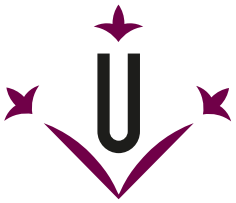
Universitat de Lleida
Màster en Enginyeria Informàtica
High Performance Computing
Prof. Francesc Gine de Sola
Prof. Jordi Ricard Onrubia Palacios



HPC Project

Heat Diffussion Equation

CUDA

Jordi García Ventura
Christian López García

June 8, 2025

Contents

List of codes

List of figures

List of tables

1 Introduction

The purpose of this assignment is to make a reimplementation of the *Heat Diffusion* program using CUDA.

With the nature of CUDA, the program will be divided into the different blocks of the GPU, trying to maximize its thread's usage.

2 Code Implementation

In this section we will comment the changes done to adapt the *C* program to *CUDA*, we will go over function by function after commenting the *CUDA* blocks initialization.

2.1 Initialization

First of all, we parametrized the GPU's *Block X* and *Block Y* to get it from the CLI as in *Code 2*, then proceeded with the initialization of the CUDA grid, blocks, and device memory as in *Code 3*. In this way, we will have a way to test different block sizes easily from the code execution.

Now that we have initialized the dimensions of the grid and blocks, we can start the timer and the calculations. For the timer, we used CUDA events as in *Code 4*, which calculates the elapsed time from the 'start' and 'stop' events in milliseconds. In this way, we're only calculating the runtime of the grid initialization and heat diffusion, and not of the writing the grid to the *bmp* file, but we will comment on it later, since we also parallelized that function.

2.1.1 Function: *initialize_grid*

Once the initializations are done, we will initialize the grid with the initial heat values (active diagonal). As you can see in *Code 5*, the process is simple, from the main function, we run an 'orchestrator' function that is in charge of executing the parallel code and waiting for it. We run a function in the GPU with the block and grid dimensions defined earlier, and inside this function instead of using for loops, we use the running global identifiers i and j, for global row and column index respectively.

2.2 Heat calculations

2.2.1 Function: *solve_heat_equation*

As in *Section 2.1.1* and as we can see in *Code 6*, we call an orchestrator function that for every step manages the calculation of the new heat values, the application of boundary conditions and the swap of grid buffers.

For the heat calculations as we can see in *Code 7*, the code changes were pretty simple compared to previous implementations with MPI or OMP, and even looks simpler than plain C/C++. The code just identifies itself, checks that it's not accessing out of bounds memory, and makes the calculation for its' assigned pixel.

To apply boundary conditions, we make an interesting choice, since the grid is always of size *NxN*, meaning it has same rows as columns, we can only take one of those dimensions, check if it's within the grid X and Y boundaries, and set to zero the top/bottom rows and left/right columns respectively.

2.3 Output

Once the calculations are done, in order to print an output image, we need a way for getting back our results from the device memory into our CPU, but first, we don't really need the heat calculations for writing in the output, so we will make this transmission once the final pixel colors are calculated.

2.3.1 Function: *write_grid*

As commented earlier, we did parallelize this function but did not include it in our time calculations, the reason being is that for the implementation, we needed 'square' sized blocks, otherwise we did not manage to generate a correct image. For this reason, as we can see in *Code 8*, we used a custom block/grid size, independent from the current execution block/grid size, and also allocated a new grid for storing the pixel colors. As we also see in that code, once the pixel data is calculated, we copy it to the CPU and finally write it to the image file.

2.3.2 Function: *prepare_pixel_data*

As commented in the *write_grid* orchestrator, we are calling this *prepare_pixel_data* GPU function. As we can see in *Code 9*, this function fills a grid of size *NxNx3*, with the RGB values of each pixel. For curiosity, this function calls *get_color*, which we marked as *___device___* meaning that it should only be called from a GPU.

