Danielle Maddix
May 30, 2014
CME 213 Final Project

Optimizations of Algorithms to Identify Strongly Connected Components in CUDA

Introduction

Identifying the connected components in graphs is a classical problem in graph theory. As discussed in the literature, the general problem involves using the adjacency list to find which nodes are connected to each other. When given an image as the initial data, the problem simplifies, since we no longer need the adjacency set and know the neighbors in the classical sense from the array. Every pixel is connected to the pixels adjacent to it, in the x and y directions. Still challenges arise in finding the neighbor sets with the same colors in an optimal fashion, where various methods are detailed in the literature.

The GTC conference details how to use overlapped blocks of shared memory to access the neighbors of a given pixel and then goes into hardware details specific for the new Kepler devices, which are not applicable to the Tesla devices on our computing resources [5]. Thus, our given implementation follows the pseudocode for the various kernels, namely A, B and D, as described in [1]. Kernel A performs a simple algorithm, where each thread represents a cell and checks whether its neighbors have the same color, using 4-connectivity, selecting the lowest label. This algorithm requires many iterations, since a label can only propagate by at most one cell. As shown in the below timing table, the performance is very poor and almost as slow as the serial algorithm on the CPU. Note we used Kernel A for benchmarking and also first tested that Kernel A worked properly, since Kernel B is an extension and optimization of Kernel A. Kernel B utilizes shared memory, rather than only relying on the much slower global memory accesses. In Kernel B, one thread block is labeled per iteration. It still requires some iterations, due to the boundaries which will change, but is significantly faster than Kernel A. As expected, Kernel D is by far the most efficient algorithm. It uses an algorithm most similar to the CPU serial algorithm of building equivalence classes of labels. Kernel D's method combines labels rather than cells, allowing it to process large meshes with very few iterations.

Implementation

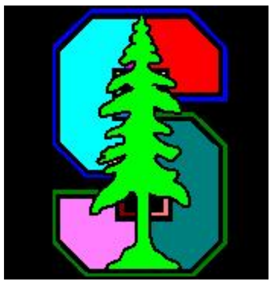
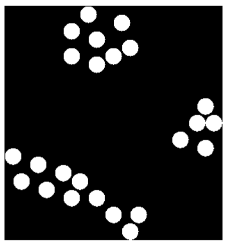
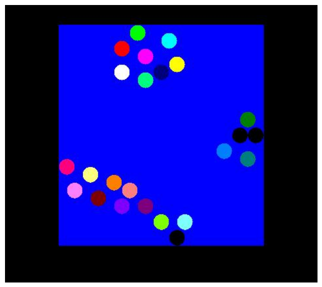
In Kernel A, we begin by creating a simple kernel to initialize the label/result array so that each element has a unique label. The code on the CPU, namely the host, repeatedly calls Kernel A, until no labels have been changed, where Kernel A finds the neighbors of the same color and takes the lowest label of these neighbors. A device boolean pointer is passed into the kernel which is initialized on the CPU as false every iteration, by using *cudaMemcpy* to copy the host value to the value stored in the device pointer. Once the kernel returns, the device boolean pointer is then copied back to the host to determine whether or not to terminate the while loop.

The first part of Kernel B is identical to that of Kernel A, that is, each thread acts as a cell and identifies its neighbors with the same color and takes the lowest of these labels. Instead of only one cell being updated per iteration, we design Kernel B so that an entire block is updated per iteration via the use of shared memory. Thus, we initialize a label block in shared memory and a boolean to shared memory to conduct a while loop until none of the shared memory labels have been changed. Caution must be advised with the use of *__syncthreads()*. Unlike in Kernel A, it is no longer safe to return the threads whose *tidx = blockIdx.x * blockDim.x + threadIdx.x* is greater than or equal to the number of pixels in the x-direction and analogously for y, since when calling *__syncthreads()* each thread must reach that point or a deadlock can occur depending on the compiler or hardware. Thus, instead of returning threads, whenever necessary we conduct bounds checks and only compute the neighbors on threads that are within the given range, allowing for all the threads to reach the *__syncthreads()* point. The host code is almost identical to that of Kernel's A, except for the hardware details, such as the choice of the number of threads per block, as discussed in the timing sections.

