CMPUT 481 Assignment 1

Brock Toews

October 2, 2013

btoews
1284088

List of Figures

1 Chart of speedups 5
2 Chart of times to multiply matrices 5
3 Chart of times to complete segments 6

Contents

1 Introduction **2**

2 Implementation **2**
2.1 Description 2
2.2 Correctness 2
2.3 Performance Considerations 3

3 Analysis of Results **3**
3.1 Dataset Decisions 3
3.2 Performance Gains 3

4 Conclusion **4**

5 Source Code **6**
5.1 Parallel Implementation 6
5.2 Sequential Implementation 11
5.3 Analysis Script 13
5.4 Gnuplot Script 14
5.5 Makefile 15
5.6 Raw Data 16

1 Introduction

The purpose of this paper is to analyze the speedups gained by parallelizing a simple widely spread algorithm: that of dense matrix multiplication. As one will be able to see from what follows, the potential speedup of using multiple CPU cores is quite great.

2 Implementation

2.1 Description

My sequential implementation is a simple, standard matrix multiplier. It was written in C, and performs the following steps. First, it allocates three single dimensional arrays: the two origin arrays, and the product array (this step is shown as the *Initalize* step in Figure 3). Then, it walks through the two origin arrays, assigning random long integers to each index (the *Generate* step in Figure 3). Now, it walks through both the origin arrays multiplying the proper indices together, storing the results in an array of its own. Once an array has been completed, it sums it together the array and assigns the result to the proper index of the product array. This is the *Multiply* step in Figure 3.

The parallelized version is a fairly intuitive adaptation of the above implementation. Its initialization step is the same as the sequential version's, except for one added component; it also generates a list of structs storing the beginning and end of segments, where each of these segments are computed by one thread. The generation step is performed in the same manner, except that each segment is operated upon by a separate thread. After the generation operations are complete, the threads are closed. And, just like the generation step, the multiplication step is broken up into threads and closed upon completion, with each thread operating upon one segment each.

2.2 Correctness

The correctness of the values generated by both implementation have not been experimentally verified, as this is not exceedingly important to the experiment. Instead, it has been checked that both implementations perform the correct number of operations. It is also fairly certain that the resultant values must be correct, from much reading of the code.

2.3 Performance Considerations

There are a few points regarding performance of the implementation that are worth looking at. Firstly, one will immediately notice that between the generation and multiplication steps, I close and reopen all of the threads, while better performance could certainly be had by eliminated this sequential part of the code. there are a couple reasons that this was not done. One, it is somewhat easier to time the phases as a whole by returning to the main function. Secondly, It as something of an arbitrary decision, as it seemed easier to write at the time.

Another place where performance is in question is whether I make the program cooperate with the machine's cache. In answer to this, yes. The nested for-loops are functionally equivalent to looping through the single dimensional arrays with one loop, without making any large jumps. As such, most of the matrix accesses should be cache hits.

3 Analysis of Results

3.1 Dataset Decisions

It is, of course, quite necessary to establish what kind of data was used in the execution of the experiment. There were three sizes of matrix that were considered: 1024x1024, 2048x2048, and 4096x4096. Each of these sizes were run in the sequential version, and in the parallel version with 2 through 8 threads. Finally, for each set of parameters, the test was run 10 times, throwing out the first 5.

It should be noted that 512x512 tests were also run, but the results were not considered for various reasons. While the results were interesting and unique from the other tests, they were thrown out for various reasons, among those reasons, the small amount of time it took to run the tests (and the inherent unreliablility of short times), and the simple fact that it was inconvenient to make the charts easily readable when there was the one set of data with values so tiny.

Finally, the hardware that the tests were run on is important. For the experiment, I used a laptop with an Intel Core i7 running 4 (hyperthreaded) cores.

3.2 Performance Gains

Now, after much adeiu, we may discuss the numbers of the experiment. The first item of significance, is the speedup (Figure 1). once can see up to four cores, one almost gets linear speedup. This, on its own, very much demonstrates the advantage of parallelization.

An interesting note of this that one will definitely notice, is that after four threads (5-8), the speedup levels off. This is quite likely a result of the i7's use of hyperthreading to create logical threads.

Unsuprisingly, as one can see in Figure 3, most time of the computation is spent in the multiplication stage. This stage take less and less time as more cores are added.