3 Code execution

For this assignment we were not running in the moore cluster and, this time, our execution environment had python installed, so we did a simple Makefile as in *Code 10* to compile our program in 'dev' or 'production' (undefining the *CHECK_CUDA_ERRORS* macro for a slight improvement) and the rest was done in python as you can see in *Code 11*, for a saner developer experience.

This time, we did not implement a way of testing our resulting images and relied on faith and manual (visual) checking.

4 Results

As we saw in the code runner *Code 11*, we are executing with the combinations of the following parameters:

- Block Dimension (Thread X and Y, respectively): [(4, 256), (8, 126), (16, 64), (32, 32), (64, 16), (128, 8), (256, 4)]
- Steps count: [100, 1000, 10000, 100000]
- Sizes: [100, 1000, 2000]

We decided to use a non square block size (except for *32x32*) to try it out since in the previous parallel implementations we always tried dividing the grid into squares. Also, since we had a maximum of 1024 threads per block, if we multiply X and Y values it always satisfies that limit.

Before commenting on the comparison of the serial and CUDA execution times, comment that the serial executions were done in the moore cluster for the first assignment, so they are not done in the machine and is not fair, but we will focus in the time growth.

4.1 Speedup

We will do a comparison of the raw execution times between the parallel implementation and the serial counterparts in *Figure 1*, where we can see the execution times of the serial program compared to the mean execution time of CUDA in a logarithmic axis. We conclude that the CUDA program, even though slower in small grid sizes, scales better. In fact, we observe that the CUDA execution times do not change that much for any grid size, we can see it better in *Table 1*, where for the same Steps number, with different sizes, the serial time grows exponentially but the CUDA one stays the same, as with every matrix size and steps.

Size	Steps	CUDA Time	Serial Time
100	100	0.450	0.010
100	1000	4.562	0.110
100	10000	44.394	1.020
100	100000	445.999	10.180
1000	100	0.482	1.340
1000	1000	4.780	12.370
1000	10000	47.734	109.600
1000	100000	476.455	1072.980
2000	100	0.532	5.330
2000	1000	5.273	48.660
2000	10000	52.656	476.730
2000	100000	526.888	4310.410

Table 1: CUDA mean execution times

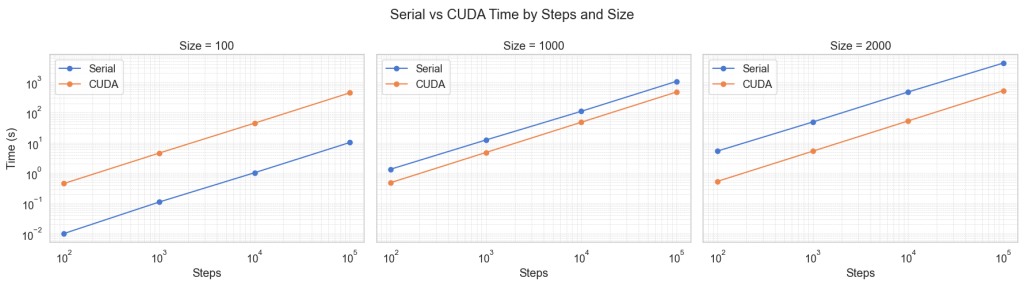


Figure 1: Serial and CUDA comparison of raw times of execution

With these times extracted, we can calculate the speedups of *Figure 2* and *Table 2* with *Formula 1*, ending up having great speedups and with a clear improving tendency, with more speedup with fewer steps to make, which is logical with our implementation, since every step we have to launch two kernels in the GPU and synchronize them.

