Optimising the LINPACK 1000 using Parallelization

Sam Serrels
40082367@napier.ac.uk
Edinburgh Napier University
Concurrent and Parallel Systems (SET10108)

1 Introduction

The LINPACK benchmark The Linpack algorithm is a popular benchmark in the high performance computing field as a floating point performance measure for ranking supercomputers. The Benchmark solves a large system of linear equations using LU decomposition and is typical of many matrix-based scientific computations. Linpack started as a Fortran maths processing application, the code for solving linear equations was extracted from the original program and turned into a benchmark. [Dongarra et al. 2003]
The Linpack1000 algorithm first generates a random 1000x1000 element matrix, A, and 1000 element vector, b. The elements in A and b are all double precision floating point numbers. Processing then takes place to find the solution, a 1000 element vector, x, such that Ax = b.
The algorithm is split into 4 main tasks: generating the initial problem numbers, Gaussian elimination, Solving, and Validation.
This report documents the process of analysing a specific sequential Linpack implementation and then converting it to a parallel task.

Related Work As Linpack is used to benchmark large scale multiprocessor supercomputers, there are many parallel versions available. The main difference between implementations is the Gaussian elimination stage, for which there are many algorithms available, some of which can split the task up into separate easily parallelizable chunks of logic. For the scope of this project, changes to the algorithm were avoided wherever possible to keep a fair comparison to the original sequential code.

High-Performance Linpack The most commonly used implantation of Linpack is the HPL implementation, written by the Innovative Computing Laboratory at the University of Tennessee. HPL is an open-source project that aims to provide a toolbox for configuring, optimising, and running the benchmark over a network. It contains many version of the algorithm with plenty of configurable options and for tuning performance to a specific system.
The HPL project was used as a rough guide to how the reference algorithm could be modified for this project, however most of the optimisation were beyond the scope of this project.

Project Scope A quick optimisation would be to drop the precision of the algorithm from double, to single precision floating point values. Another method would be to swap the original Gaussian elimination algorithm for a different mathematical approach that would lend itself to parallel processing better. This project aimed to see how much the original Linpack code can be optimised with parallelization, without changing the core logic of the algorithm or data output so these routes for optimisation were ruled out.

OpenMP The technology for processing the application in parallel was chosen to be OpenMp, an API that abstracts the creation of threads from the user and therefore allows for easier development and better cross platform portability, assuming that the chosen platform has a complier that supports OpenMP. This was chosen over creating threads manually, mainly for ease of development reasons, but also because even in a situation that OpenMp is slower than Manual threads, there should still be a noticeable performance increase over the baseline results.

SIMD For an extra level of performance, SIMD instructions were used to gain performance in the most frequently executed parts of the program.

2 Linpack Gaussian elimination

On analysis of execution of the program , it was clear that the vast majority of the execution time was based in the Gaussian Elimination stage.

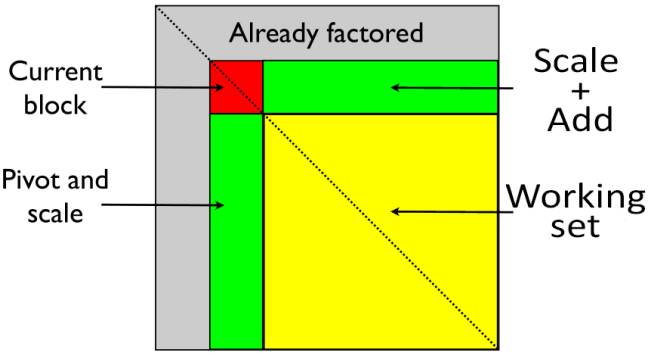

Figure 1: *Overview of GE progression* -

Initial analysis The Gaussian elimination(GE) stage, works it's way through the array, starting in the "top left corner". It examines the entire first column(C) ("Pivot and scale", figure 1) and finds the largest value (T). The row that contains the largest value(T) becomes the pivot, it is swapped with the topmost row. Then, each column is processed, by multiplying it by 1/T and then adding the value of the C column.

Once this is complete, the process restarts but in a subsection of the Matrix A that is one row and column smaller than the previous iteration. This continues until the "Bottom right" corner is reached. This process transforms Matrix A into a upper triangular matrix that is in row echelon form.

On further examination, the "Daxpy" function, which his called many times during the GE stage takes up nearly 90% of the total execution time. Daxpy is the function that does the scaling and addition of each column, and is called 424166 times during the full execution of the program.

Daxpy While the Daxpy function is called at a high frequency it contains very few lines of code. It's function is to compute Y = S * X + Y, X and Y are elements of two arrays, and S is a scaler value. In the program, this is used in a loop to process each column in the A array.

This was the first part of the code to be examined for possible speed-up, as each iteration of the loop doesn't depend on any other iteration. Initially Paralleling the loop with OpenMp was attempted, but as the loop will only ever execute a maximum of 1000 times, the overhead time of creating and running threads was always greater than the time of running the loop without threads.