What is more suprising, is that the generation phase takes more time as more cores are added. What the reason for this is, is unclear. It is not a sequential section, nor is the amount of work there is to do dependent on the number of threads. The only reasoning that comes to mind is that the compiler has an easier time optimizing the, admittedly, simple generation code when it is sequential.

4 Conclusion

As one can see from Figures 1 and 2, and above discussion, using multiple threads to perform easily parallelizable operations provides a significant performance advantage over simply running the same code sequentially, in a single thread.

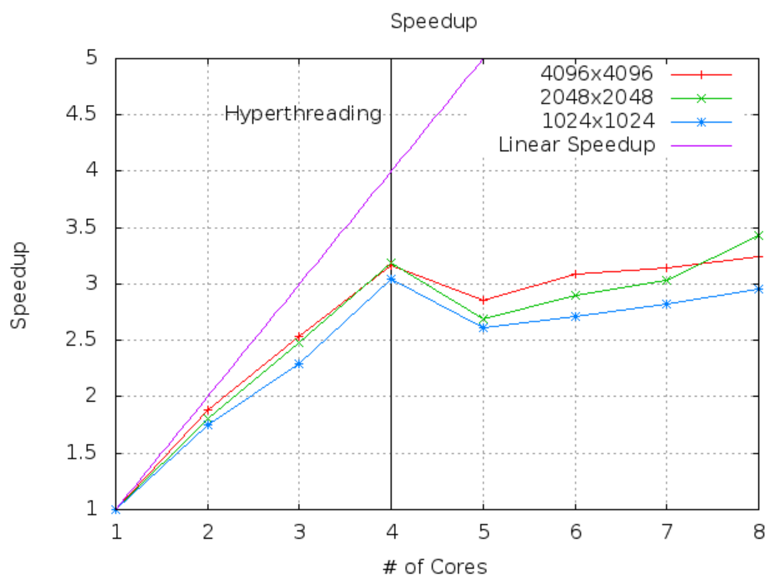

Figure 1: Chart of speedups

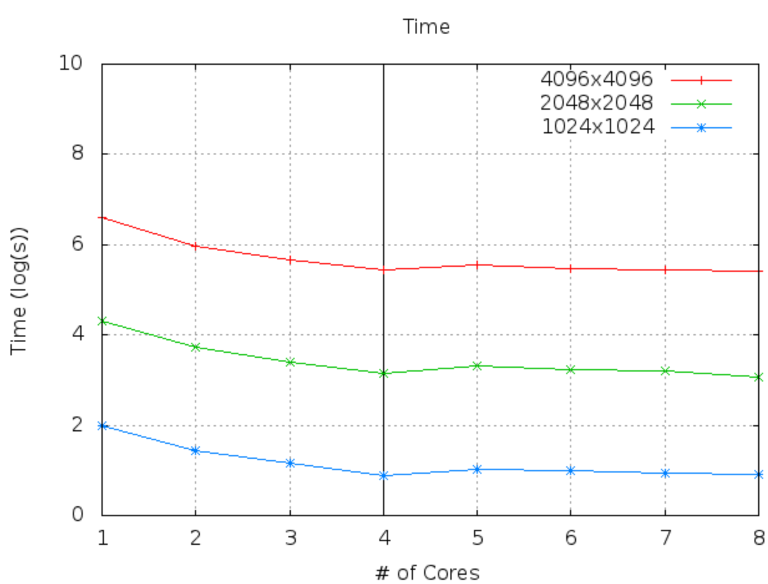

Figure 2: Chart of times to multiply matrices

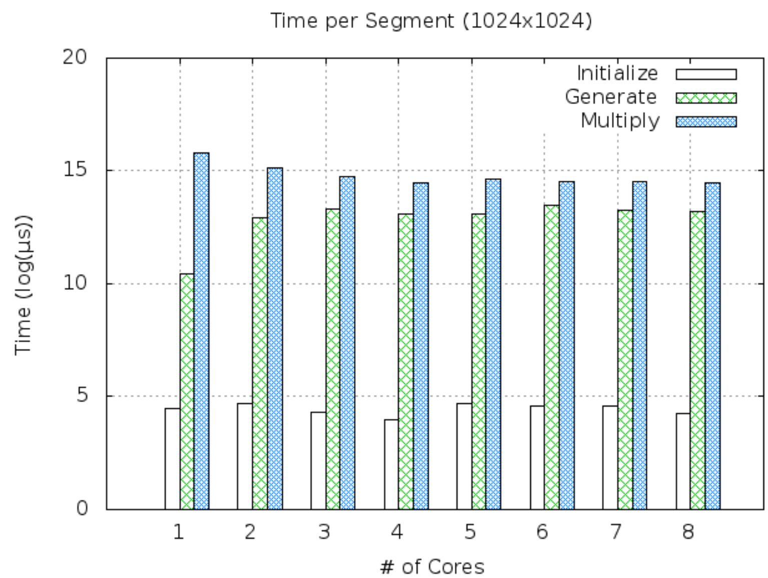

Figure 3: Chart of times to complete segments

5 Source Code

5.1 Parallel Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <sys/time.h>

static void * gen_seg(void *);
static void * mult_seg(void *);

static pthread_barrier_t bar_gen;
static pthread_barrier_t bar_mult;
static int nproc;
static int mat_size;
static long * mat1;
static long * mat2;
static long * mat_fin;
```

```
struct seg_desc {
        int start;
        int end;
};

int main(int argc, char * argv[]) {
        struct seg_desc * segs;
        struct timeval start_fin;
        struct timeval end_fin;
        struct timeval init_fin;
        struct timeval gen_fin;
        int init_sec;
        int init_usec;
        double init_time;
        int gen_sec;
        int gen_usec;
        double gen_time;
        int mult_sec;
        int mult_usec;
        double mult_time;
        int tot_sec;
        int tot_usec;
        double tot_time;
        int i;
        int seg_size;
        pthread_t * thread_ids;

        /*start timing*/
        gettimeofday(&start_fin, NULL);

        /*handle user input*/