$$Speedup = \frac{T_{Serial}}{T_{CUDA}} \tag{1}$$

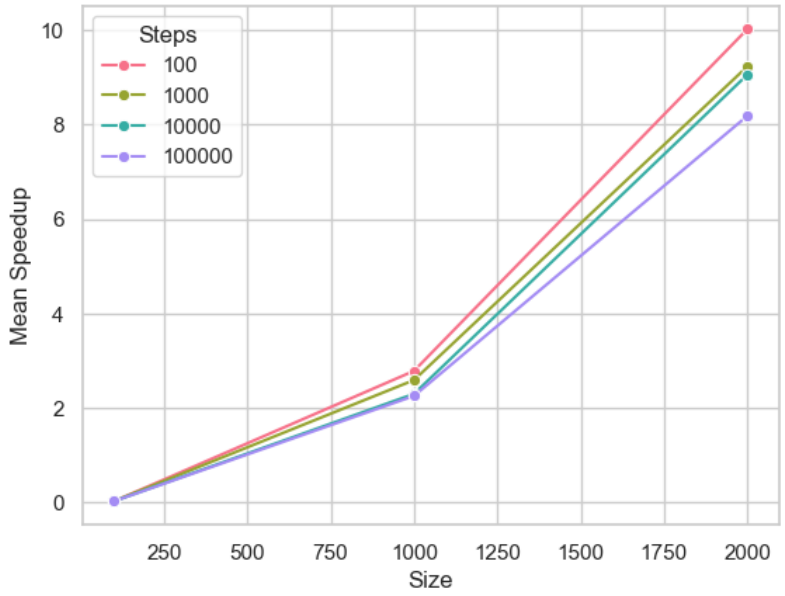


Figure 2: CUDA implementation speedup

Size	Steps	Speedup
100	**100**	**0.022**
100	1000	0.024
100	10000	0.023
100	100000	0.023
1000	100	2.782
1000	1000	2.588
1000	10000	2.296
1000	100000	2.252
2000	100	10.018
2000	1000	9.228
2000	10000	9.054
2000	**100000**	**8.181**

Table 2: CUDA implementation speedup table

4.2 Efficiency

Now that we have speedups, we can calculate efficiency. We could divide our throughput by the installed GPU peak theoretical thoughput, but since we are more interested in the

scalability of the Heat Diffusion problem, we will calculate the *Relative Efficiency* defined as in *Formula 2*. This metric lets us know about how well does our program scale with larger problem sizes, to calculate it, we will compare the speedup with the largest and smallest executed problem size, bold rows in *Table 2*, which is **368.46**, meaning that the largest of the problem sizes that we tested, shows around *400%* more improvement than the smallest.

$$Relative\_Efficiency = \frac{Speedup_{\ Large\_Problem}}{Speedup_{\ Small\_Problem}} \tag{2}$$

4.3 Block size comparison

Lastly, as we executed the program with different block sizes, we will do a comparison of the different most expensive runs (*100000 steps*) as we see in *Figure 3*, there we see that from those executions, the best ones are those with 'taller' dimensions, meaning that the best were those that had a smaller X and bigger Y value. This behavior may be because if we look at *heat_kernel* function, we see that for every pixel we access its vertical and horizontal neighbors, and with this grid dimensions this memory may be already cached, otherwise it does not make much sense.

