COMP429 Project 2

Batuhan Başkaya 64240 – Doğuhan Çiftçi 50356

We implemented all the 4 versions of the project completely. Each version is implemented according to the guidelines in the project pdf. Our version 3 is the best performant version. While we were implementing version 4, we thought it would be faster since we utilized shared memory for multiple E_prev accesses, however version 3 still performed slightly faster. Thus you can use version 3 for testing.

During our implementation, we encountered some problems and fixed these problems with the help of TA. First of all, we changed the main “cardiacsim.cpp” to “cardiacsim.cu” to fix a compilation problem. We also added “break”s into command line parser to fix an error. Finally we created a header file for cardiacsim_kernels to fix a problem in compilation.

Kernels

Each version of the program 1 through 4 can be run by giving the “-v 1” to “-v 4” command line arguments to the executable. ”-v 0” is reserved for the CPU serial implementation.

karrayinit

In “karrayinit”, we initialized the pointers of the arrays. Since these 2d arrays are formed from multiple 1d arrays and an 1d array or pointers, these pointers should have been reassigned when they were copied to the GPU. (Since they would have invalid pointers to the CPU memory.) This is done in the “karrayinit” kernel.

k1halos

In “k1halos” we initialized the halos of E_prev array, in accordance to the simulate() function. Each thread on the of the 2d array is responsible for the adjacent halos.

k1pde and k1ode

These kernels simply do the calculations in accordance to the simulate() function.

k2

In “k2” kernel, we just merged all the above kernels into one kernel. Since there is no global barrier in cuda, and the array pointer initializations must be done before any other operation, we assign 1 thread per each block to do the initializations and then call a block level “__syncthreads()” barrier. Also to save time, we only do this initialization on the first call to this kernel.

k3

In “k3” kernel, we created local variables “ee”, “e2” and “rr” to hold some numbers which will be used multiple times in the following lines. This would limit the reads from global memory.

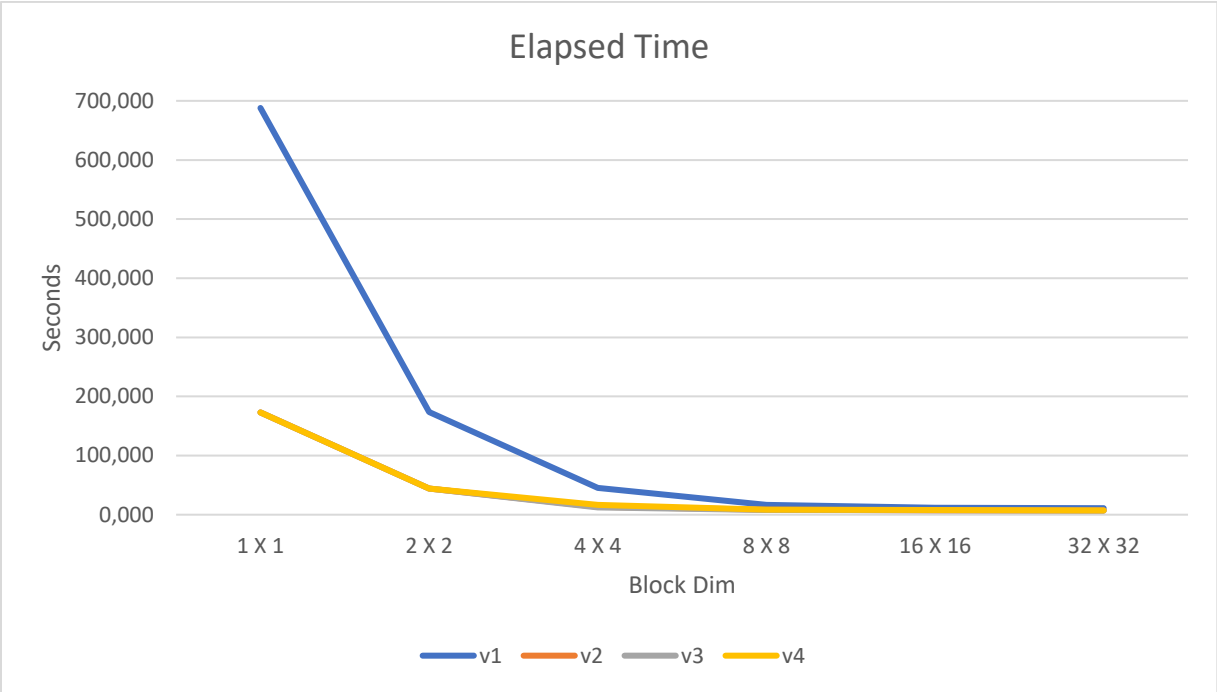
k4

In “k4” kernel, we created shared_E_prev on shared memory. Instead of making multiple readings from the global E_prev, threads now read from global once into shared memory, then make multiple readings from their fast shared memories.