        if (argc == 2) {
                nproc = strtol(argv[1], NULL, 10);
        } else if (argc == 3) {
                nproc = strtol(argv[1], NULL, 10);
                mat_size = strtol(argv[2], NULL, 10);
        } else {
                nproc = get_nprocs();
                mat_size = 1024;
```

```
	}

	segs = malloc(sizeof(struct seg_desc)*nproc);
	thread_ids = malloc(sizeof(pthread_t)*nproc);
	mat1 = malloc(sizeof(long)*mat_size*mat_size);
	mat2 = malloc(sizeof(long)*mat_size*mat_size);
	mat_fin = malloc(sizeof(long)*mat_size*mat_size);

	pthread_setconcurrency(nproc);

	/*Choose row assignments*/
	seg_size = mat_size/nproc;
	for (i = 0; i < nproc; i ++) {
		segs[i].start = i*seg_size;
		segs[i].end = (i + 1)*seg_size;
		if (i == nproc - 1) {
			segs[i].end += mat_size % nproc;
		}
	}

	gettimeofday(&init_fin, NULL);

	/*Generate matrices*/
	for (i = 0; i < nproc; i ++) {
		pthread_create(&thread_ids[i], NULL, &gen_seg, &segs[i]);
	}

	for (i = 0; i < nproc; i ++) {
		pthread_join(thread_ids[i], NULL);
	}

	gettimeofday(&gen_fin, NULL);

	/*Multiply Matrices*/
	for (i = 0; i < nproc; i ++) {
		pthread_create(&thread_ids[i], NULL, &mult_seg, &segs[i]);
	}

	for (i = 0; i < nproc; i ++) {
		pthread_join(thread_ids[i], NULL);
```

```
        }

        /*end timing*/
        gettimeofday(&end_fin, NULL);

        tot_sec = (int)end_fin.tv_sec - (int)start_fin.tv_sec;
        tot_usec = (int)end_fin.tv_usec - (int)start_fin.tv_usec;
        tot_time = (double)tot_sec + ((double)tot_usec/1000000.0);

        init_sec = (int)init_fin.tv_sec - (int)start_fin.tv_sec;
        init_usec = (int)init_fin.tv_usec - (int)start_fin.tv_usec;
        init_time = (double)init_sec + ((double)init_usec/1000000.0);

        gen_sec = (int)gen_fin.tv_sec - (int)init_fin.tv_sec;
        gen_usec = (int)gen_fin.tv_usec - (int)init_fin.tv_usec;
        gen_time = (double)gen_sec + ((double)gen_usec/1000000.0);

