Tony Ho

CPSC 4210

Project 2

Introduction:

Matrix-matrix multiplication (MM) is a mathematical operation that has been studied and researched extensively, especially for high performance and parallel computing calculation. The reason is because of MM's huge size of data when computing matrices. To compute two 512 matrices, 268,435,456 operations are needed. For two 1024 matrices, 2,147,483,648 operations are computed. Therefore, the MM is a time-consuming and hardware heavy program. This report will show the improvement of MM when using parallel computing with OpenMP and CUDA.

The problem study for this project is: How to implement serial MM using OpenMP in its original, loop optimized, and blocking forms. Then I will improve those serial MM forms by parallelizing them with OpenMP multithreading. Finally, I will use CUDA to implement those MM forms for GPU computation. However, there are several problems:

1. For serial MM computing, what techniques and methods to use to improve runtime?
2. After improving the serial code with OpenMP, how does the performance compare to serial MM?
3. How performance compare when using different thread scheduling for the OpenMP?
4. How performance of CUDA compare to OpenMP? Multi-processor compares to multithreading (GPU vs CPU).

To address those problems. I implemented three programs for serial computing, OpenMP parallel computing, and CUDA parallel computing. The three programs record the runtime (sec) and the rate (MegaFLOPS/sec) of each technique three times and calculate the average of runtime and rate, then display it to the user for comparison.

Implementation:

- **Serial Computing:**

  I used two strategies to improve the original serial code:

  1. Loop optimization: replace the i-j-k sequence in the original with i-k-j for better spatial locality in cache memory (stride-1).
  2. Blocking optimization: splitting each matrix into smaller blocks, size of 16 each, for better temporal locality in cache memory. This strategy will maximize the reusing of elements.

  I tested this program with matrix's dimension of 1024, 2048, 4096, 8192. Also, each strategy will run 3 times and return the average of elapse time and rate.

- **OpenMP parallel computing:**

  I implement OpenMP based on the serial code. I implemented the original, loop optimized, and blocking serial code each with static, dynamic, and guided threading schedule. This resulted in 9 functions total.

  I tested this program by running it with 1, 2, 4, 8, 16, and 32 threads to see the differences in multithreading. Each function executes 3 times and return the average of elapse time and rate.

- **CUDA**:

  I implemented CUDA based on the OpenMP code. The program uses the loop with i-k-j sequence to execute on global memory, and block optimization to execute on shared memory. It will then return the average of elapse time and rate after running those each function three times.

Hardware Specification:

- **Local (Lab computer)**:

  36 aquaman

- Number of physical processing cores: 4
- Number of logical processing cores: 8
- CPU speed: 3.4 GHz

- CPU registers: MSR, MTRR, FXSR, and XTPR
- Number of threads available: 8
- Simultaneous Multiprocessing availability: No, uniprocessor
- Cache memory information:
    - L1d cache size: 4 x 32KB
    - L1i cache size: 4 x 32KB
    - L2 cache size: 4 x 256 KB
    - L3 cache size: 8192 KB

- **OpenMP**:

Medusa

- Number of physical processing cores: 8
- Number of logical processing cores: 16
- CPU speed: 3.0 GHz
- CPU registers: MSR, MTRR, FXSR, and XTPR
- Number of threads available: 16
- Simultaneous Multiprocessing availability: Yes, up to 2 processors
- Cache memory information:
    - L1d cache size: 8 x 16 KB 4-way set associative data caches
    - L1i cache size: 4 x 64 KB 2-way set associative shared instruction caches
    - L2 cache size: 8192 KB 16-way set associative shared exclusive L2 caches
    - L3 cache size up 8192 KB

- **CUDA:**

17siler

- Number of physical processing cores: 4
- Number of logical processing cores: 8
- CPU speed: 3.4 GHz
- CPU registers: MSR, MTRR, FXSR, and XTPR
- CUDA cores: 192
- GPU clock rate: 854.98 MHz

- GPU Registers per block: 65536
- Total global memory: 981.875 MB
- Total constant memory: 64 KB
- Shared memory per block: 48 KB
- Maximum threads per block: 1024
- Warp size: 32
- Maximum resident threads per multiprocessor: 2048

Experiment:

The experiments include serial computation, OpenMP parallel computation, and CUDA parallel computation. Each of the experiment was executed with dimension of 1024, 2048, 4096, and 8192.

