*ITERSIZE+rx;
    for (int i=0;i<4;++i)
      len[i]=adjmat[pos+((i*dim.stride)<<4)];
    for (int i=0;i<4;++i)
    {
      tmpL[y][x]=adjmat[posn+(i<<4)];
      for (int k=0;k<16;++k)
      {
        //if (ry==k)
        tmpR[ry][rx]=len[i];
        __syncthreads();
        for (int j=0;j<4;++j)
        {
          nlen=tmpL[(j<<4)+ry][k]+tmpR[k][rx];
          len[j]=len[j]>nlen?nlen:len[j];
        }
        __syncthreads();
      }

    }
    for (int i=0;i<4;++i)
      adjmat[pos+((i*dim.stride)<<4)]=len[i];
  }
  else
  {
    posn=(node+ry)*dim.stride+node+rx;
    pos=(bx*ITERSIZE+y)*dim.stride+node+x;
    for (int i=0;i<4;++i)
      len[i]=adjmat[pos+(i<<4)];
    for (int i=0;i<4;++i)
    {
      tmpR[ry][rx]=adjmat[posn+((i*dim.stride)<<4)];
```

```
        for (int k=0;k<16;++k)
        {
          //if (x==k)
          tmpL[y][x]=len[i];
          __syncthreads();
          for (int j=0;j<4;++j)
          {
            nlen=tmpL[y][k]+tmpR[k][(j<<4)+x];
            len[j]=len[j]>nlen?nlen:len[j];
          }
          __syncthreads();
        }
        //__syncthreads();
      }
      for (int i=0;i<4;++i)
        adjmat[pos+(i<<4)]=len[i];
    }
}
```

Listing 8: Kernel for Singly dependent Block

```
template<typename T>
__global__ void
__launch_bounds__(1024, 2)
cuFloydDDKnl(T *adjmat,const MatrixDim dim,const unsigned
   int node)
{
  unsigned int x=threadIdx.x,y=threadIdx.y,ry,rx;
  ry=y&15;
  rx=(y&0xFFF0)+x;
  unsigned int bx=blockIdx.x,by=blockIdx.y;
  unsigned int posl=(by*ITERSIZE+y)*dim.stride+node+x
        ,posr=(node+ry)*dim.stride+bx*ITERSIZE+rx
        ,pos=(by*ITERSIZE+y)*dim.stride+bx*ITERSIZE+x;
  __shared__ T tmpL[ITERSIZE][16],tmpR[16][ITERSIZE];
  T len[4];
  T nlen;
  for (int i=0;i<4;++i)
    len[i]=adjmat[pos+(i<<4)];
  for (int i=0;i<4;++i)
  {
    tmpL[y][x]=adjmat[posl];
    posl+=16;
    tmpR[ry][rx]=adjmat[posr];
    posr+=(dim.stride<<4);
    __syncthreads();
    for (int j=0;j<4;++j)
    for (int k=0;k<16;++k)
    {
            nlen=tmpL[y][k]+tmpR[k][(j<<4)+x];
            len[j]=len[j]>nlen?nlen:len[j];
        }
        __syncthreads();
    }
    for (int i=0;i<4;++i)
        adjmat[pos+(i<<4)]=len[i];
}
```
Listing 9: Kernel for Doubly dependent Block