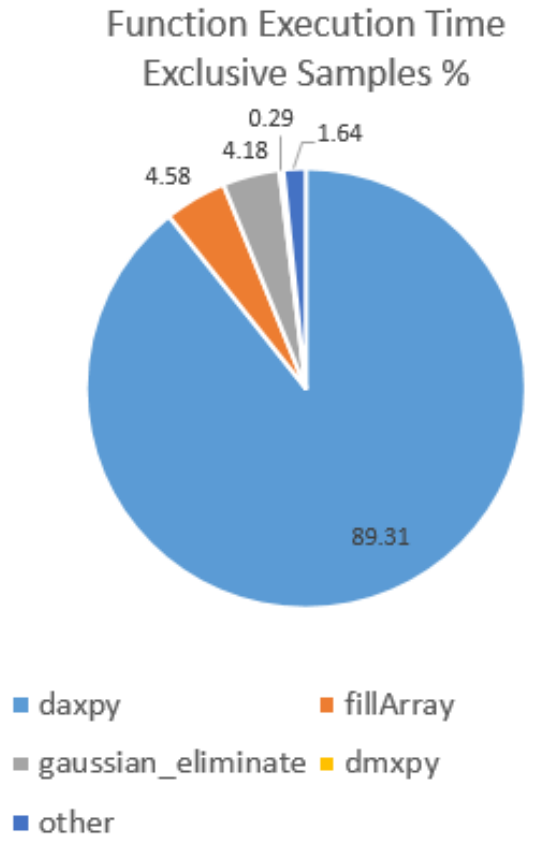

Figure 2: ***Total program execution*** -

Listing 1: *daxpy Code*

```
1 void (int n,double scaler, double *dx, double *dy, int offset) {
2    double *const y = &dy[offset];
3    double *const x = &dx[offset];
4    for (int i = 0; i < n; ++i) {
5       y[i] += scaler * x[i];
6    }
7 }
```

Listing 2: *Simd daxpy Code*

```
1 void (int n, double scaler, double *dx, double *dy,int offset) {
2   if ((n <= 0) || (scaler == 0)) {
3      return;
4   }
5   double *const y = &dy[offset];
6   double *const x = &dx[offset];
7
8   const __m256d scalers = _mm256_set1_pd(scaler);
9   const int remainder = n % 4;
10  const int nm1 = n - 3;
11
12  for (int i = 0; i < nm1; i += 4) {
13     // load X
14     const __m256d xs = _mm256_loadu_pd(&x[i]);
15     // load y
16     __m256d ys = _mm256_loadu_pd(&y[i]);
17     // mutliply X by scalers, add to Y
18     ys = _mm256_add_pd(ys, _mm256_mul_pd(xs, scalers));
19     // load back into y
20     _mm256_storeu_pd(&y[i], ys);
21  }
22
23  for (int i = n - remainder; i < n; ++i) {
24     y[i] += scaler * x[i];
25  }
26 }
```

Simd Daxpy As no increase of performance could be gained by using more threads, Simd instructions were investigated. Daxpy processes N amount of numbers from the two input arrays, using a loop. The numbers to be processed are sequentially laid out in the input arrays, so the logic to convert the loop from processing one item at a time, to multiple, was a simple task. Additional lines of code were unavoidable, e.g if the size of N was not divisible by the amount of items that the loop processes in one pass, then the remainder would have to be processed at the end.
Simd instructions require aligned memory, therefore the numbers form the input arrays had to be loaded into special aligned containers. This could be avoided if the whole program was converted to use aligned data arrays.

Simd Daxpy Results Both 128bit and 256bit (Largest supported by available hardware) Simd versions of Daxpy were implemented and tested. The 128bit version provided no performance gains, due to the overhead of converting input data to aligned data. 256 bit instructions however provided a significant increase in performance, but only when used with an N parameter greater than around 100. As seen in Figure 3.

The Collumn loop Moving back up the call stack to the code area that calls Daxpy, is the loop within the GE function that loops through each column. The function of this loop is to set-up the array pointers to send to Daxpy, and swap the pivot row elements to the top. Fortunately iterations of this loop could be run independently as each column relies only on the data within the very first column, which has already been processed.

OpenMp Collumn loop Panellising the Collumn loop with OpenMP provided positive results and a large boost to performance. While running threads witin daxy proved unsecsffuly, running threads at the collumn loop level which would each run their own copy of Daxpy simultaneously, proved highly beneficial to processing time (Figure 4, Figure 5, Table 1)

MDaxpy and Merging Loops An approach taken by similar implementations of the algorithm, is to merge the Column loop and Daxpy into one function (commonly named "MDaxpy"), by either moving the row swap stage to it's own dedicated loop, or by including it into the logic of Daxpy. Practically, this approach does reduce the total number of instructions executed per loop, while the number of loop iterations stays the same. This approach was investigated, but provided no measurable difference in performance while incurring a cost to code complexity, so this line of code restructuring was discarded.

The Pivot loop

3 Results

Different versions of the algorithm were run using a system containing an Intel i7-4790K cpu at 4GHz. Each configuration was run 1000 times, the time taken for each section of the code was taken for each run and the mean average is shown in the following results.
Looking at Table 1 and Figure 4, The highest speed-up achieved was 71% using 256bit Simd and 7 threads. Looking at the trend in general, speed-up increase rapidly drops off after 4 threads. This could be due to the cpu used in this test only having 4 physical cores, hyperthreaded to 8 logical cores.
As previously mentioned 128bit simd instructions do not contribute to ay significant performance gains, while 256bit Simd keeps a constant lead in speedup. Overall the benefits of Simd were vastly overshadowed by the method of multi-threading the Column loop.

4 System utilisation

Comparison of the cpu utilisation, core allocation and thread status for single threaded, 4 threads and 8 threads are given below. For each of these results, the 256bit Simd versions were used. For clarity, only the first 6 runs are shown for each figure.

Single threaded No anomalies were observed here, the Main Thread stays constantly busy(Figure 9). The main thread is reallocated to different physical cores multiple times during execution, this did not have an impact on processing time.

