

MP1

by Abhishek Johri



February 22, 2017

CS 484 Parallel Programming

University of Illinois at Urbana Champaign

**Part A**

Part 1:

The following table is the data that I collected by running the following bash commands

./basic\_perf.exe 128 128 32 32

./basic\_perf.exe 256 256 32 32

./basic\_perf.exe 512 512 32 32

./basic\_perf.exe 1024 1024 32 32

./basic\_perf.exe 2048 2048 32 32

./basic\_perf.exe 4096 4096 32 32

./basic\_perf.exe 5000 5000 32 32

./basic\_perf.exe 6000 6000 32 32

./basic\_perf.exe 8192 8192 32 32

./basic\_perf.exe 10000 10000 32 32

Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Vanilla | Flipped | Unroll-4 | Unroll-8 |
| 128 | 246248 | 326721 | 198310 | 204565 |
| 256 | 1948228 | 4397125 | 956315 | 1052237 |
| 512 | 7848766 | 18042924 | 4302470 | 4365532 |
| 1024 | 29907185 | 71471315 | 17107408 | 15393311 |
| 2048 | 152598258 | 521175636 | 115509850 | 116417792 |
| 4096 | 674744449 | 4044595264 | 497935343 | 488876274 |
| 5000 | 731384655 | 1677261508 | 347155133 | 365839351 |
| 6000 | 1066980769 | 2450581355 | 511367341 | 533807923 |
| 8192 | 2583739079 | 15651455066 | 1953372307 | 1888490992 |
| 10000 | 2954561669 | 6725570670 | 1411969686 | 1470220268 |

Here is the graph of the following data:

Analysis:

So the first thing to notice in this experiment is that we kept the block size constant but change the size of the array thus increasing the number of elements we have in total. So we have 4 different parts ways of executing the same code to see which way give us the best cache performance. We have a Vanilla structure which is a simple brute force method, a Flipped structure which is the same as the Vanilla’s brute force structure just that the inner and out loops are switched. The final two structures are similar since they are both unrolling structures but with a different unrolling amount. From the graph we can see that based off the Vanilla structure (being our baseline due to basic brute force structure) that our Flipped structure is not very cache efficient due to the increase in runtime. This increase in runtime is due from the fact that the program needs to access the actual memory more often than what is stored in the cache. After this we can see that the two programs that are more cache efficient than the Vanilla structure are the Unrolling structures. This is due to the fact that both of these method load a certain amount of elements at once in each iteration thus loading it into the cache right away. This helps because then a certain number of elements, that are repeatedly needed, are on the cache for quick access to. The only time that this wouldn’t work would be if the fixed number of elements you choose to go through in one iteration is greater than the cache line.

Part 2:

The following table is the data that I collected by running the following bash commands:

./basic\_perf.exe 2048 2048 4 4

./basic\_perf.exe 2048 2048 8 8

./basic\_perf.exe 2048 2048 16 16

./basic\_perf.exe 2048 2048 32 32

./basic\_perf.exe 2048 2048 64 64

./basic\_perf.exe 2048 2048 128 128

./basic\_perf.exe 2048 2048 256 256

./basic\_perf.exe 2048 2048 512 512

./basic\_perf.exe 2048 2048 1024 1024

./basic\_perf.exe 4096 4096 4 4

./basic\_perf.exe 4096 4096 8 8

./basic\_perf.exe 4096 4096 16 16

./basic\_perf.exe 4096 4096 32 32

./basic\_perf.exe 4096 4096 64 64

./basic\_perf.exe 4096 4096 128 128

./basic\_perf.exe 4096 4096 256 256

./basic\_perf.exe 4096 4096 512 512

./basic\_perf.exe 4096 4096 1024 1024

Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2048 Matrix |  |  | 4096 Matrix |  |
| Block Size | Time |  | Block Size | Time |
| 4 | 143529251 |  | 4 | 651767983 |
| 8 | 124894741 |  | 8 | 513867767 |
| 16 | 122279548 |  | 16 | 514259963 |
| 32 | 122835633 |  | 32 | 493769709 |
| 64 | 119327383 |  | 64 | 480038509 |
| 128 | 103872830 |  | 128 | 417607486 |
| 256 | 100811823 |  | 256 | 407882871 |
| 512 | 102547141 |  | 512 | 985631486 |
| 1024 | 230939097 |  | 1024 | 1024154124 |

Here are the graphs for the following data:

Analysis:

So the first thing to note is that we are keeping the size of our matrix constant throughout all of the experiments for this part. Different from the first part our program now has a blocking structure which in some ways is similar to that of the unrolling structure. The main thing that we are changing here is the block size for our program to see around where the program is most cache efficient. So what we expect is that the graph will dip down till it hits a global minimum. This minimum indicates the point at which the program is most cache efficient. This is because since more as-needed data is on the cache the program will run faster giving a lower time. This is shown and supported in the graph.

**Part 2**