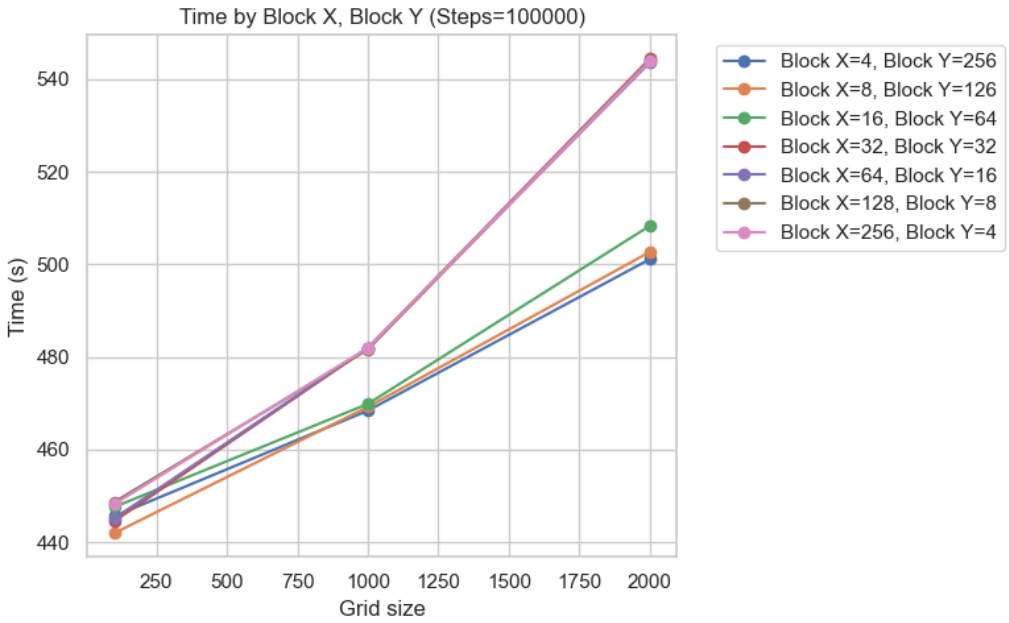


Figure 3: Comparison of time by different block dimension

5 CUDA API instructions

Since we had the opportunity to perform profiling, we decided to check for bottlenecks in our code. With *Code 1*, we execute the *NSight Systems CLI* profiling and from the resulting report we get *Table 3*, where we can see two instructions that clearly are called more than the rest. This behaviour is expected since the biggest part of our program is calculating the heat equation, which is composed basically of multiple calls of *cudaLaunchKernel* and *cudaDeviceSynchronize*.

Before, we assumed that having a lot of kernel launches would have a big impact to the performance, but instead here we can see that the *cudaDeviceSynchronize* instruction is the most impactful, and takes up *98.7*% of the time from the different CUDA API calls, so we should aim to minimize these synchronizations.

Code 1 Command to execute profiling

```
$ nsys profile --stats=true --force-overwrite true -o nsys_report_b32x32_s2000_t100000
↪  ./heat_cuda 32 32 2000 100000 ./results/out_b32x32_s2000_t100000.bmp
```

Time Percentage	Total Time	Num Calls	...	Name
98.7	64384947286	200002	...	cudaDeviceSynchronize
1.2	762839561	200002	...	cudaLaunchKernel
0.1	67209822	3	...	cudaMalloc
0.0	5962123	1	...	cudaMemcpy
0.0	1194703	3	...	cudaFree
0.0	20415	2	...	cudaEventRecord
0.0	14040	2	...	cudaEventCreate
0.0	5699	2	...	cudaEventDestroy
0.0	2602	1	...	cudaEventSynchronize
0.0	1627	1	...	cuModuleGetLoadingMode

Table 3: CUDA API called instructions profiling

6 About GPU programming

These results have shown that the Heat Diffusion Equation is a highly scalable problem and is a good fit for GPU programming.

Although this has not been our first experience with GPU programming since we had done some tutorials on *GLSL* and shaders in Unity, this project has been great to settle that knowledge outside the game engines field and to being able to directly measure the impact it has in our code execution time.

From a developer experience point of view, once you get used to not having for loops and visualize how the program distributes its threads, the code feels cleaner and better to work with, which did not happen with other parallel APIs that we worked with before. Although OMP did not need many changes to the code, it did not make much improvements to its execution. In the other hand, MPI offered more improvement with the downside of needing to make a lot of changes to the code. CUDA feels like the best of both mentioned APIs, since it offered even better results than MPI by changing a minimal amount of code.