4 threads A near perfect 50% utilisation was achieved when using 4 threads, with a consistent processing time for each run. The 4 threads were kept constantly busy, and while the core allocation allocation did change frequently, there was enough spare processing resources available to keep the relocation from effecting performance.

8 threads Using the full 8 threads available on the cpu provided a more interesting set of results. The first run struggled to get above 50% cpu utilisation, with most threads in a waiting state. Looking at the core allocation for this first run, cores 2, 4, and 6 were running two threads at once. This sub-optimal layout resulted in a longer than usual first run.
Looking at the second run, the core allocation is vastly improved, but not perfect as Core 0 is still running 2 threads while core 7 is idle. The third run moves to a perfect allocation almost immediately, which ever subsequent run continues.
This could be caused by a number of factors, but the primary cause is most likely the operation system's scheduler and the CPUs internal programming. Asking for 100% usage of all available cores on a system means that there is nothing left over for other applications and background tasks. This could also be the reason behind 7 threads being the optimal solution, as it leaves 1 logical core for background tasks.

5 Conclusions

Computation time Both Implementations of the Clarke-Wright algorithms produced expected results, with the parallel version producing larger and fewer routes. As for the time taken to calculate, the performance is roughly equal. The discrepancies shown in Figure **??** when the amount of customers increases beyond 800 is possibly due to optimisations carried out by the Java virtual machine. The total operations carried out is roughly the same for each algorithm, however the arrays are accessed and modified at different times, this is a possible cause for the difference in processing time.

References

DONGARRA, J. J., LUSZCZEK, P., AND PETITET, A. 2003. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience 15*, 2003.

6 Appendix

Listing 3: *Gaussian eliminate Code*

```
1  int (double **a, int n, int *ipivot) {
2    // Pointers to columns being worked on
3    double *col_k;
4    int nm1 = n - 1;
5    int info = 0;
6
7    if (nm1 >= 0) {
8      int kp1, l;
9      for (int k = 0; k < nm1; ++k) {
10       // Set pointer for col_k to relevant column in a
11       col_k = &a[k][0];
12       kp1 = k + 1;
13
14       // Find pivot index
15       l = indexOfLargestElement(n - k, col_k, k) + k;
16       ipivot[k] = l;
17
18       // Zero pivot means that this column is already triangularized
19       if (col_k[l] != 0) {
20         double t;
21         // Check if we need to interchange
22         if (l != k) {
23           t = col_k[l];
24           col_k[l] = col_k[k];
25           col_k[k] = t;
26         }
27
28         // Compute multipliers
29         t = -1.0 / col_k[k];
30         scaleVecByConstant(n - kp1, t, col_k, kp1, 1);
31
32         // Row elimination with column indexing
33         #pragma omp parallel for
34         for (int j = kp1; j < n; ++j) {
35           // Set pointer for col_j to relevant column in a
36           double *col_j = &a[j][0];
37
38           double t = col_j[l];
39           if (l != k) {
40             col_j[l] = col_j[k];
41             col_j[k] = t;
42           }
43           daxpy(n - kp1, t, col_k, col_j, kp1);
44         }
45       } else{
46         info = k;
47       }
48     }
49     ipivot[n - 1] = n - 1;
50     if (a[n - 1][n - 1] == 0) {
51     info = n - 1;
52   }
53
54   return info;
55 }
```