- The experiments of serial computing were executed with OpenMP using 1 thread. The test cases used were original, i-j-k, matrix-matrix multiplication (MM), loop optimized, i-k-j, MM, and MM using blocking.
- With the OpenMP experiments, I executed the program with those 4 dimensions, each with 2, 4, 8, 16, and 32 threads. Furthermore, original, loop optimized, and blocking MM was executed with static, dynamic, and guided schedule.
- For CUDA, I tested the program with loop optimization of global memory and blocking optimization with shared memory.

Discussion of Results:

- **Serial computation:**
    - Loop and blocking optimization gave about 50% runtime improvement than original serial with dimension size of 1024, 2048, 4096, and 8192. The loop optimization has better result than block optimization with dimension of 1024 and 2048.
    - The loop optimized MM use i-k-j loop to get stride-1 in row-order loop. I saw better result in speed up, runtime, and cache misses because of better spatial locality.

- The blocking MM splits the matrices into block of size 16 for better temporal locality. I chose block size of 16 after experimented different size. The blocking MM has less effect on small dimension (1024 and 2048) compare to loop optimized MM. However, it achieved about 50% more efficient with dimension bigger than 2048 (4096 and 8192 compare to loop optimized MM.

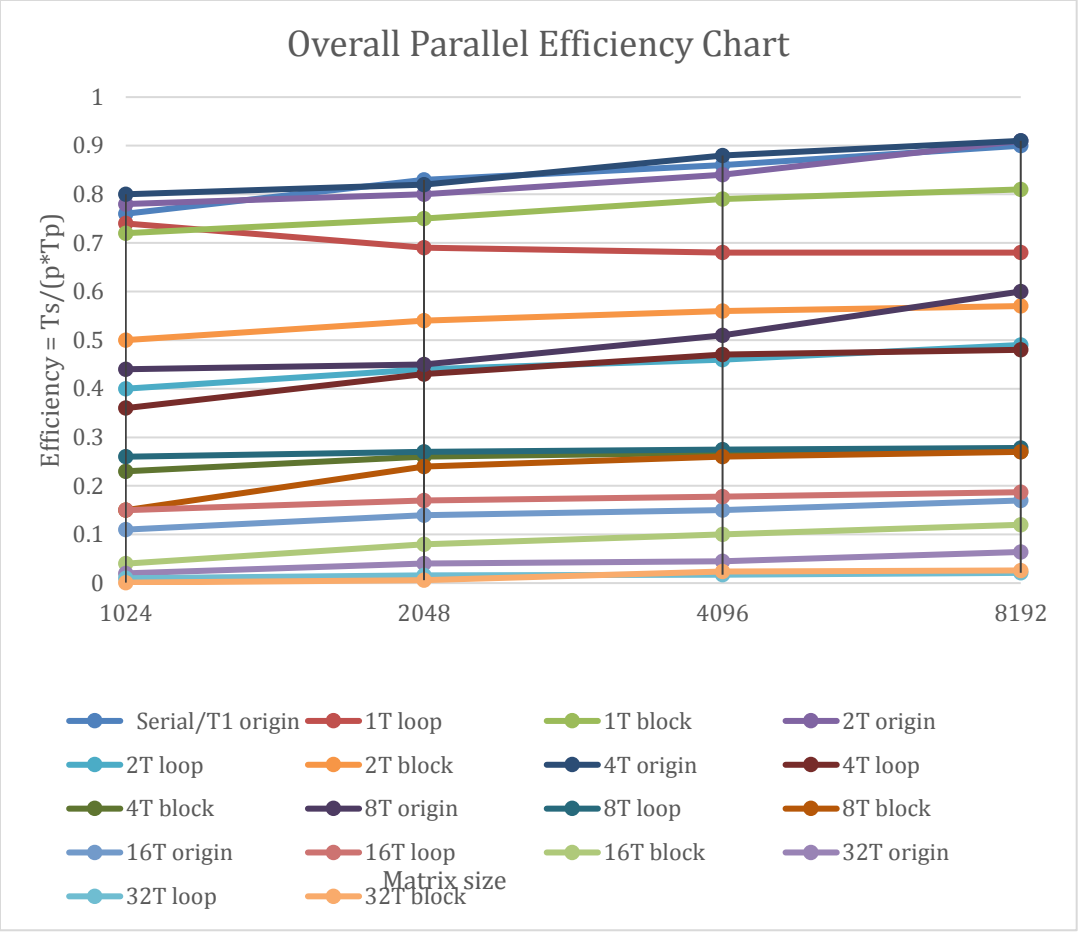
- **OpenMP:**
  - With 2 threads, I got a small improvement compared to serial computing because of the overhead of thread switching decreases as each thread has a precisive amount of computations to execute. However, in theory, the performance should be double with double the number of threads, but it did not happen here. I measured an improvement of around 183% instead of 200% using MFLOPS/sec compare to serial computing which use 1 thread.
  - I saw a notable improvement, around 159%, with 4 threads as overhead starts to decrease along with rescheduling. With 8 threads, again I saw an improvement of around 183% compare to 4 threads as the overhead is constant cost. The actual improvement from 4 to 8 threads came close to the theoretical improvement of 200%.
  - However, I came to the closest of theoretical improvement from 8 to 16 threads, of around 187%. This was a peek improvement because from 16 to 32 threads only saw around 113% improvement. But the program got the biggest MFLOPS/sec with 32 threads, 937.310795.
- **OpenMP with static, dynamic, and guided schedule:**
  - Static scheduling divides the computation into equal size chunks, but there is nothing else to affect the runtime. It had the second-best result.
  - Dynamic scheduling had the worst performance because it needs to start from chunk size of 1. Therefore, the thread must be rescheduled rather than having threads to work on appropriate chunk size.
  - Guided scheduling had the best result out of the three. This was because the chunk size is adjusted dynamically based on a periodically calculation by the OS to maximize performance.