        mult_sec = (int)end_fin.tv_sec - (int)gen_fin.tv_sec;
        mult_usec = (int)end_fin.tv_usec - (int)gen_fin.tv_usec;
        mult_time = (double)mult_sec + ((double)mult_usec/1000000.0);

        printf("%lf %lf %lf %lf %d %d\n", tot_time, init_time, gen_time,
        mult_time, nproc, mat_size);

        return 0;
}

/*generates a given segment of a matrix*/
static void * gen_seg(void * arg) {
        int row;
        int col;
        struct seg_desc seg = *((struct seg_desc*)arg);

        srandom(time(0));

        for (row = seg.start; row < seg.end; row++) {
                for (col = 0; col < mat_size; col++) {
                        mat1[col + row*mat_size] = random();
                        mat2[col + row*mat_size] = random();
                }
```

```
        }

        pthread_exit(0);
}

/*multiples a given segment of mat1 to the corresponding segment of mat2
 *and places the result in mat_fin
 */
static void * mult_seg(void * arg) {
        int row;
        int col;
        int cell;
        struct seg_desc seg = *((struct seg_desc*)arg);

        for (row = seg.start; row < seg.end; row ++) {
                for (col = 0; col < mat_size; col ++) {
                        int res_ind = row*mat_size + col;
                        mat_fin[res_ind] = 0;
                        for (cell = 0; cell < mat_size; cell ++) {
                                int ind1 = row*mat_size + cell;
                                int ind2 = cell*mat_size + col;
                                long prod = mat1[ind1] * mat2[ind2];
                                mat_fin[res_ind] += prod;
                        }
                }
        }

        pthread_exit(0);
}
```

5.2 Sequential Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>
#include <sys/time.h>

int main(int argc, char * argv[]) {
        int row;
        int col;
        int cell;
        int mat_size;
        long * mat1;
        long * mat2;
        long * mat_fin;
        struct timeval start_fin;
        struct timeval init_fin;
        struct timeval gen_fin;
        struct timeval end_fin;
        int init_sec;
        int init_usec;
        double init_time;
        int gen_sec;
        int gen_usec;
        double gen_time;
        int mult_sec;
        int mult_usec;
        double mult_time;
        int tot_sec;
        int tot_usec;
        double tot_time;

        /*start timing*/
        gettimeofday(&start_fin, NULL);

        if (argc == 2) {
                mat_size = strtol(argv[1], NULL, 10);
        } else {
                mat_size = 1024;
        }
```

```
	mat1 = malloc(sizeof(long)*mat_size*mat_size);
	mat2 = malloc(sizeof(long)*mat_size*mat_size);
	mat_fin = malloc(sizeof(long)*mat_size*mat_size);

	/*Generate matrices*/
	srandom(time(0));

	gettimeofday(&init_fin, NULL);

	for (row = 0; row < mat_size; row++) {
		for (col = 0; col < mat_size; col++) {
			mat1[col + row*mat_size] = random();
			mat2[col + row*mat_size] = random();
		}
	}

	gettimeofday(&gen_fin, NULL);

	/*Multiply Matrices*/
	for (row = 0; row < mat_size; row++) {
		for (col = 0; col < mat_size; col++) {
			int res_ind = row*mat_size + col;
			mat_fin[res_ind] = 0;
			for (cell = 0; cell < mat_size; cell++) {
				int ind1 = row*mat_size + cell;
				int ind2 = cell*mat_size + col;
				long prod = mat1[ind1] * mat2[ind2];
				mat_fin[res_ind] += prod;
			}
		}
	}

	/*end timing*/
	gettimeofday(&end_fin, NULL);

	tot_sec = end_fin.tv_sec - start_fin.tv_sec;
	tot_usec = end_fin.tv_usec - start_fin.tv_usec;
	tot_time = (double)tot_sec + ((double)tot_usec/1000000.0);