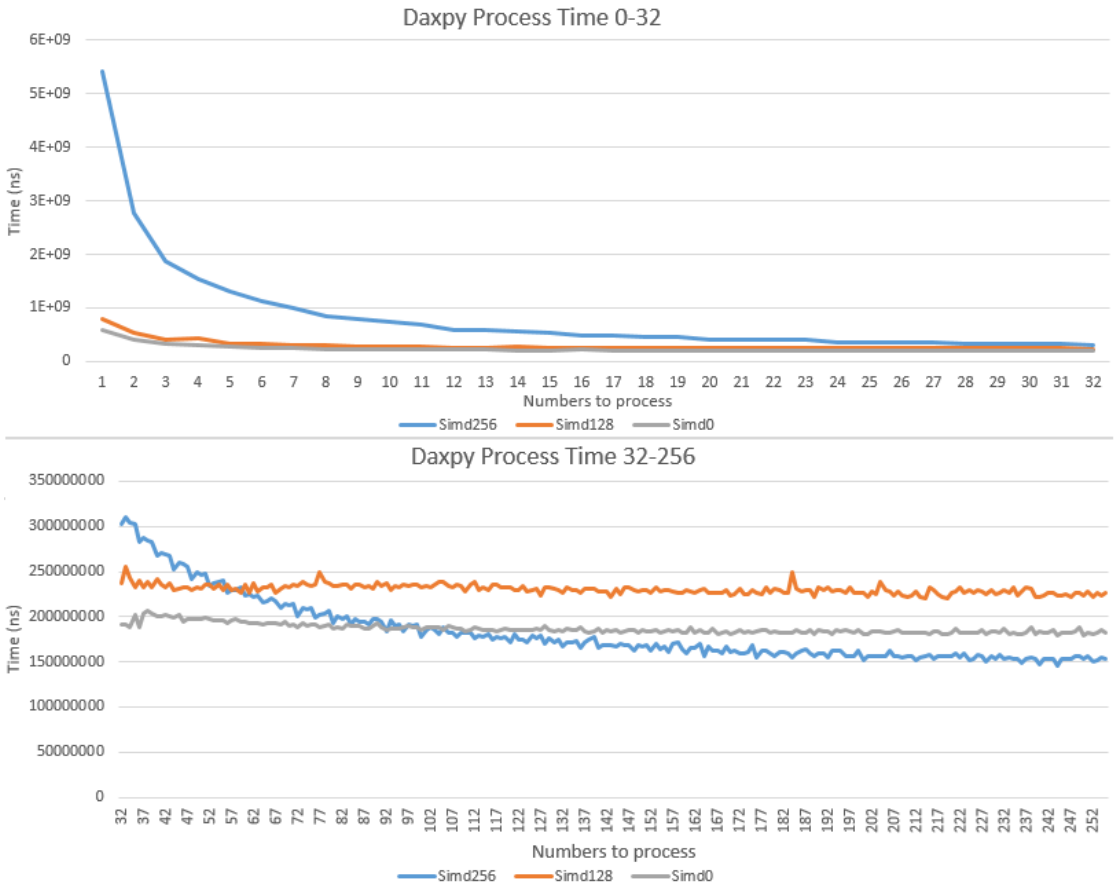

Figure 3: ***Daxpy Simd Comparisons*** *- Time taken to process 10'000 numbers, 10'000 times Each subsequent call to Daxpy increases the amount of number to calculate at once.*

Name	Allocate Memory (ms)	Create Input Numbers (ms)	gaussian eliminate (ms)	Solve (ms)	Validate (ms)	Total Time (ms)	Total Speedup	Speedup (With Simd)*
Threads: 1 No Simd	0.48	5.22	157.27	0.69	5.27	168.93	0%	0%
Threads: 1 Simd128	0.45	5.00	152.68	0.67	5.06	163.85	3%	0%
Threads: 1 Simd256	0.44	5.06	142.04	0.67	5.14	153.34	9%	0%
Threads: 2 No Simd	0.42	5.48	78.76	0.68	5.52	90.86	46%	46%
Threads: 2 Simd128	0.46	5.36	85.35	0.66	5.49	97.32	42%	41%
Threads: 2 Simd256	0.45	4.82	72.86	0.63	4.83	83.59	51%	45%
Threads: 3 No Simd	0.45	5.37	63.49	0.68	5.46	75.44	55%	55%
Threads: 3 Simd128	0.46	5.52	60.00	0.66	5.64	72.28	57%	56%
Threads: 3 Simd256	0.46	5.74	52.51	0.58	5.85	65.14	61%	58%
Threads: 4 No Simd	0.44	5.28	42.67	0.62	5.33	54.34	68%	68%
Threads: 4 Simd128	0.43	5.72	41.51	0.54	5.78	53.97	68%	67%
Threads: 4 Simd256	0.48	5.80	39.61	0.55	5.91	52.36	69%	66%
Threads: 5 No Simd	0.46	5.41	50.89	0.67	5.50	62.93	63%	63%
Threads: 5 Simd128	0.45	5.20	49.49	0.67	5.24	61.05	64%	63%
Threads: 5 Simd256	0.46	5.11	46.46	0.60	5.24	57.86	66%	62%
Threads: 6 No Simd	0.50	5.80	44.44	0.75	5.93	57.41	66%	66%
Threads: 6 Simd128	0.48	5.53	43.04	0.69	5.66	55.41	67%	66%
Threads: 6 Simd256	0.49	5.55	41.63	0.65	5.72	54.04	68%	65%
Threads: 7 No Simd	0.53	6.00	40.21	0.75	6.18	53.66	68%	68%
Threads: 7 Simd128	0.52	5.40	40.02	0.73	5.55	52.22	69%	68%
Threads: 7 Simd256	0.55	5.58	35.69	0.68	5.77	48.28	71%	69%
Threads: 8 No Simd	0.51	5.01	43.25	0.80	5.17	54.75	68%	68%
Threads: 8 Simd128	0.53	5.48	43.44	0.78	5.73	55.96	67%	66%
Threads: 8 Simd256	0.51	5.47	38.64	0.71	5.72	51.05	70%	67%

Table 1: *Results of all tests*
**Speed-up with Simd, compares times against the Simd equivalent 1 Thread run*