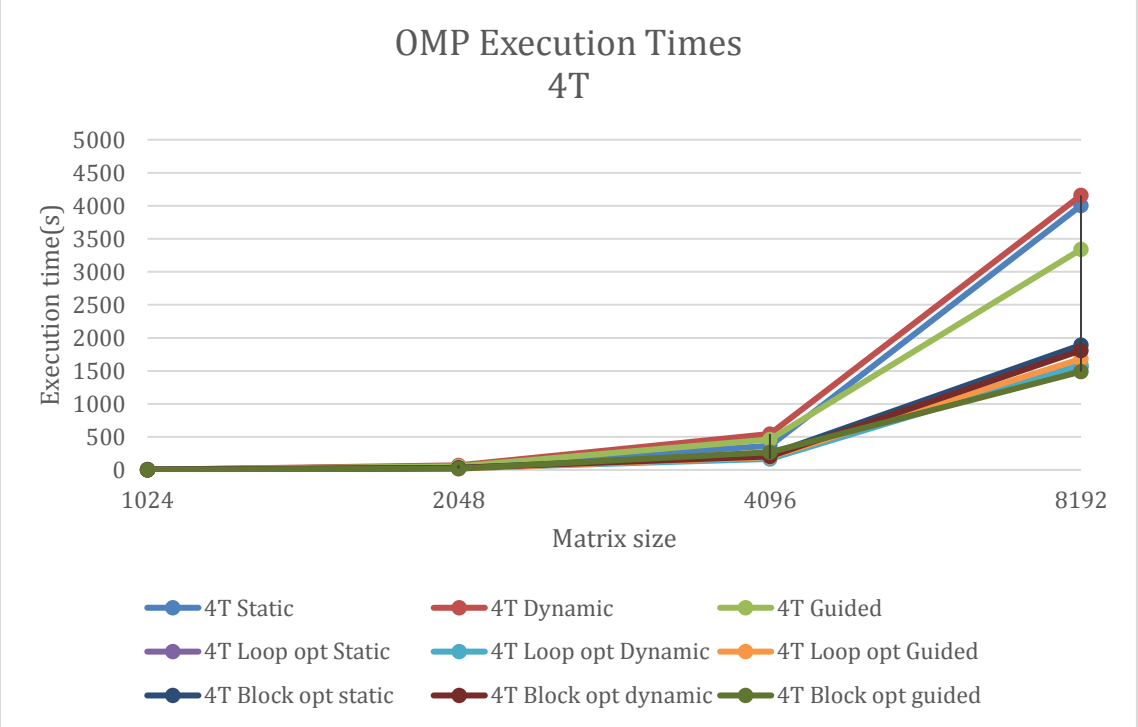
- **CUDA**:
    - The CUDA program saw a significant improvement compare to OpenMP. I used the 17SILER computer to test CUDA. The computer's GPU has 6,144 maximum available threads. The CUDA MM has the block size of 16 for all the test cases. However, I got segmentation fault when tried to test the program with size of 4096 and 8192. CUDA has the most efficient computing power compare to OpenMP for MM.