7 Code changes

Code 2 Main function CLI arguments

```
int main(int argc, char *argv[])
{
    if (argc != 6)
    {
        printf("Usage: %s <block_x> <block_y> <size> <steps> <name_output_file>\n",
        ↪  argv[0]);
        return 1;
    }

    double r = ALPHA * DT / (DX * DY);
    unsigned int arg = 1;
    int nx, ny, steps, block_x, block_y;
    char *filename;
    float ms = 0;

    // EXECUTION CLI PARAMS
    block_x     = atoi(argv[arg++]),
    block_y     = atoi(argv[arg++]),
    nx = ny     = atoi(argv[arg++]),
    steps       = atoi(argv[arg++]),
    filename    = argv[arg++];

    ...
}
```

Code 3 CUDA grid and data initialization

```
#define CHECK_CUDA_ERRORS                                          \
    cudaError_t err = cudaGetLastError();                          \
    if (err != cudaSuccess) {                                      \
        printf("CUDA Error: %s\n", cudaGetErrorString(err));       \
    }                                                              \

int main(int argc, char *argv[])
{
    ...

    double *d_grid, *d_new_grid;
    cudaMalloc(&d_grid, nx * ny * sizeof(double));
    cudaMalloc(&d_new_grid, nx * ny * sizeof(double));

    dim3 blockDim(block_x, block_y);
    dim3 gridDim(
        (ny + block_x - 1) / block_x,
        (nx + block_y - 1) / block_y
    );

    CHECK_CUDA_ERRORS;

    ...
}
```

Code 4 CUDA event time calculation

```
int main(int argc, char *argv[])
{
    ... Initialization

    // TIMING :: START
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    initialize_grid(...);
    solve_heat_equation(...);

    // TIMING :: END
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&ms, start, stop);

    ... Writing the grid

    printf("Execution Time = %.3fms for %dx%d grid and %d steps\n", ms, nx, ny,
    ↪  steps);

    // FREE
    cudaFree(d_grid);
    cudaFree(d_new_grid);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

Code 5 Initialize grid

```
__global__
void initialize_grid_kernel(
    double *grid,
    int nx,
    int ny
)
{
    // Initialize the grid diagonals to be heat sources.
    //
    // Steps:
    //  1. Identify executing code with i/j.
    //  2. Initialize the diagonal heat grid, excluding the matrix borders.

    int i = blockIdx.y * blockDim.y + threadIdx.y + 1,
        j = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure safe CUDA memory access
    if (i > nx || j >= ny) return;

    int is_edge     = i == 0 || i == nx - 1 || j == 0 || j == ny - 1,
        is_diagonal = i == j || i == nx - j - 1;

    grid[i * ny + j] = !is_edge && is_diagonal ? T : 0.0;
}

void initialize_grid(
    double *grid,
    int nx,
    int ny,
    const dim3 gridDim,
    const dim3 blockDim
)
{
    initialize_grid_kernel<<<gridDim, blockDim>>>(grid, nx, ny);
    cudaDeviceSynchronize();
}
```

Code 6 Heat diffusion orchestrator

```
void solve_heat_equation(
    double *d_grid,
    double *d_new_grid,
    int nx,
    int ny,
    int steps,
    double r,
    const dim3 gridDim,
    const dim3 blockDim
)
{
    for (int step = 0; step < steps; step++)
    {
        heat_kernel<<<gridDim, blockDim>>>(d_grid, d_new_grid, nx, ny, r);
        cudaDeviceSynchronize();
        apply_boundary_conditions<<<gridDim, blockDim>>>(d_grid, nx, ny);
        cudaDeviceSynchronize();

        double *temp = d_grid;
        d_grid = d_new_grid;
        d_new_grid = temp;
    }
}
```

Code 7 Heat diffusion calculation and boundary conditions

```
__global__
void heat_kernel(
    double* grid,
    double* new_grid,
    int nx,
    int ny,
    double r
) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i <= 0 || i >= nx - 1 || j <= 0 || j >= ny - 1) return;

    int idx = i * ny + j;

    new_grid[idx] = grid[idx]
        + r * (grid[idx + ny] + grid[idx - ny] - 2.0 * grid[idx])  // vertical
        + r * (grid[idx + 1] + grid[idx - 1] - 2.0 * grid[idx]);   // horizontal
}