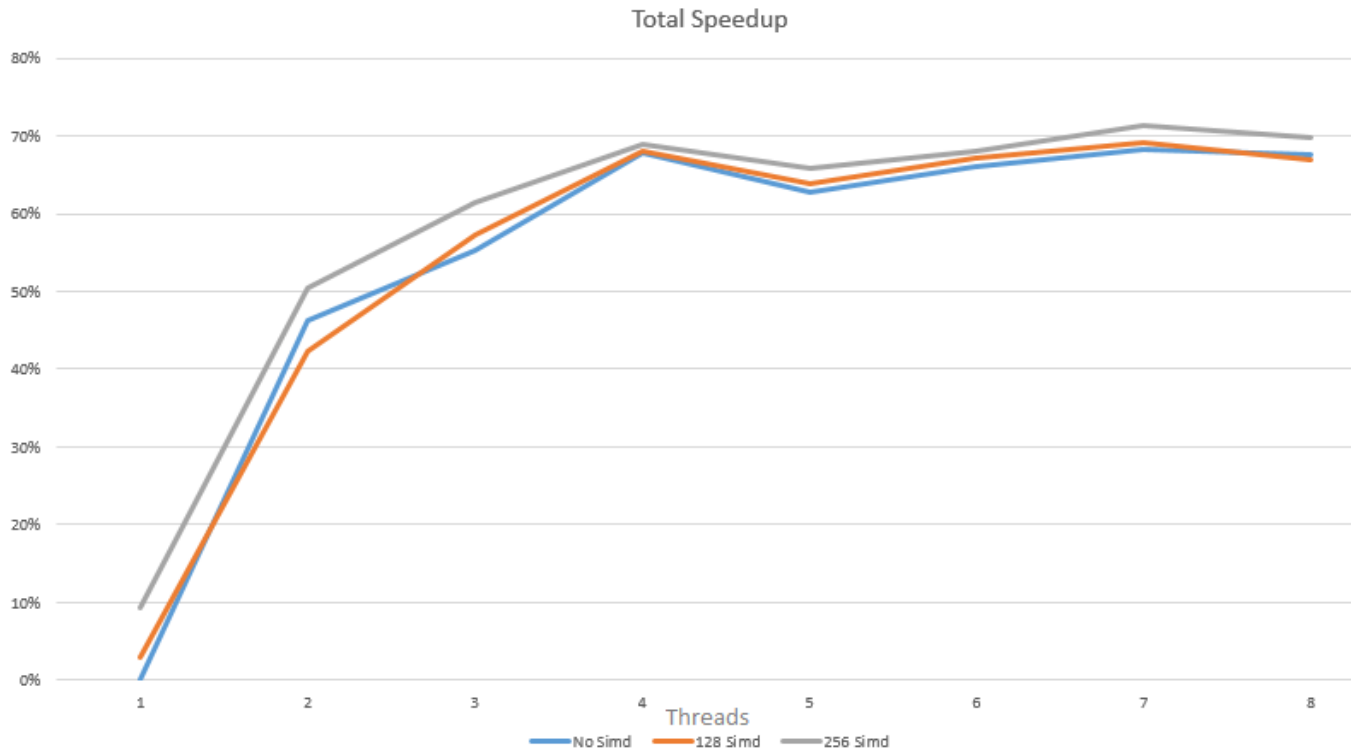

Figure 4: ***Total Speed-up percentage, for each number of threads*** -

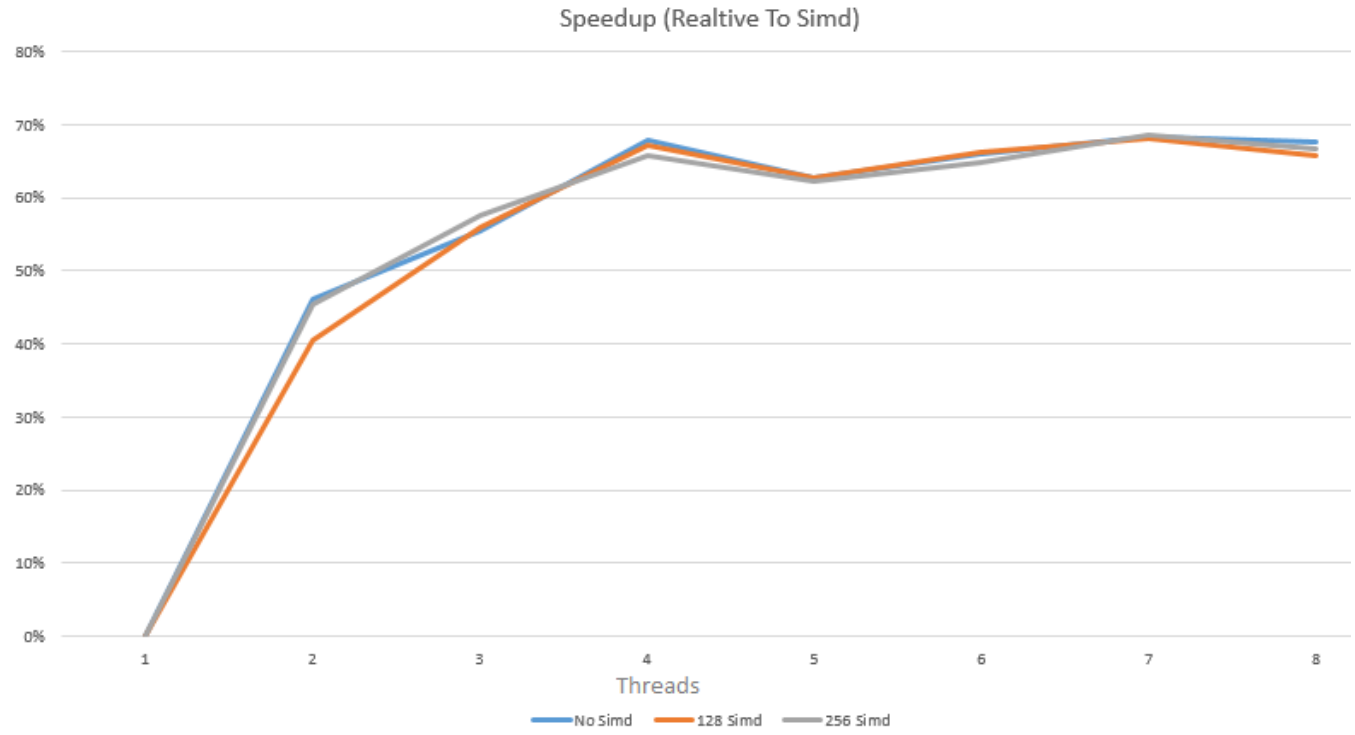

Figure 5: ***Total Speed-up (Relative to baseline Simd) percentage, for each number of threads*** -

