

Abhinandan Majumdar (am2993)
 Shrivathsa Bhargav (sb2784)
 CSEE w4824 – Computer Architecture
 Project

Goals

The goal of this project is to optimize the run time of a matrix multiplication on both the software and hardware levels. There are no limits to the software optimization, but there are some imposed limits on what can be modified on the hardware level.

The run time is denoted by the number of clock ticks the SESC simulator takes to complete execution. As a starting point, the unoptimized program takes 3,694,272 clock ticks to execute.

Software optimization

The possibilities here are numerous. We will go over this in the order attempted.

1. **Matrix ordering through dynamic programming.** For details on dynamic programming, see Chapter 15.2, “Introduction to Algorithms” by Cormen. Reordering the matrices ensures that the multiplication being performed takes the least amount of time. For instance, the default program structure multiplies the matrices as $(((((A*B)*C)*D)*E)*F)$.

This may be terribly inefficient. The reordering algorithm looks at the matrix sizes (which are randomly generated), and then changes the order of multiplication accordingly. For instance, the reordered multiplication may be:

$$((A*(B*C))*D)*(E*F))$$

The brunt of the analysis then becomes handling multiplication between only two matrices.

Approximate clock ticks: 1,006,650 i.e. one third of the original one (3,694,272)

2. **Strassen’s algorithm for matrix multiplication.** This was suggested in the project handout, but did not yield a reduction in clock ticks, which might be because of a large number of recursions being called. This method was then discarded.

Approximate clock ticks: 3,072,812 (increased after ordering)

3. **Threaded multiplication.** Since the random numbers being generated are multiples of 4, we pumped four threads at a time multiplying each one fourth part of first matrix to the second matrix and storing it to the corresponding one-fourth part of resultant matrix. By doing this, the clock ticks for our default case (for one four-way out-of-order big core with 256k cache) increased from 1,006,650 to 1,242,811. This motivated us to use multiple cores, after which we performed hardware optimization minimize clock ticks with two 3 way out-of-order middle cores with 128k cache.

Approximate clock ticks: 762,123

4. **Using stack variables instead of heap variables.** While doing matrix multiplication, we were storing the result in the matrix element itself, presenting a 3-level iteration. Since the matrix was created dynamically using malloc, which uses a portion of heap that takes much more time to access, we tried using the stack memory portion of main memory by using local variables to store the intermediate result and transfer the final result to the matrix columns. By doing this, instead of iteratively accessing the matrix elements (which might add lot of cache misses because of column-wise access), we used temporary variables. This reduced the clock ticks even further from 762,123 to 621,214.

Approximate clock ticks: 621,214

5. **Further usage of threaded multiplication.** Instead of having only 4 threads spread across all the rows of the matrix, we used one thread for each row. Hence, for a 32x18 and 18x28 matrix multiplication, we have 32 threads running, computing each row elements of the final matrix. This reduced the number of iterative FOR loops inside the program from 3 to 2 saving quite a few branch penalties, and resulted in even better clock ticks.

Approximate clock ticks: 471,312

6. **Loop unrolling (First level).** This technique was also suggested in the project handout. There are four areas where loop unrolling can be effected:
 - a. **Cell * cell unrolling.** For AxB matrix multiplication, we need to multiply every element of a particular row of A to every element of the corresponding column of B. We unrolled the inner-most loop twice and four times so that it does two and four multiplications at a time saving unnecessary branches. However, when we unrolled the loop twice, the clock ticks got reduced, but when unrolled four times, it increased by 5000 ticks.
 - b. **Row * column unrolling.** For AxB, matrix multiplication, we multiply a particular row of A by every column of B. Hence we unrolled the outer most loop (we have only two levels of iterations for matrix multiplication after we did multithreading), twice and four times. The clock ticks reduced for the first case and increase slight a bit for the second case.
 - c. **Thread level unrolling.** We can further unroll at the thread creation level where we are calling multiple threads to do row vs. matrix multiplication. Instead of calling it one by one in a FOR loop, we tried calling it twice and four times in a loop. However, for the first case, the clock ticks got reduced but when unrolled four times, it increased drastically by 7000 ticks, indicating we are pumping the threads earlier than it could finish.
 - d. **Memory creation.** We can also unroll at the memory creation level required to store intermediate matrix results. Instead of creating columns for each row one by one in a

loop, we tried creating columns for two and four rows. For the first case, clock ticks got reduced but for the second case, it increased slightly.

Now, since there was a discrepancy related to how much unrolling we should do, we tried obtaining results for unrolling by two and unrolling by four, and observed that for two 3-way middle cores with 128k cache, we got less clock ticks by unrolling it by four (there was a difference of 2000 ticks). Hence we reject the unrolling by two iterations.