__global__
void apply_boundary_conditions(
    double* grid,
    int nx,
    int ny
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < ny) {
        grid[0 * ny + idx] = 0.0;
        grid[(nx - 1) * ny + idx] = 0.0;
    }

    if (idx < nx) {
        grid[idx * ny + 0] = 0.0;
        grid[idx * ny + (ny - 1)] = 0.0;
    }
}
```

Code 8 Parallel pixel color calculation orchestrator

```
void write_grid(
    FILE* file,
    double* d_grid,
    int nx,
    int ny
) {
    int row_stride = ny * 3;
    int padding = (4 - (row_stride % 4)) % 4;
    int padded_row_size = row_stride + padding;
    int total_size = nx * padded_row_size;

    // Allocate output buffer on GPU
    unsigned char* d_pixel_data;
    cudaMalloc(&d_pixel_data, total_size);

    dim3 writeBlockDim(16, 16);
    dim3 writeGridDim(
        (ny + writeBlockDim.x - 1) / writeBlockDim.x,
        (nx + writeBlockDim.y - 1) / writeBlockDim.y
    );

    CHECK_CUDA_ERRORS;

    prepare_pixel_data<<<writeGridDim, writeBlockDim>>>(d_grid, d_pixel_data, nx, ny,
    ↪  padded_row_size, padding);
    cudaDeviceSynchronize();

    // Copy back to host
    unsigned char* h_pixel_data = (unsigned char*)malloc(total_size);
    cudaMemcpy(h_pixel_data, d_pixel_data, total_size, cudaMemcpyDeviceToHost);

    fwrite(h_pixel_data, 1, total_size, file);

    cudaFree(d_pixel_data);
    free(h_pixel_data);
}
```

Code 9 Parallel pixel color calculation

```
__device__ // GPU only function !!
void get_color(
    double value,
    unsigned char *r,
    unsigned char *g,
    unsigned char *b
)
{
...
}

__global__
void prepare_pixel_data(
    double* grid,
    unsigned char* pixel_data,
    int nx,
    int ny,
    int padded_row_size,
    int padding
)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y,
        j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i >= nx || j >= ny) return;

    int row_index           = nx - 1 - i,
        row_offset          = i * padded_row_size,
        pixel_offset        = row_offset + j * 3,
        pixel_row_offset    = row_index * padded_row_size;


    unsigned char r, g, b;
    get_color(grid[row_index * ny + j], &r, &g, &b);

    pixel_data[pixel_offset + 0] = b;
    pixel_data[pixel_offset + 1] = g;
    pixel_data[pixel_offset + 2] = r;

    // Only one thread per row handles padding
    if (j == 0 && padding > 0) {
        for (int p = 0; p < padding; ++p) {
            pixel_data[pixel_row_offset + nx * 3 + p] = 0;
        }
    }
}
```

8 Makefile and Python runner

Code 10 Makefile

```
compile:
	nvcc -o heat_cuda heat.cu

compile_prod:
	nvcc -U CHECK_CUDA_ERRORS heat.cu -o heat_cuda

sample: compile
	./heat_cuda 2000 10000 sample.bmp

all: clear compile_prod
	python run.py -r ./results.csv

clear:
	rm -fr ./results/* results.csv
```

Code 11 Python runner

```
# ...imports and helpers

def main() -> None:
    block_dim = [(4, 256), (8, 126), (16, 64), (32, 32), (64, 16), (128, 8), (256, 4)]
    steps_dim = [100, 1_000, 10_000, 100_000]
    size_dim = [100, 1_000, 2_000]
    combinations = list(product(block_dim, steps_dim, size_dim))

    with open(args.results, "w", newline="") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["Block Dim", "Size", "Steps", "Time"])
        for (block_x, block_y), steps, size in combinations:
            output_filename =
             ↪ f"./results/out_b{block_x}x{block_y}_s{size}_t{steps}.bmp"

            command = [
                "./heat_cuda",
                str(block_x),
                str(block_y),
                str(size),
                str(steps),
                output_filename,
            ]