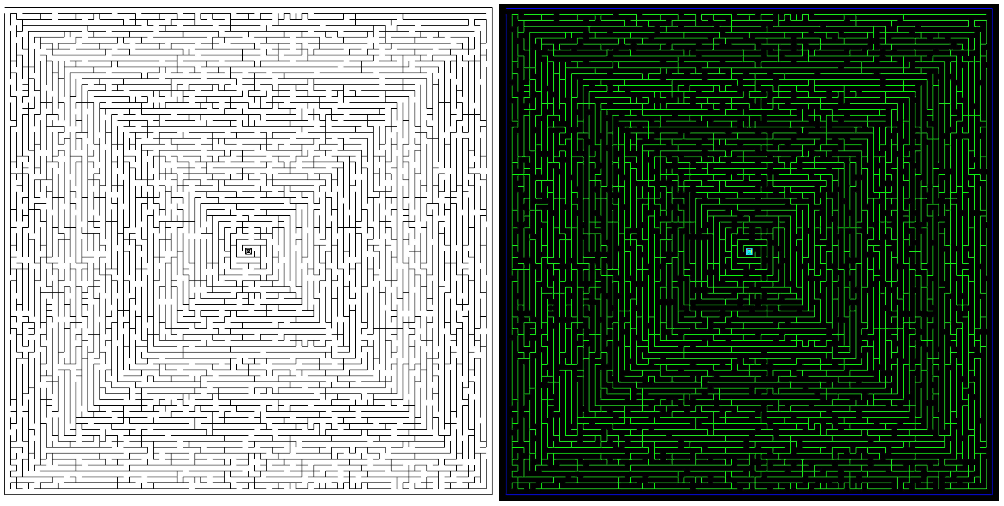
Kernel D is the most similar to the standard and fastest algorithm for identifying the connected components on the CPU, via equivalence classes. As noted in the literature, that algorithm is not simple to parallelize, so some changes are needed to make the implementation of the parallel version possible. Kernel D consists of three phases and thus three kernels, namely the Scanning Phase where the neighbors with the same state are computed and the smallest label is stored, the Analysis Phase, which uses the transitivity property of equivalence classes to resolve the labels and lastly the simple Labeling Phase which assigns each thread the label of its corresponding equivalence class. Note that in this case we begin by initializing both the unique label array and an additional unique label array for the equivalence classes. Both of these arrays just to be used on the device and so the *cudaMalloc* function is utilized. In all the cases, we must transfer the image data from the device to the host using *cudaMemcpy* and once the computation has finished, copy the result on the device back to the result host vector for further image processing and testing. We note that it is not necessary to pass in the image data array to the Analysis Phase and the Labeling Phase kernels, contrary to what is indicated by the pseudocode, since the image data is only utilized in the Scanning Phase to determine the number of neighbors with the same color. This is a simple optimization, which was made in the final version of the code.

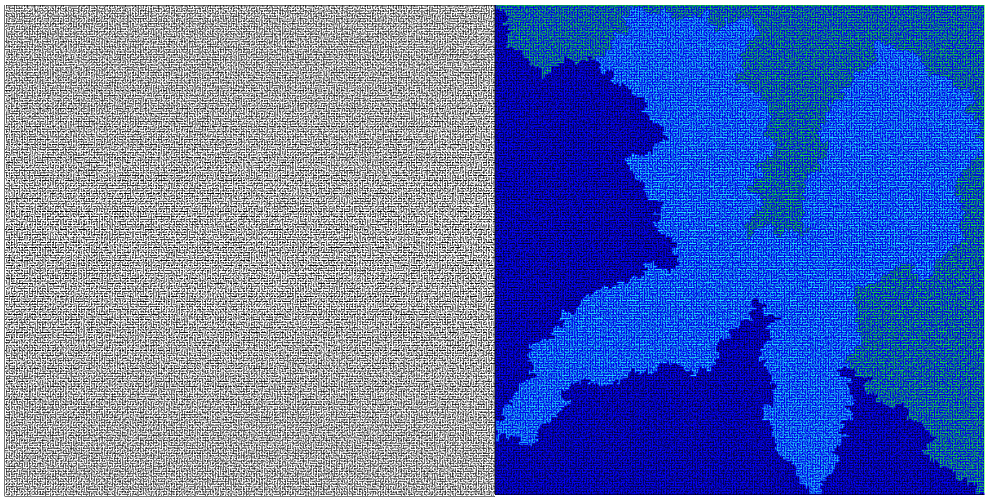
Image Results

The below results first show some of the smaller and simpler images, as a guide for what the algorithm is actually accomplishing before delving into the more complicated cases. On the left, we see the small Stanford image input and the output with the connected components labeled, as well as the simple balls images on the right. It is evident that each connected component is correctly identified and labeled.

The below maze images represent the largest tested data set size of millions of elements, $5.7x10^6$ in the largest case for maze.in, which is the first maze shown below. This shows that not only is the algorithm capable of resolving the connected components on a coarse level as in the above images, but also on a very fine level, as indicated by the second maze below, maze2.in.