```

```c
        init_sec = init_fin.tv_sec - start_fin.tv_sec;
        init_usec = init_fin.tv_usec - start_fin.tv_usec;
        init_time = (double)init_sec + ((double)init_usec/1000000.0);

        gen_sec = gen_fin.tv_sec - init_fin.tv_sec;
        gen_usec = gen_fin.tv_usec - init_fin.tv_usec;
        gen_time = (double)gen_sec + ((double)gen_usec/1000000.0);

        mult_sec = end_fin.tv_sec - gen_fin.tv_sec;
        mult_usec = end_fin.tv_usec - gen_fin.tv_usec;
        mult_time = (double)mult_sec + ((double)mult_usec/1000000.0);

        printf("%lf %lf %lf %lf %d %d\n", tot_time, init_time, gen_time,
        mult_time, 1, mat_size);

        return 0;
}
```

5.3 Analysis Script

```python
data_file = open("output.dat")
total_runs = 5
total_sizes = 4
data_points = []

for cores in range(1,9):
    for size in range(total_sizes):
        values = None
        tot_avg = 0
        init_avg = 0
        gen_avg = 0
        mult_avg= 0
        for runs in range(total_runs):
            line = data_file.readline()
            if len(line) > 0:
                values = line.split()
                values[0] = float(values[0])
                values[1] = float(values[1])
                values[2] = float(values[2])
                values[3] = float(values[3])
```

```
                    values[4] = int(values[4])
                    values[5] = int(values[5])
                    tot_avg += values[0]
                    init_avg += values[1]
                    gen_avg += values[2]
                    mult_avg += values[3]
            tot_avg /= total_runs
            init_avg /= total_runs
            gen_avg /= total_runs
            mult_avg /= total_runs
            data_string = "{} {} {} {} {} {}\n".format(tot_avg, init_avg,
                gen_avg, mult_avg, values[4], values[5])
            data_points.append(data_string)

output_file = open("average.dat", 'w')
for item in data_points:
    output_file.writelines(item)
```

5.4 Gnuplot Script

```
set grid
set datafile missing '@'

set autoscale

stats "average.dat" u 1 every 4::3 name 'A'
stats "average.dat" u 1 every 4::2 name 'B'
stats "average.dat" u 1 every 4::1 name 'C'
stats "average.dat" u 1 every 4::0 name 'D'

set term png

set arrow from 4,1 to 4,5 nohead
set label "Hyperthreading" at 2.2,4.5
set output "speedup.png"
set xlabel '# of Cores'
set ylabel 'Speedup'
set title 'Speedup'
set xrange [1:8]
set yrange [1:5]
```

```
plot "average.dat" u 5:(A_max/$1) every 4::3 ti '4096x4096' w lp, \
'' u 5:(B_max/$1) every 4::2 ti '2048x2048' w lp, \
'' u 5:(C_max/$1) every 4::1 ti '1024x1024' w lp, \
x ti "Linear Speedup"
#'' u 5:(D_max/$1) every 4::0 ti '512x512' w lp, \

unset label
unset arrow
set arrow from 4,0 to 4,10 nohead
set label "Hyperthreading" at 2.2,70
set output "time.png"
set xlabel '# of Cores'
set ylabel 'Time (log(s))'
set title 'Time'
set xrange [1:8]
set yrange [0:10]
plot "average.dat" u 5:(log($1)) every 4::3 ti '4096x4096' w lp, \
'' u 5:(log($1)) every 4::2 ti '2048x2048' w lp, \
'' u 5:(log($1)) every 4::1 ti '1024x1024' w lp
#'' u 5:(log($1)) every 4::0 ti '512x512' w lp

unset arrow
unset label
set title "Time per Segment (1024x1024)"
set ylabel "Time (log(  s ))"
set output "segment.png"
set style data histogram
set style histogram cluster gap 2
set style fill pattern border rgb "black"
set auto x
set yrange [0:20]
plot "average.dat" u (log($2*1000000)):xtic(5) every 4::1 ti "Initialize",
'' u (log($3*1000000)):xtic(5) every 4::1 ti "Generate", \
'' u (log($4*1000000)):xtic(5) every 4::1 ti "Multiply"
```

5.5 Makefile

```
main: main.c
	gcc main.c -O4 -pthread -o main

```

```
seq: seq.c
	gcc seq.c -O4 -o seq

test:
	bash test_scr

report: report.tex
	texi2pdf report.tex
	open report.pdf
```