            _print(f"Running: {' '.join(command)}")
            try:
                result = subprocess.run(
                    command, capture_output=True, text=True, check=True
                )

                _print(result.stdout)
                time = extract_time(result.stdout)
                _print(f"{time}\n")

                writer.writerow([block_x, block_y, size, steps, time])
            except subprocess.CalledProcessError as e:
                print(f"Error during execution: {e}")
                exit(1)
# ... main call
```

9 Plain results

Block X	Block Y	Size	Steps	Time
4	256	100	100	458.301
4	256	1000	100	474.464
4	256	2000	100	505.127
4	256	100	1000	4546.784
4	256	1000	1000	4706.944
4	256	2000	1000	5014.381
4	256	100	10000	45698.949
4	256	1000	10000	47047.078
4	256	2000	10000	49992.352
4	256	100	100000	445655.594
4	256	1000	100000	468449.812
4	256	2000	100000	501137.031
8	126	100	100	421.156
8	126	1000	100	475.849
8	126	2000	100	511.239
8	126	100	1000	4582.521
8	126	1000	1000	4710.59
8	126	2000	1000	5033.29
8	126	100	10000	40571.789
8	126	1000	10000	47047.141
8	126	2000	10000	50255.668
8	126	100	100000	441985.531
8	126	1000	100000	469245.438
8	126	2000	100000	502696.188
16	64	100	100	462.765
16	64	1000	100	477.476

16	64	2000	100	514.465
16	64	100	1000	4516.707
16	64	1000	1000	4722.947
16	64	2000	1000	5081.395
16	64	100	10000	45720.352
16	64	1000	10000	47234.938
16	64	2000	10000	50670.863
16	64	100	100000	447631.062
16	64	1000	100000	469904.469
16	64	2000	100000	508339.656
32	32	100	100	461.52
32	32	1000	100	483.703
32	32	2000	100	549.235
32	32	100	1000	4583.285
32	32	1000	1000	4828.854
32	32	2000	1000	5450.304
32	32	100	10000	45743.98
32	32	1000	10000	48270.859
32	32	2000	10000	54483.594
32	32	100	100000	444581.562
32	32	1000	100000	481960.656
32	32	2000	100000	544399.0
64	16	100	100	461.066
64	16	1000	100	484.503
64	16	2000	100	546.084
64	16	100	1000	4581.335
64	16	1000	1000	4831.66
64	16	2000	1000	5443.014

64	16	100	10000	41647.891
64	16	1000	10000	48267.117
64	16	2000	10000	54364.484
64	16	100	100000	445172.906
64	16	1000	100000	481990.406
64	16	2000	100000	543602.938
128	8	100	100	425.976
128	8	1000	100	488.536
128	8	2000	100	547.101
128	8	100	1000	4578.743
128	8	1000	1000	4832.119
128	8	2000	1000	5444.092
128	8	100	10000	45650.367
128	8	1000	10000	47987.488
128	8	2000	10000	54456.543
128	8	100	100000	448711.312
128	8	1000	100000	481643.312
128	8	2000	100000	544161.312
256	4	100	100	461.954
256	4	1000	100	487.337
256	4	2000	100	551.132
256	4	100	1000	4541.861
256	4	1000	1000	4829.228
256	4	2000	1000	5444.631
256	4	100	10000	45727.109
256	4	1000	10000	48286.688
256	4	2000	10000	54371.23
256	4	100	100000	448256.438

256	4	1000	100000	481991.188
256	4	2000	100000	543877.25