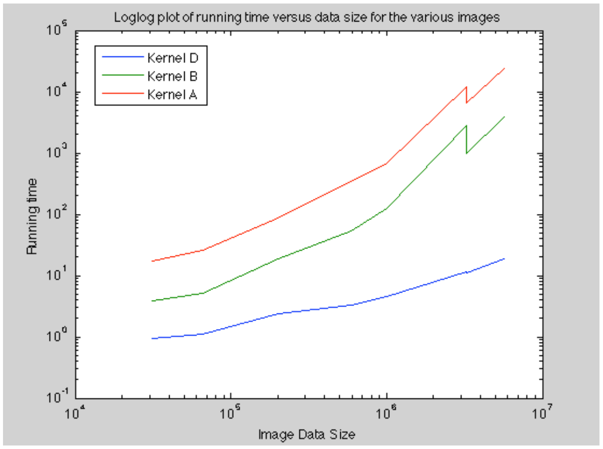


Performance and Data Analysis

After thoroughly testing that the kernels produced the correct and consistent results for all the images, timing results were conducted for each of the images. Note that the times include the time for the data transfers from the CPU to GPU. Furthermore, note that in running the code we have turned Kernel A off since it takes by far the most the time.

Images	Kernel A (ms)	16 Kernel B (ms)	32 Kernel B (ms)	Kernel D (ms)
maze.in	23803.6	5266.77	3803.62	18.3437
maze2.in	11817.3	3484.54	2848.66	11.2892
simple.in	336.007	75.6597	52.7191	3.29133
simple2.in	672.339	184.219	126.026	4.46285
smallballs.in	87.9673	22.515	18.6664	2.41507
smallcrosses.in	25.5015	6.41411	5.03501	1.09056
smallstanford.in	17.5999	4.0679	3.88627	0.931296
stanford.in	6594.6	1481.25	971.11	11.2306

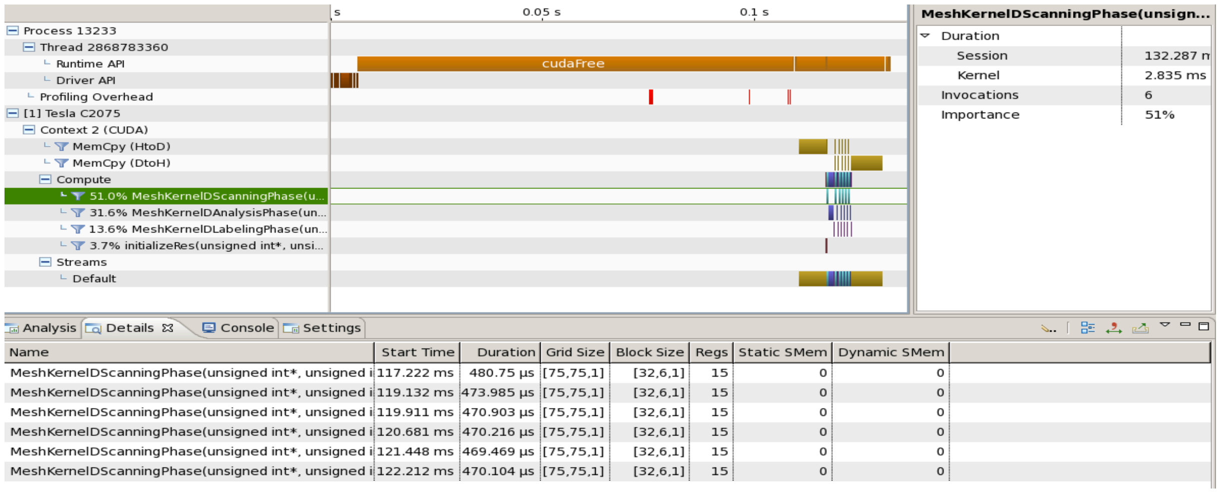
These timing results were done for a *32 x 6* threads per blocks for both Kernel A and D and as indicated by the 16 Kernel B was done on a *16 x 16* square thread block and the 32 Kernel B was done on a *32 x 32* square thread block. We utilize 2-dimensional thread blocks, so that each cell in the image can be represented as a thread. In Kernel B, the use of square blocks is used to take advantage of the caching in the first phase of global neighbor calculation and also to fully populate the shared memory. The caching and use of shared memory is very important when accessing neighbors. While it may take longer for neighbors to propagate in the larger *32 x 32* block in Kernel B, it is clear that the computations within a block are faster in accessing the neighbors due to the use of shared memory and so the *32 x 32* blocks seem to actually perform better than the suggested *16 x 16* blocks in [1]. A simple optimization as such was done by experimenting with the block size. In general we want about 192 threads per block with 32 threads in the x direction. The 32 threads enables coalescing, since an integer is 4 bytes and then one warp, the smallest collective unit can access 128 bytes, which is an entire cache line. We see that on the largest dataset, namely *2400 x 2380* for maze.in that Kernel D is about *1000x* times faster than Kernel A and even about *165x* times faster than the fastest version of Kernel B.