Approximate clock ticks: 432,504

7. **Loop Unrolling (Optimization).** After 6th step, we tried playing with unrolling by 2 or 4 at all the four levels, and could reduced the clock ticks by 2000 if we pump two threads in a loop instead of four, and do two level row – column unrolling.

Approximate clock ticks: 428,192

8. **Code Cleanup.** We removed unnecessary statements / variable declarations and could further reduce the clock ticks by 1000.

Approximate clock ticks: 426,713

9. **Load balancing.** At this level, we were mostly sure that we were going to stick with using two 3-way out-of-order middle cores with 128k cache with area = 4.87mm². However, we were auto assigning the threads to any idle processor, which caused one processor to be unevenly loaded than the other. By carefully managing the assignments of the threads, we found that evenly allocating threads yielded better performance. So, we allocated two threads to two cores in a round robin fashion such as the first thread is allocated to processor 0, the second to processor 1, and the third to processor 0, etc. However, by doing this, it prevented us from playing with other core combination as the program would throw a segmentation fault if we try to allocate a thread to a non-existent third core if we have only two cores in our architecture.

Approximate clock ticks: 418,207¹

10. **Backtrack.** However, until now, we haven't attempted all possible core combinations which might yield better results (still within the 5mm² area limit), that were not specified in the project description. Hence we undo the step 9 changes, and restart from step 8. We tried doing an exhaustive search on all possible core combinations, and could drastically reduce the clock ticks to 363,331 by using three 2-way out-of-order middle cores with 32k cache with a total area of 4.91mm².

Approximate clock ticks: 363,331

11. **Load Balancing (revisited).** Here, we try to do load-balancing but it did not yield a better result as we have three cores and we need to allocate either two or four threads to three cores, which

¹ This is the optimal result as per the (limited subset of) core combinations in the project description handout.

involves additional computation, and hence increases the clock ticks. Hence we stick to the following core combination shown in table 1.

Table 1 - Final hardware configuration

Core(s)	3-middle cores
Type	2-way out of order
Cache size	32k
Area used	4.91 mm ²
Clock ticks	363,331

Hardware optimization

We performed an exhaustive analysis of the hardware in parallel with our optimized software and eliminated combinations resulting in areas greater than the acceptable limit of 5 square-mm. Figure 1 gives a good overview of the approximate clock ticks for each combination (or combination set).