Reporting Performance

When testing our code, we used 1024x1024 mesh points, 500 simulation time. For block sizes we used 1x1, 2x2, 4x4, 8x8, 16x16 and 32x32. We repeated each experiment 2 times and took their averages when plotting. We made our tests on a Tesla V100 device whose bandwidth limit is 731GB/s.Our findings are as follows:

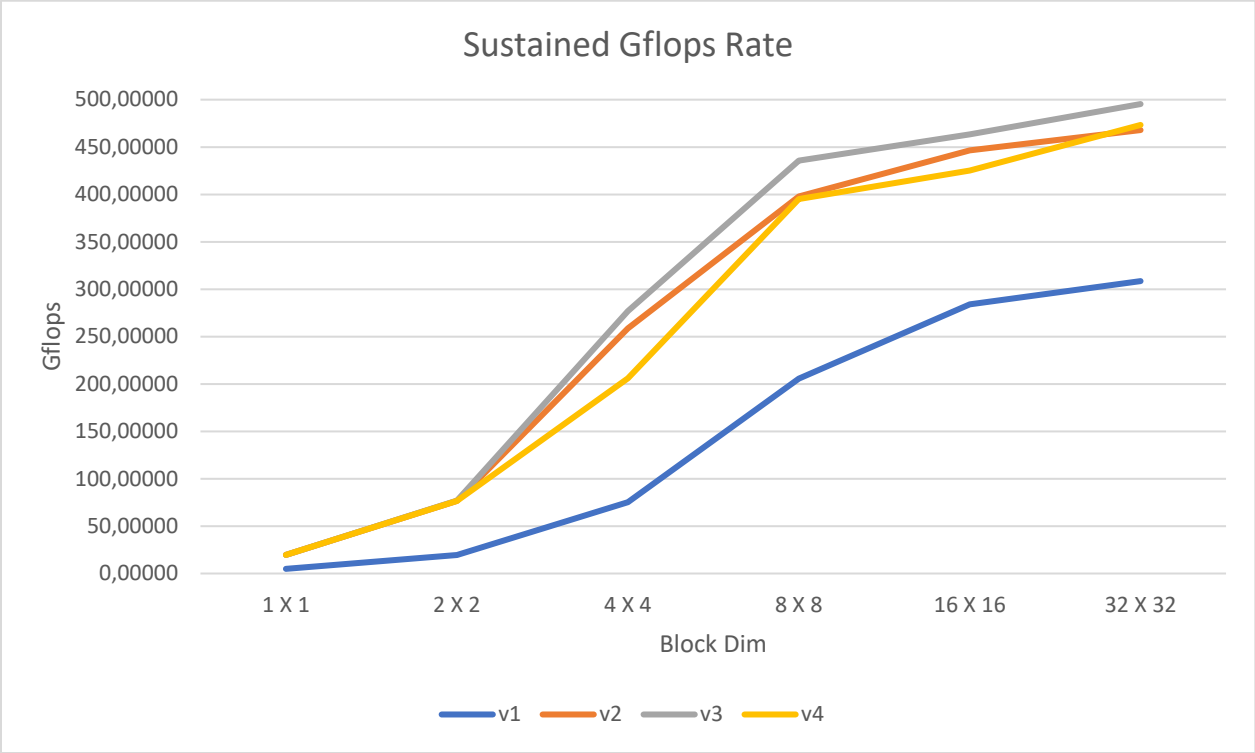
Elapsed Time



The lines of v2 through v4 collide on the provided graph because of the maximum of the graph.