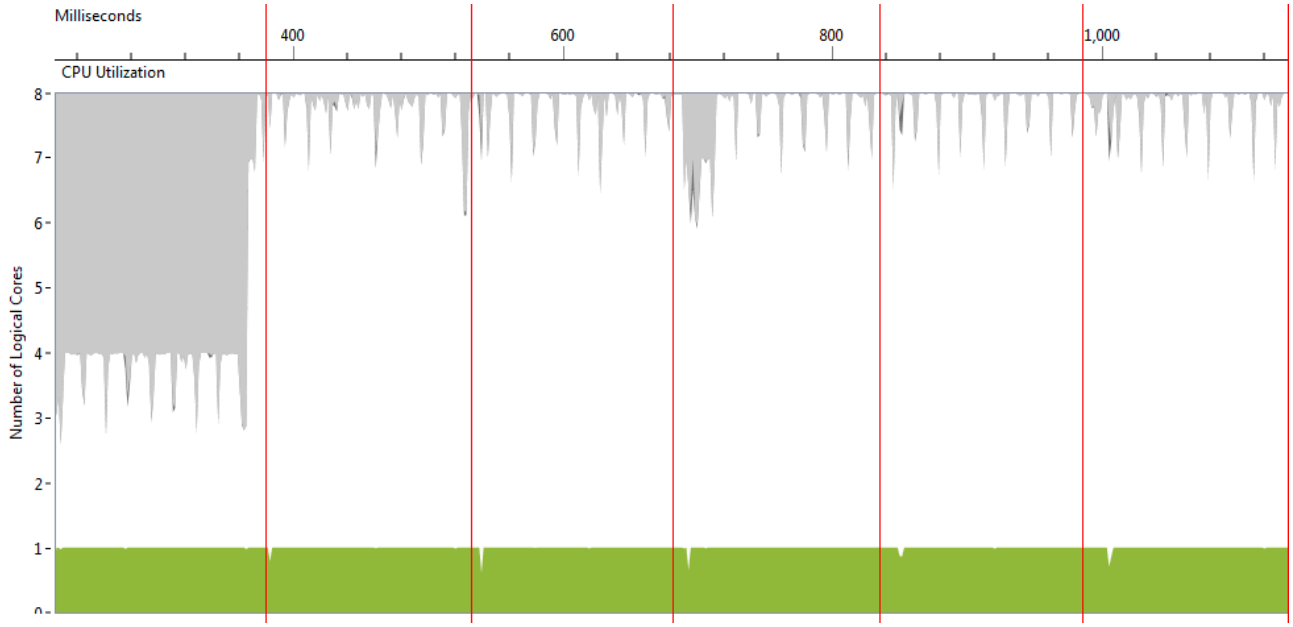

Figure 6: ***Single Threaded, 6 runs, simd256 Daxpy*** *- Overall system CPU utilisation*

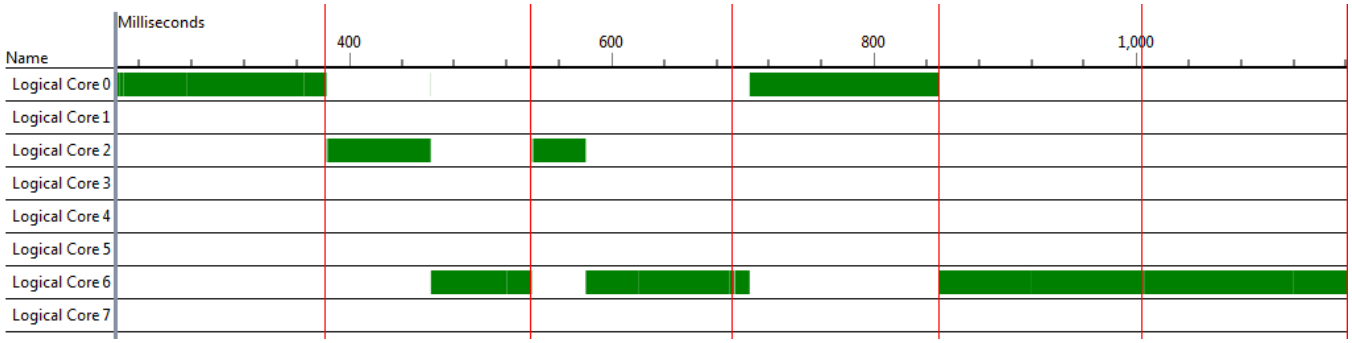

Figure 7: ***Single Threaded, 6 runs, simd256 Daxpy*** *- Thread to CPU Core allocation*