5.6 Raw Data

```
#total_s init_s gen_s mult_s cores mat_size
0.206786 0.000101 0.010790 0.195895 1 512
0.250920 0.000101 0.012926 0.237893 1 512
0.213285 0.000103 0.012881 0.200301 1 512
0.202880 0.000102 0.007839 0.194939 1 512
0.240743 0.000105 0.007472 0.233166 1 512
7.426597 0.000102 0.032204 7.394291 1 1024
7.330393 0.000103 0.033901 7.296389 1 1024
7.258880 0.000101 0.032754 7.226025 1 1024
7.227244 0.000038 0.025834 7.201372 1 1024
7.399402 0.000102 0.043635 7.355665 1 1024
73.794940 0.000102 0.083058 73.711780 1 2048
74.965592 0.000037 0.070259 74.895296 1 2048
74.698588 0.000099 0.084116 74.614373 1 2048
75.718912 0.000101 0.088560 75.630251 1 2048
73.576071 0.000100 0.084736 73.491235 1 2048
712.892804 0.000101 0.298345 712.594358 1 4096
723.002421 0.000103 0.292367 722.709951 1 4096
709.304318 0.000102 0.292710 709.011506 1 4096
715.163671 0.000102 0.289810 714.873759 1 4096
781.544611 0.000102 0.293260 781.251249 1 4096
0.265660 0.000040 0.104207 0.161413 2 512
0.263157 0.000111 0.102774 0.160272 2 512
0.260584 0.000111 0.099884 0.160589 2 512
0.261635 0.000109 0.100224 0.161302 2 512
0.263681 0.000040 0.102161 0.161480 2 512
4.238569 0.000110 0.420336 3.818123 2 1024
4.169859 0.000111 0.415479 3.754269 2 1024
```

29	4.116364	0.000109	0.397142	3.719113	2	1024
30	4.166607	0.000111	0.407713	3.758783	2	1024
31	4.173396	0.000110	0.411045	3.762241	2	1024
32	40.962820	0.000110	1.418113	39.544597	2	2048
33	40.737116	0.000111	1.583755	39.153250	2	2048
34	41.749754	0.000109	1.659357	40.090288	2	2048
35	41.819841	0.000110	1.723938	40.095793	2	2048
36	41.207431	0.000111	1.670994	39.536326	2	2048
37	386.320689	0.000039	6.952247	379.368403	2	4096
38	386.293981	0.000110	6.691557	379.602314	2	4096
39	388.180284	0.000110	6.676509	381.503665	2	4096
40	387.757712	0.000105	6.859361	380.898246	2	4096
41	388.044479	0.000105	6.718013	381.326361	2	4096
42	0.257060	0.000037	0.146598	0.110425	3	512
43	0.233961	0.000038	0.123695	0.110228	3	512
44	0.254045	0.000109	0.143045	0.110891	3	512
45	0.260513	0.000040	0.149037	0.111436	3	512
46	0.262635	0.000040	0.151717	0.110878	3	512
47	3.223560	0.000110	0.620879	2.602571	3	1024
48	3.220371	0.000146	0.607273	2.612952	3	1024
49	3.162072	0.000041	0.613252	2.548779	3	1024
50	3.160939	0.000040	0.609508	2.551391	3	1024
51	3.191903	0.000038	0.624855	2.567010	3	1024
52	32.182240	0.000113	5.151504	27.030623	3	2048
53	29.443448	0.000041	2.407528	27.035879	3	2048
54	29.546961	0.000110	2.450511	27.096340	3	2048
55	29.168509	0.000039	2.486500	26.681970	3	2048
56	29.649332	0.000107	2.534939	27.114286	3	2048
57	283.171371	0.000040	9.731733	273.439598	3	4096
58	301.907963	0.000039	27.864411	274.043513	3	4096
59	289.203954	0.000074	9.832229	279.371651	3	4096
60	282.260252	0.000110	9.543278	272.716864	3	4096
61	279.206985	0.000134	9.302923	269.903928	3	4096
62	0.206815	0.000041	0.120095	0.086679	4	512
63	0.202058	0.000041	0.119080	0.082937	4	512
64	0.211365	0.000041	0.123148	0.088176	4	512
65	0.195435	0.000041	0.112486	0.082908	4	512
66	0.205773	0.000042	0.122841	0.082890	4	512
67	2.392387	0.000041	0.490974	1.901372	4	1024
68	2.477509	0.000040	0.545347	1.932122	4	1024