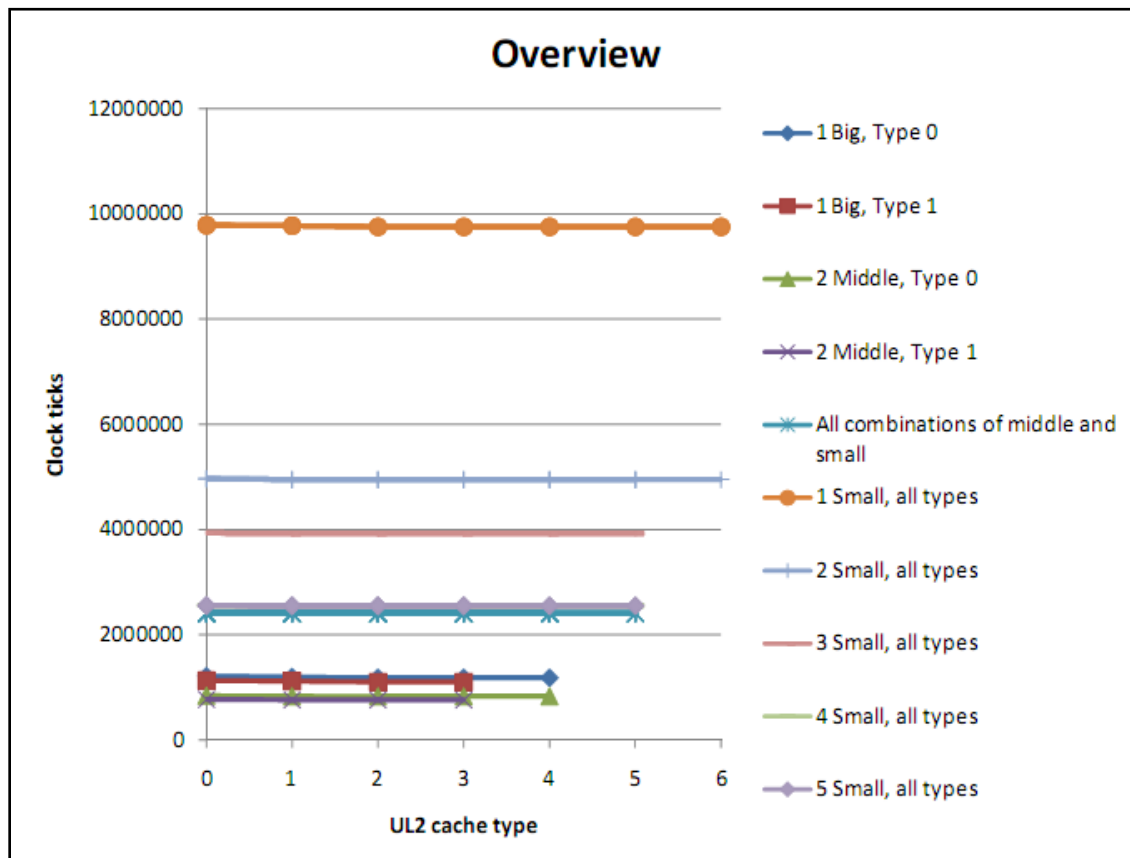


Figure 1 - Overview of exhaustive hardware optimization

For Big cores, type 0 = 4-way out of order, and type 1 = 6-way out of order. For Middle cores, type 0 = 2-way out of order, and type 1 = 3-way out of order.

Right off the start, everything resulting in clock ticks higher than 2,000,000 can be discarded. This yields figure 2.

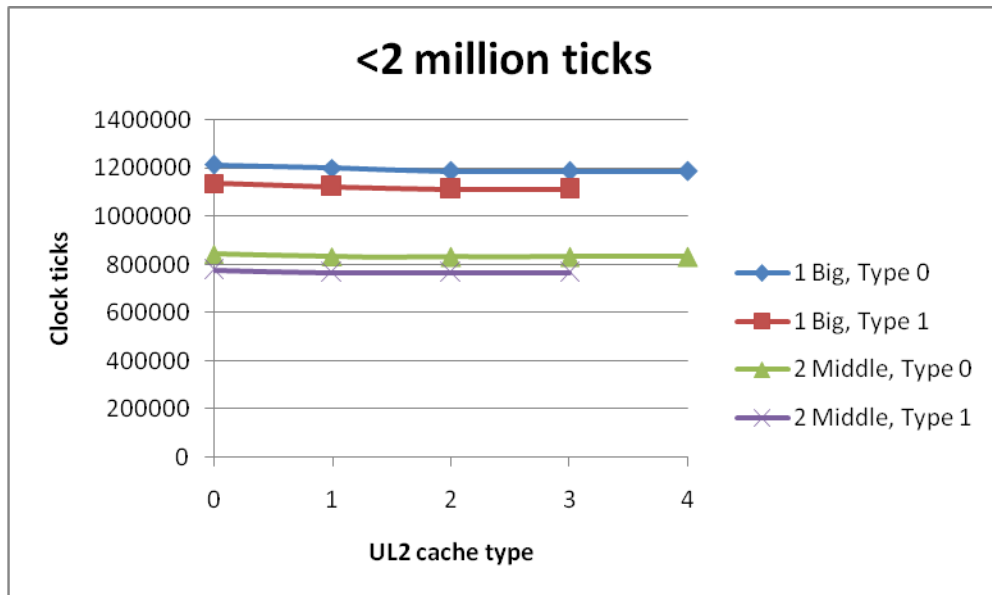


Figure 2 - Hardware choices resulting in <2 million clock ticks

The real contest, then, is between the big and middle cores.

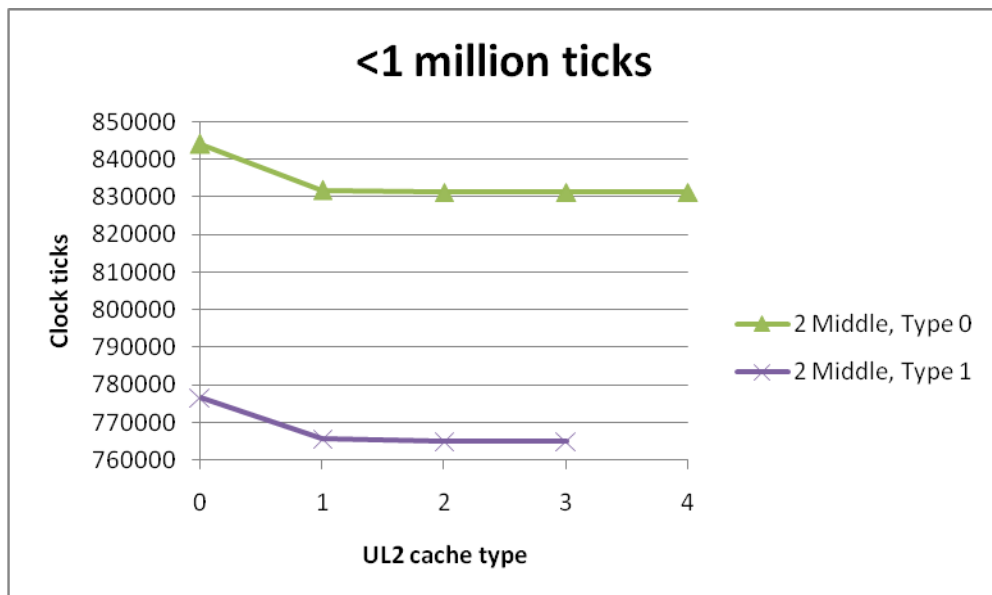


Figure 3 - Hardware choices resulting in <1 million clock ticks

Two middle cores of type 1 seems to be the best choice. Now, we can attempt to find the optimum UL2 cache type. The UL2 types are as follows: type 0 = 16k, 1 = 32k, 2 = 64k, 3 = 128k