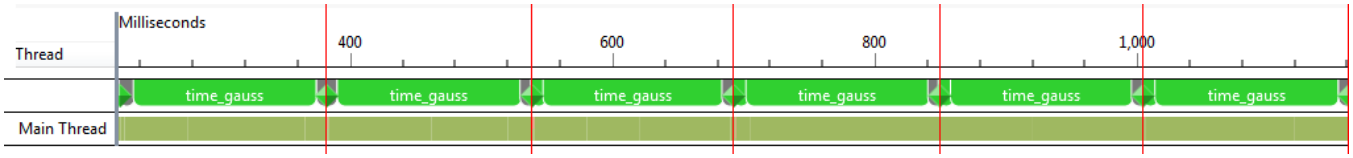

Figure 8: ***Single Threaded, 6 runs, simd256 Daxpy*** *- Thread Status*

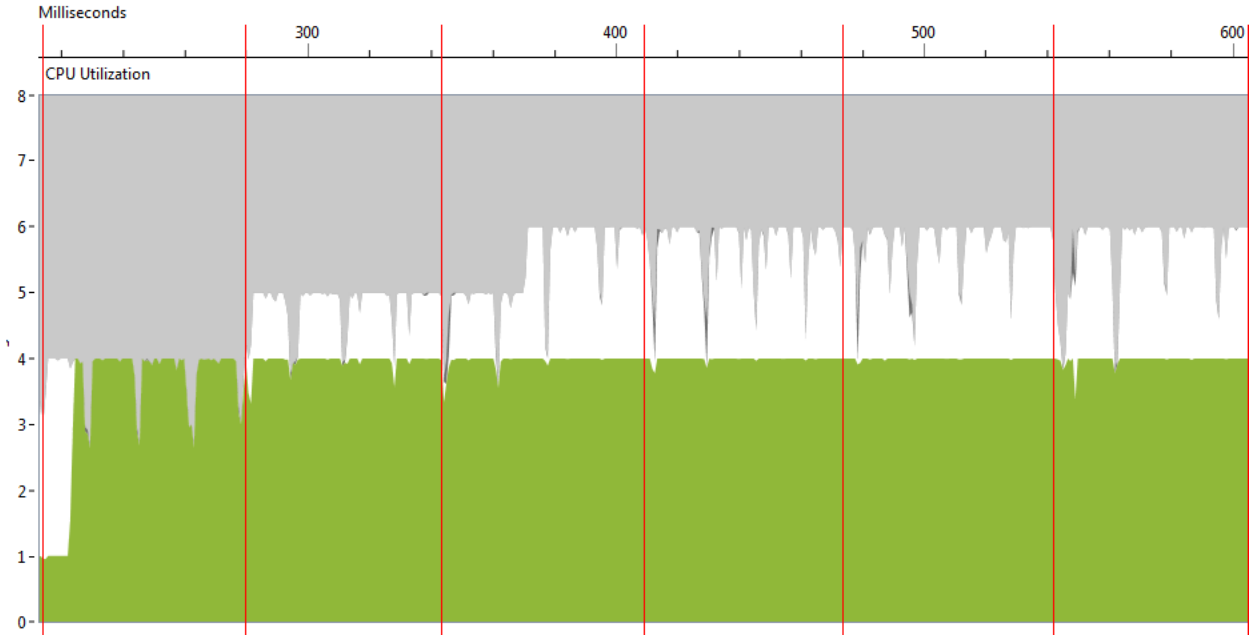

Figure 9: ***4 Threads, 6 runs, simd256 Daxpy*** *- Overall system CPU utilisation*

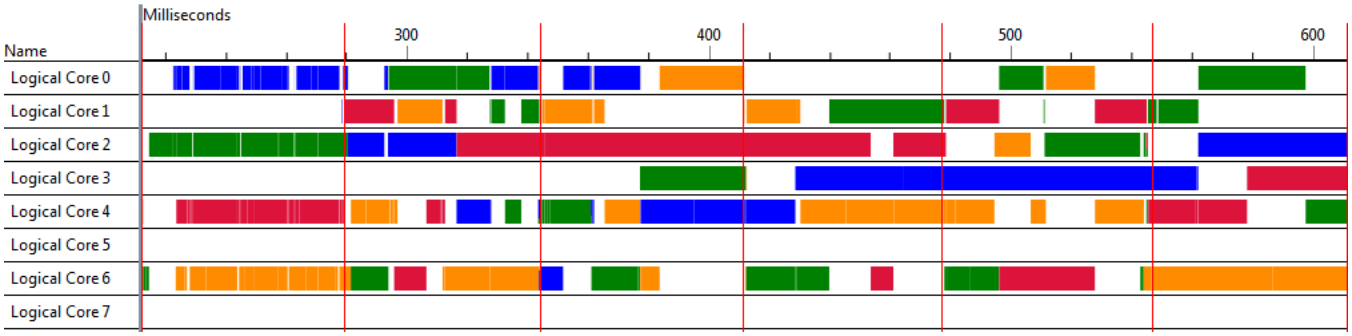

Figure 10: ***4 Threads, 6 runs, simd256 Daxpy*** *- Thread to CPU Core allocation*