69	2.404062	0.000039	0.474649	1.929374	4	1024
70	2.381462	0.000110	0.466269	1.915083	4	1024
71	2.389322	0.000039	0.481249	1.908034	4	1024
72	22.734584	0.000112	1.937486	20.796986	4	2048
73	22.769472	0.000068	1.928305	20.841099	4	2048
74	22.713619	0.000109	1.964738	20.748772	4	2048
75	22.649657	0.000111	1.908269	20.741277	4	2048
76	26.199102	0.000039	5.503073	20.695990	4	2048
77	229.284079	0.000041	7.817896	221.466142	4	4096
78	227.621998	0.000107	7.715440	219.906451	4	4096
79	228.159853	0.000154	8.333649	219.826050	4	4096
80	227.580547	0.000101	7.667025	219.913421	4	4096
81	237.232761	0.000111	21.587070	215.645580	4	4096
82	0.261908	0.000110	0.118469	0.143329	5	512
83	0.268515	0.000111	0.121381	0.147023	5	512
84	0.251460	0.000111	0.112747	0.138602	5	512
85	0.251038	0.000113	0.114532	0.136393	5	512
86	0.270436	0.000040	0.115900	0.154496	5	512
87	2.806237	0.000110	0.468481	2.337646	5	1024
88	2.692321	0.000110	0.388885	2.303326	5	1024
89	2.917014	0.000111	0.600936	2.315967	5	1024
90	2.794352	0.000111	0.479946	2.314295	5	1024
91	2.794056	0.000110	0.459088	2.334858	5	1024
92	30.151571	0.000109	5.218029	24.933433	5	2048
93	27.393440	0.000112	1.843805	25.549523	5	2048
94	26.959250	0.000111	1.863001	25.096138	5	2048
95	26.710109	0.000110	1.693159	25.016840	5	2048
96	27.257437	0.000111	1.863891	25.393435	5	2048
97	264.678255	0.000105	7.381702	257.296448	5	4096
98	259.196528	0.000107	7.243404	251.953017	5	4096
99	260.073130	0.000110	6.806854	253.266166	5	4096
100	239.453893	0.000109	6.314054	233.139730	5	4096
101	253.860192	0.000109	7.346521	246.513562	5	4096
102	0.275015	0.000110	0.124694	0.150211	6	512
103	0.254239	0.000110	0.123534	0.130595	6	512
104	0.274878	0.000120	0.127442	0.147316	6	512
105	0.277553	0.000111	0.128935	0.148507	6	512
106	0.278521	0.000110	0.128907	0.149504	6	512
107	2.455511	0.000037	0.511182	1.944292	6	1024
108	2.475223	0.000111	0.507611	1.967501	6	1024