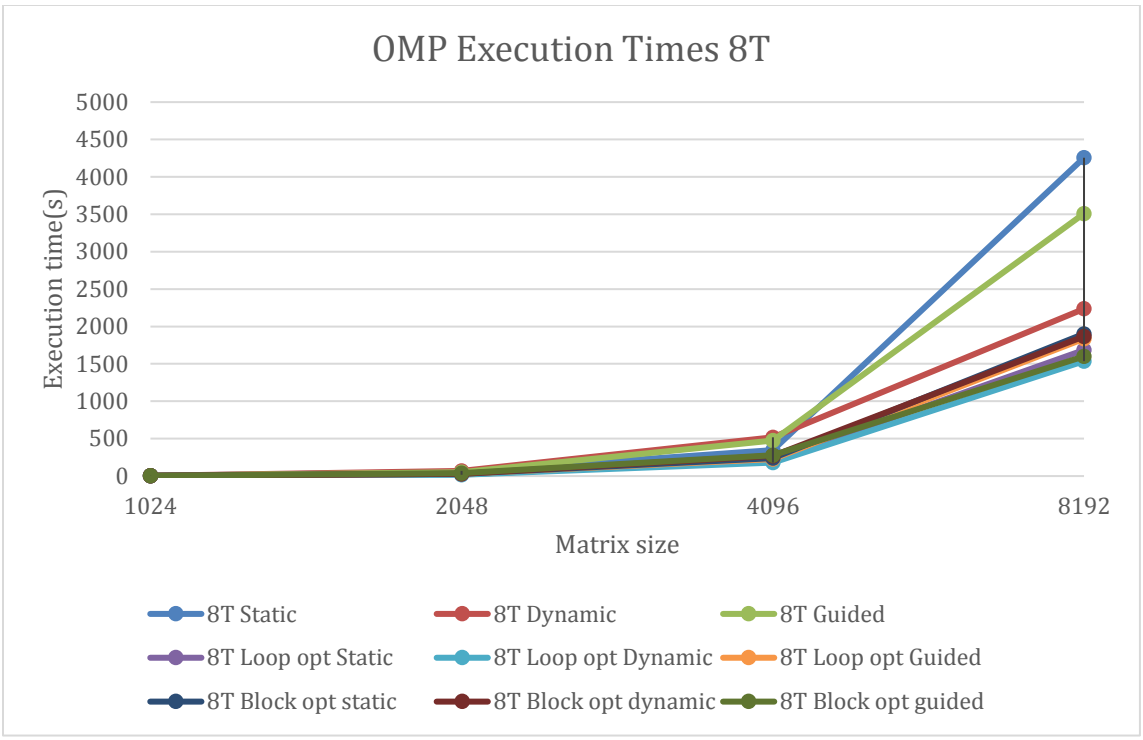
To compare FLOPS, I use FLOPS in Mega unit, and the average of 1024 and 2048 because I could not get the results for 4096 and 8192.