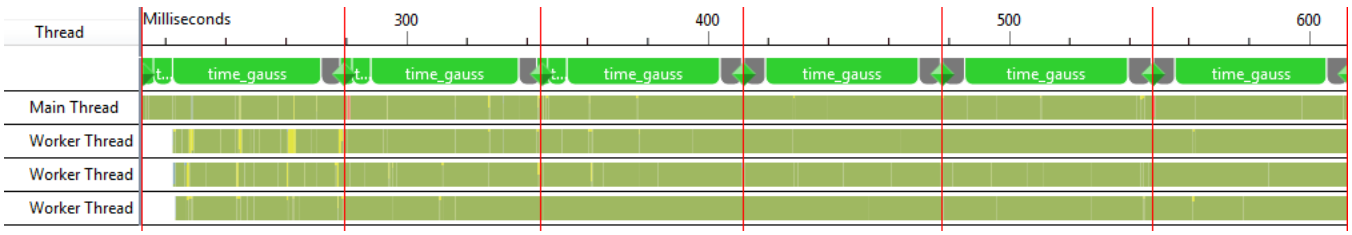

Figure 11: ***4 Threads, 6 runs, simd256 Daxpy*** *- Thread Status*

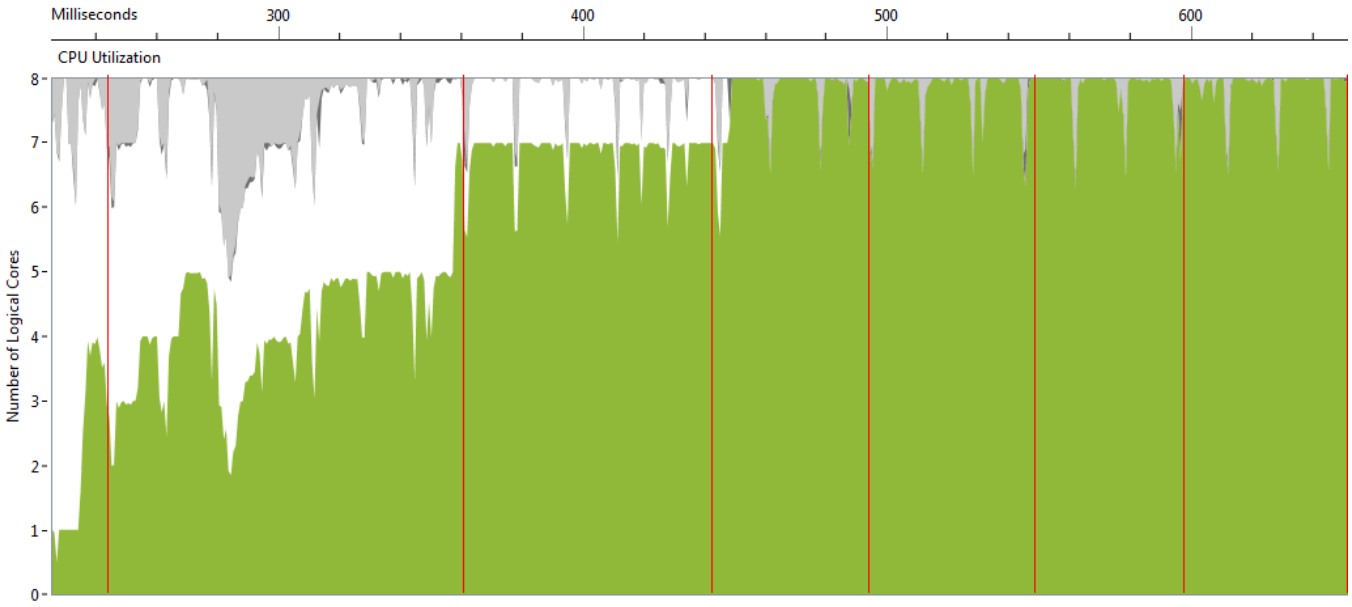

Figure 12: ***8 Threads, 6 runs, simd256 Daxpy*** *- Overall system CPU utilisation*

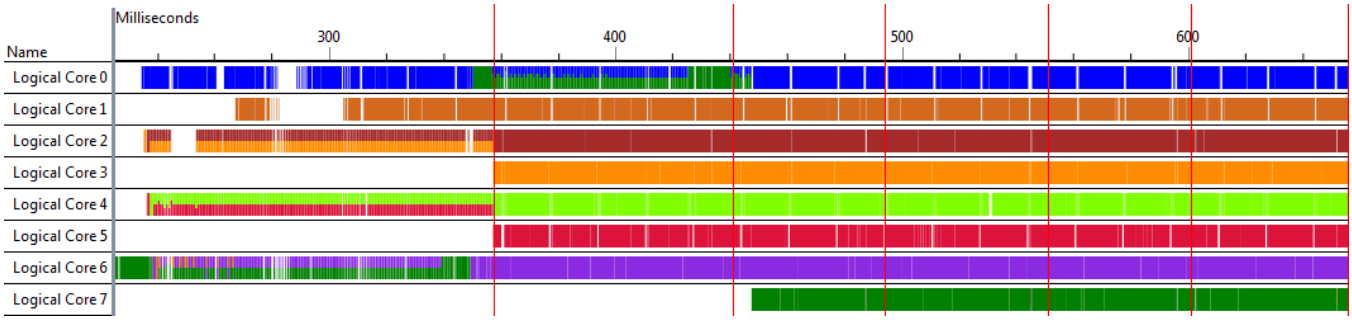

Figure 13: ***8 Threads, 6 runs, simd256 Daxpy*** *- Thread to CPU Core allocation*

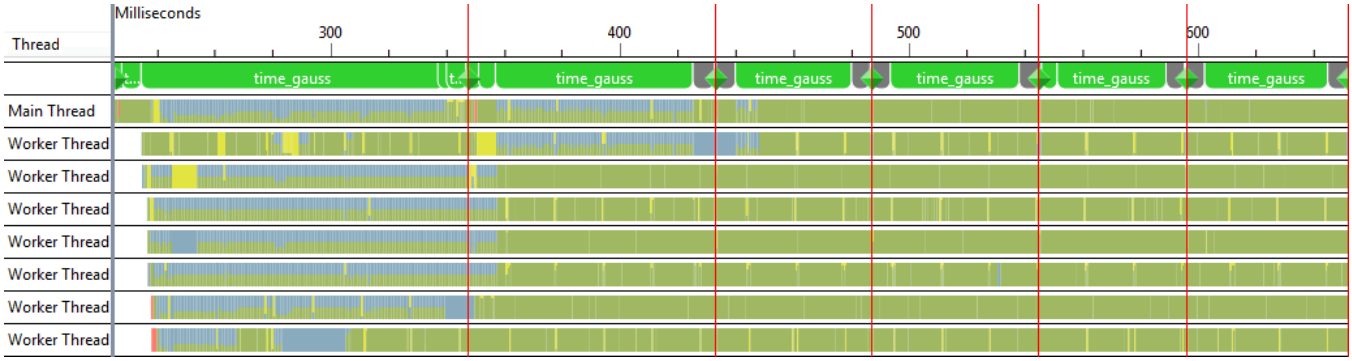

Figure 14: ***8 Threads, 6 runs, simd256 Daxpy*** *- Thread Status*