109	3.417116	0.000110	1.468747	1.948259	6	1024
110	2.462016	0.000111	0.504856	1.957049	6	1024
111	2.702673	0.000113	0.519007	2.183553	6	1024
112	24.129165	0.000140	2.058085	22.070940	6	2048
113	28.044679	0.000111	5.878773	22.165795	6	2048
114	28.087329	0.000111	5.823907	22.263311	6	2048
115	24.009676	0.000112	2.068803	21.940761	6	2048
116	24.013462	0.000109	2.036576	21.976777	6	2048
117	233.798825	0.000111	8.036566	225.762148	6	4096
118	232.057409	0.000110	8.255809	223.801490	6	4096
119	232.472379	0.000112	8.234173	224.238094	6	4096
120	237.641158	0.000111	7.163314	230.477733	6	4096
121	242.127048	0.000038	8.227809	233.899201	6	4096
122	0.405782	0.000041	0.142599	0.263142	7	512
123	0.352422	0.000042	0.147127	0.205253	7	512
124	0.399287	0.000041	0.145954	0.253292	7	512
125	0.414854	0.000110	0.148139	0.266605	7	512
126	0.329588	0.000110	0.141885	0.187593	7	512
127	2.523778	0.000110	0.568044	1.955624	7	1024
128	2.570747	0.000110	0.550592	2.020045	7	1024
129	2.620252	0.000107	0.580591	2.039554	7	1024
130	2.564208	0.000042	0.587855	1.976311	7	1024
131	2.684238	0.000112	0.553764	2.130362	7	1024
132	24.202126	0.000110	2.325155	21.876861	7	2048
133	23.424662	0.000109	2.320224	21.104329	7	2048
134	23.018848	0.000111	2.020031	20.998706	7	2048
135	24.517571	0.000049	2.292656	22.224866	7	2048
136	27.526739	0.000039	5.839534	21.687166	7	2048
137	228.957231	0.000111	8.129281	220.827839	7	4096
138	233.261078	0.000110	9.284470	223.976498	7	4096
139	231.300152	0.000040	9.043786	222.256326	7	4096
140	234.141242	0.000112	9.368062	224.773068	7	4096
141	231.254358	0.000111	8.723718	222.530529	7	4096
142	0.376519	0.000040	0.138080	0.238399	8	512
143	0.374877	0.000038	0.138049	0.236790	8	512
144	0.377778	0.000040	0.142567	0.235171	8	512
145	0.375676	0.000043	0.139303	0.236330	8	512
146	0.379960	0.000043	0.143980	0.235937	8	512
147	2.448079	0.000110	0.495898	1.952071	8	1024
148	2.467748	0.000042	0.534718	1.932988	8	1024

```
2.496587 0.000055 0.560858 1.935674 8 1024
2.509166 0.000039 0.573664 1.935463 8 1024
2.486479 0.000113 0.562523 1.923843 8 1024
23.009035 0.000112 2.300510 20.708413 8 2048
21.574633 0.000111 2.200621 19.373901 8 2048
20.996316 0.000113 2.251458 18.744745 8 2048
21.641063 0.000112 2.244310 19.396641 8 2048
21.498175 0.000113 2.272304 19.225758 8 2048
224.772050 0.000112 9.098900 215.673038 8 4096
226.622632 0.000112 9.008552 217.613968 8 4096
224.770204 0.000110 9.100859 215.669235 8 4096
225.934416 0.000110 9.094506 216.839800 8 4096
222.015833 0.000107 9.090105 212.925621 8 4096
```

```
#total_s   init_s     gen_s      mult_s   cores size
0.2229228 0.0001024 0.0103816 0.2124388 1 512
7.3285032 8.92e-05 0.0336656 7.2947484 1 1024
74.5508206 8.78e-05 0.0821458 74.468587 1 2048
728.381565 0.000102 0.2932984 728.0881646 1 4096
0.2629434 8.22e-05 0.10185 0.1610112 2 512
4.172959 0.0001102 0.410343 3.7625058 2 1024
41.2953924 0.0001102 1.6112314 39.6840508 2 2048
387.319429 9.38e-05 6.7795374 380.5397978 2 4096
0.2536428 5.28e-05 0.1428184 0.1107716 3 512
3.191769 7.5e-05 0.6151534 2.5765406 3 1024
29.998098 8.2e-05 3.0061964 26.9918196 3 2048
287.150105 7.94e-05 13.2549148 273.8951108 3 4096
0.2042892 4.12e-05 0.11953 0.084718 4 512
2.4089484 5.38e-05 0.4916976 1.917197 4 1024
23.4132868 8.78e-05 2.6483742 20.7648248 4 2048
229.9758476 0.0001028 10.624216 219.3515288 4 4096
0.2606714 9.7e-05 0.1166058 0.1439686 5 512
2.800796 0.0001104 0.4794672 2.3212184 5 1024
27.6943614 0.0001106 2.496377 25.1978738 5 2048
255.4523996 0.000108 7.018507 248.4337846 5 4096
0.2720412 0.0001122 0.1267024 0.1452266 6 512
2.7025078 9.64e-05 0.7022806 2.0001308 6 1024
25.6568622 0.0001166 3.5732288 22.0835168 6 2048
235.6193638 9.64e-05 7.9835342 227.6357332 6 4096
0.3803866 6.88e-05 0.1451408 0.235177 7 512
```

```
2.5926446 9.62e-05 0.5681692 2.0243792 7 1024
24.5379892 8.36e-05 2.95952 21.5783856 7 2048
231.7828122 9.68e-05 8.9098634 222.872852 7 4096
0.376962 4.08e-05 0.1403958 0.2365254 8 512
2.4816118 7.18e-05 0.5455322 1.9360078 8 1024
21.7438444 0.0001122 2.2538406 19.4898916 8 2048
224.823027 0.0001102 9.0785844 215.7443324 8 4096
```