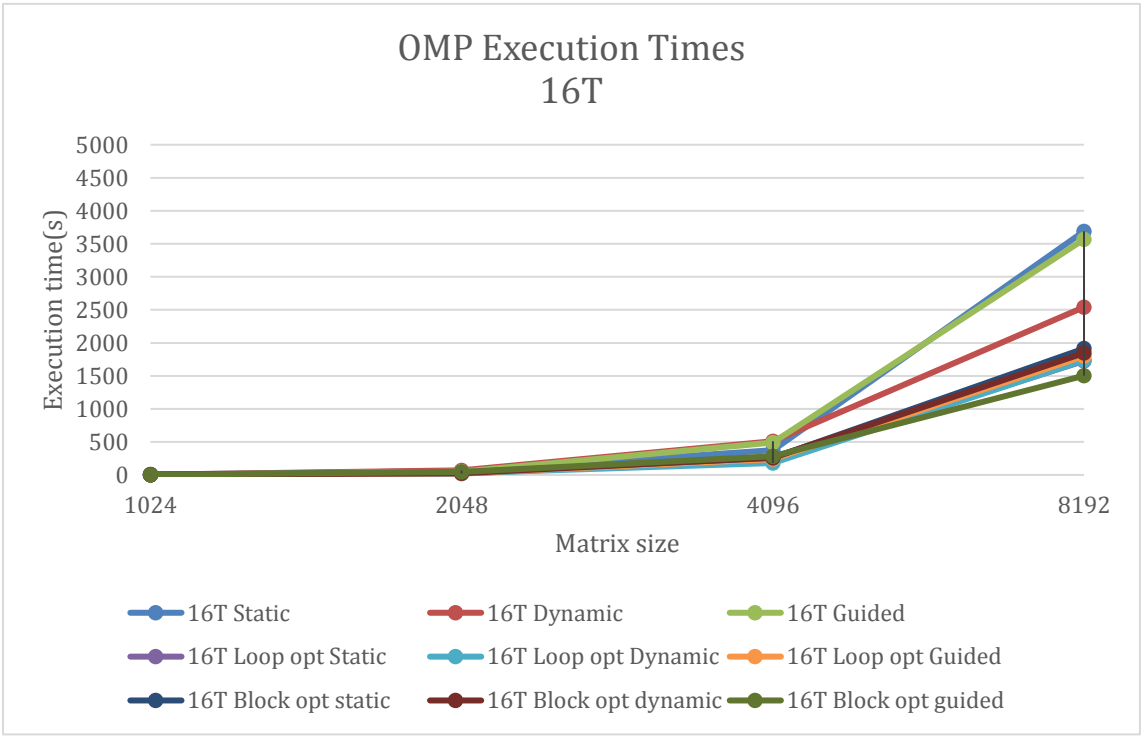
# of thread	MFLOPS/sec
Serial/1T	86.176584
2T	157.597314
4T	250.350479
8T	458.547030
16T	830.257481
32T	937.310795
CUDA	6589.214996

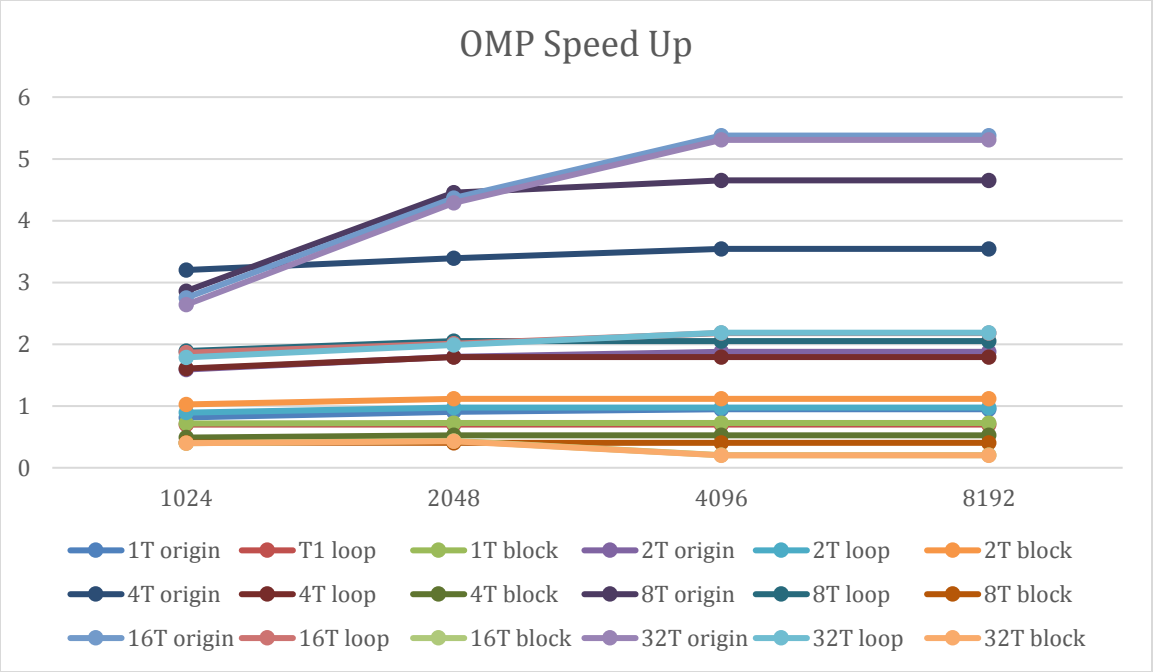
Figures:

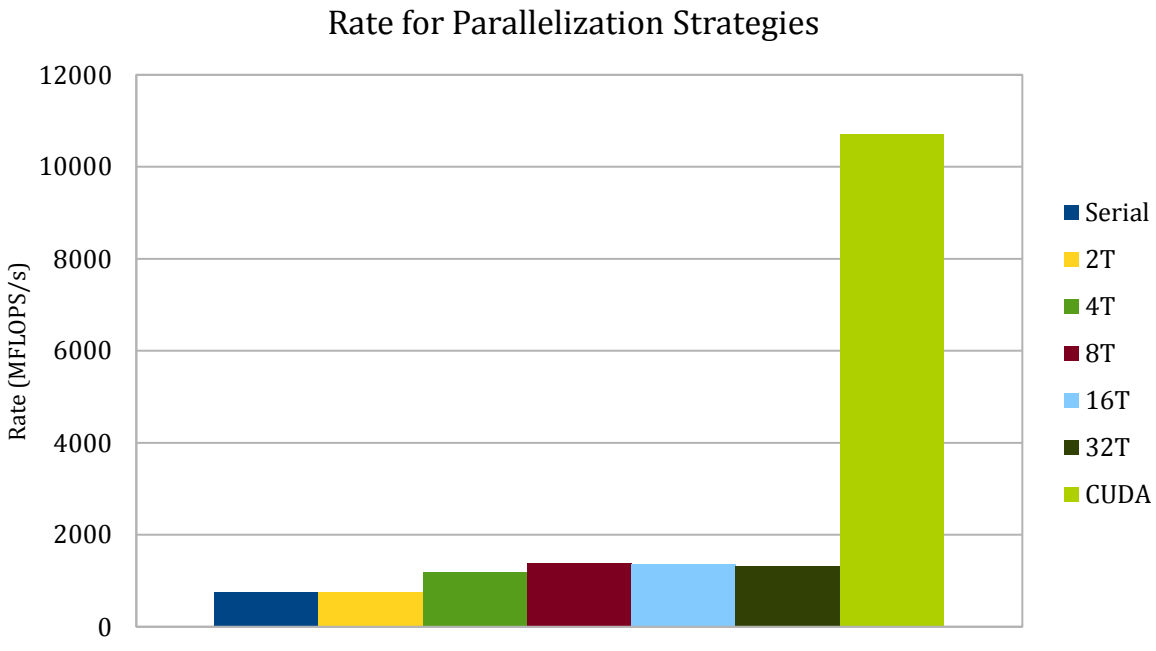

Overall Parallel Efficiency Chart
Efficiency = Ts/(p*Tp)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1024
2048
4096
8192
Serial/T1 origin
1T loop
1T block
2T origin
2T loop
2T block
4T origin
4T loop
4T block
8T origin
8T loop
8T block
16T origin
16T loop
16T block
32T origin
32T loop
32T block
Matrix size


OMP Execution Times
4T
Execution time(s)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1024
2048
4096
8192
Matrix size
4T Static
4T Dynamic
4T Guided
4T Loop opt Static
4T Loop opt Dynamic
4T Loop opt Guided
4T Block opt static
4T Block opt dynamic
4T Block opt guided


OMP Execution Times 8T
Execution time(s)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1024
2048
4096
8192
Matrix size
8T Static
8T Dynamic
8T Guided
8T Loop opt Static
8T Loop opt Dynamic
8T Loop opt Guided
8T Block opt static
8T Block opt dynamic
8T Block opt guided


OMP Execution Times
16T
Execution time(s)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1024
2048
4096
8192
Matrix size
16T Static
16T Dynamic
16T Guided
16T Loop opt Static
16T Loop opt Dynamic
16T Loop opt Guided
16T Block opt static
16T Block opt dynamic
16T Block opt guided


OMP Speed Up
6
5
4
3
2
1
0
1024
2048
4096
8192
1T origin
T1 loop
1T block
2T origin
2T loop
2T block
4T origin
4T loop
4T block
8T origin
8T loop
8T block
16T origin
16T loop
16T block
32T origin
32T loop
32T block


Rate for Parallelization Strategies
Rate (MFLOPS/s)
12000
10000
8000
6000
4000
2000
0
Serial
2T
4T
8T
16T
32T
CUDA