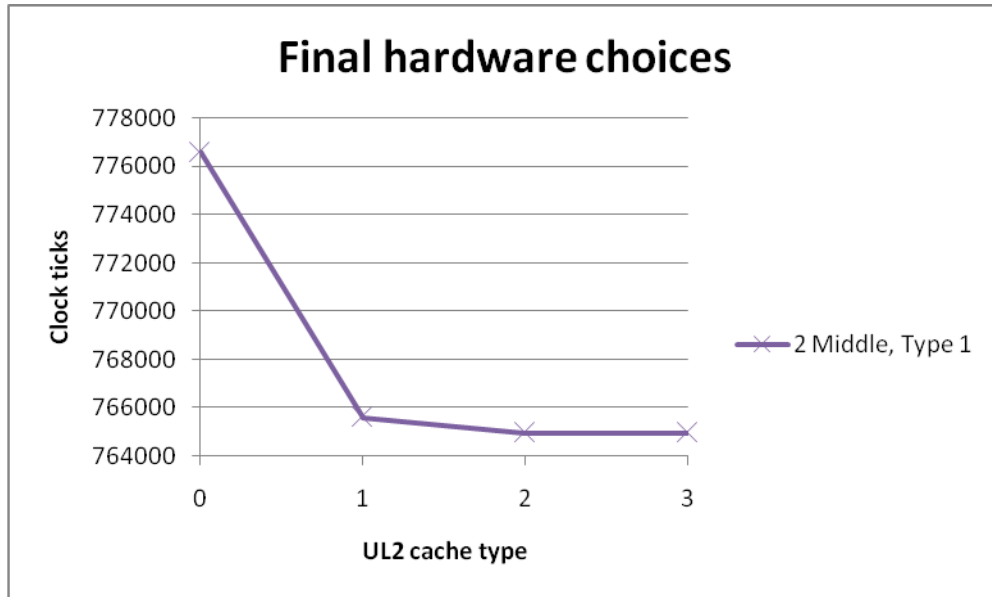


Figure 4 - Final hardware choices for UL2 cache size

There's no difference in clock ticks between having a 64k cache and a 128k cache. In the interest of saving area, we pick the 64k cache.

We then went beyond the combinations presented in the project description, and attempted using 3 middle cores (still within the area limit). The results are illustrated in figure 5.

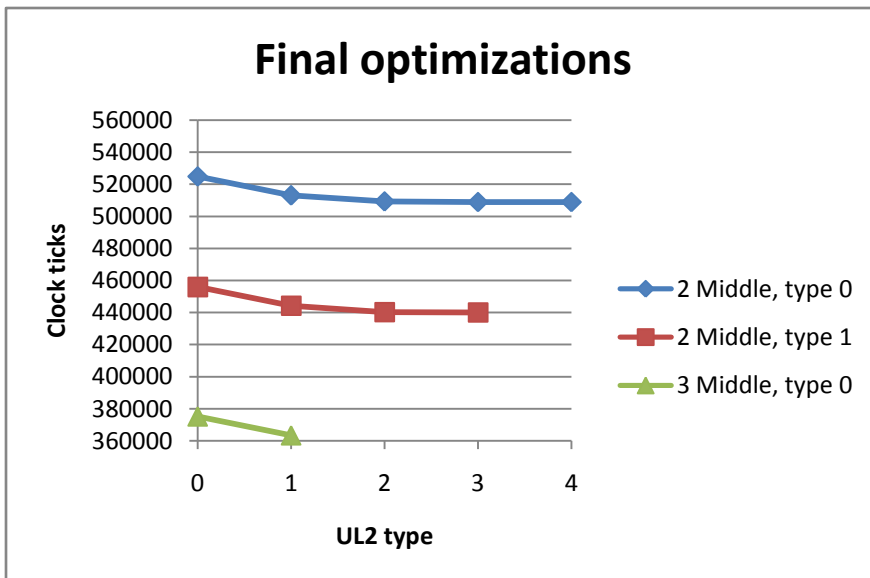


Table 2 - Final hardware choice

Core(s)	3-middle cores
Type	2-way out of order
Cache size	32k
Area used	4.91 mm ²
Clock ticks	363,331

Since 3 middle cores reduce the clock ticks drastically, that is our final choice.

The final optimized hardware choice is shown in table 2.