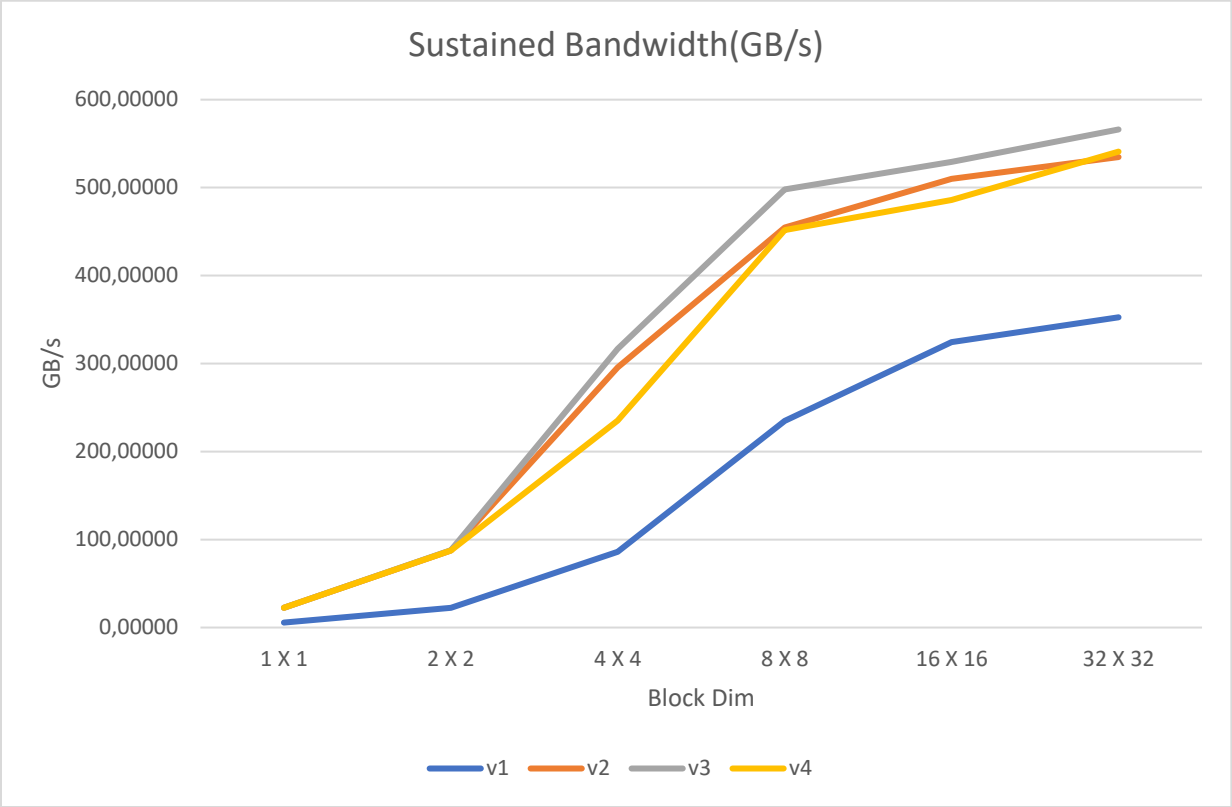
Elapsed Time					
Block Size	v0	v1	v2	v3	v4
1 X 1	2990,55	687,952	173,026	172,989	172,989
2 X 2		173,671	44,366	44,263	44,407
4 X 4		45,118	13,146	12,276	16,520
8 X 8		16,542	8,553	7,807	8,607
16 X 16		11,979	7,621	7,343	8,000
32 X 32		11,026	7,270	6,867	7,187

Sustained Gflops Rate



GFLOPS					
Block Size	v0	v1	v2	v3	v4
1 X 1	1,13748	4,94467	19,66000	19,66420	19,66430
2 X 2		19,58695	76,67410	76,85270	76,60275
4 X 4		75,39525	258,77250	277,09650	205,90900
8 X 8		205,64300	397,70500	435,74450	395,23300
16 X 16		283,96550	446,34450	463,25600	425,20950
32 X 32		308,52700	467,90450	495,34800	473,30750

Sustained Bandwidth



Bandwidth					
Block Size	v0	v1	v2	v3	v4
1 X 1	1,29998	5,65105	22,46860	22,47335	22,47345
2 X 2		22,38510	87,62755	87,83165	87,54600
4 X 4		86,16600	295,74000	316,68150	235,32450
8 X 8		235,02100	454,51950	497,99400	451,69450
16 X 16		324,53250	510,10750	529,43500	485,95350
32 X 32		352,60200	534,74800	566,11250	540,92350

We used a Tesla V100 device for our tests, which has a 731GB/s device to device bandwidth limit. As we optimized our code, we achieved 566GB/s bandwidth usage in our version 3 while using 32x32 block sizes. This means we managed to used 77% of the maximum bandwidth. The same settings also provided the fastest run(6,867 seconds), and the max Gflops rate(495,348).

Our best version was 292 times faster than the serial version. It had x438 times better Gflops and x435 more bandwidth usage than the serial version.

In conclusion, we started with a serial program, and we improved the program execution further in every version. In version 1, we parallelized the program and it enhanced the execution drastically. In version 2, we fused our kernels into one kernel so that kernel launch overhead is eliminated, and program execution became better. In version 3, we introduced temporary variables so that global memory references are eliminated, and memory access times are decreased

drastically. Finally, we introduced shared memory to decrease memory access time for E_prev. Until version 4 every upgrade increased our performance. However in version 4, we couldn’t achieve what we expected. Version 4 performs very similar to the version 3.