It is evident from the plot that Kernel D is much more efficient than the other kernels. Moreover, it also scales better, since as the data size increases, its running time is increasing much less quickly than that of the other Kernels. The curves of Kernel A and Kernel B exhibit about the same shape, with that of Kernel A shifted above Kernel B's, indicating the longer time Kernel A takes. It is also interesting to note that even though Stanford.in, has approximately *$2.0x10^{4}$* more pixels than maze2.in, more time is spent in maze2.in, explaining the kinks in the above plot.

Optimizations and Further Work

From the timing results, it is evident that Kernel D performs significantly better than Kernel B and A, by orders of magnitudes and so the majority of the optimization should be spent in optimizing Kernel D further. Thus, a first step is to profile Kernel D, as shown below and to find the inner kernel where it is spending the majority of time. A way to optimize this would be to use streams for concurrent memory transfer. This could help on the larger arrays, but it is evident from the profiling results that the majority of the time spent is not in transferring the data, but in the Scanning Phase, as expected since this is the most computationally expensive neighbor access. Note that streams could also produce minimal speed up, since we only need to copy the label array storing the result back and forth once and would be difficult to implement in this case.



MeshKernelDScanningPhase(unsign...	
Duration	
Session	132.287 m
Kernel	2.835 ms
Invocations	6
Importance	51%

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem
MeshKernelDScanningPhase(unsigned int*, unsigned i	117.222 ms	480.75 μs	[75,75,1]	[32,6,1]	15	0	0
MeshKernelDScanningPhase(unsigned int*, unsigned i	119.132 ms	473.985 μs	[75,75,1]	[32,6,1]	15	0	0
MeshKernelDScanningPhase(unsigned int*, unsigned i	119.911 ms	470.903 μs	[75,75,1]	[32,6,1]	15	0	0
MeshKernelDScanningPhase(unsigned int*, unsigned i	120.681 ms	470.216 μs	[75,75,1]	[32,6,1]	15	0	0
MeshKernelDScanningPhase(unsigned int*, unsigned i	121.448 ms	469.469 μs	[75,75,1]	[32,6,1]	15	0	0
MeshKernelDScanningPhase(unsigned int*, unsigned i	122.212 ms	470.104 μs	[75,75,1]	[32,6,1]	15	0	0

We see from the profiler that Kernel D is only called 6 times rather than 100s or 1000s of times as the other kernels. Since 51% of the time is spent in the Scanning phase and the Analysis Phase uses the results of the Scanning Phase to determine the equivalences via the transitivity property of equivalence classes, it is important to optimize the scanning kernel. One way to optimize the Scanning Phase, would be to launch *32 x 8* threads per block, but then to a loop over y. Each thread then updates 4 cells in this case rather than acting as a single cell. This is analogous to the y loop we did in the heat equation code. This can also potentially be used in Kernel B to speed up the shared memory calculations. Since we know that for caching the optimal shape is square, this allow us to still us 192 threads per block, but to obtain the desired square of *32 x 32* in each direction for coalescing in the y-direction as well. Furthermore, one could also consider the use of shared memory in the scanning phase of Kernel D, just as how we optimized the neighbor search from Kernel B to Kernel A with a while loop to minimize the numbers of labels that the Analysis must process. Lastly, another optimization could be found in avoiding the use of the *atomicMin* function, which is used to prevent a race condition, but can be slow waiting on all threads. We know from other parallel languages, such as openMP that it is important to minimize the usage of atomic operations whenever possible, especially in this case when we are using global memory accesses, rather than shared memory. This is another reason why the use of shared memory could be advantageous in this case. Another promising idea is to conduct some calculations on the CPU, while some of the other calculations are being done on the GPU.

References

[1] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with GPUs and CUDA. Parallel Comput., 36(12):655–678, December 2010.

[2] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider. Research note: Connected component labeling on a 2D grid using CUDA. J. Parallel Distrib. Comput., 71(4):615–620, April 2011.

[3] V. M. A. Oliveira and R. A. Lotufo. A study on connected components labeling algorithms using GPUs.

[4] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on CUDA. In IPDPS, pages 544–555, 2011.

[5] Zieglar G., Ceska, Connected Components Revisited on Kepler, GTC Conference. http://on-demand-gtc.gputechconf.com.