UNIVERSITY OF AMSTERDAM

Simulating Heat Dissipation

An in-depth description of a sequential and vectorized implementation and experiments

Kian Paimani / 11622849
Jonas Theis / 12142964

Universiteit van Amsterdam
Vrije Universiteit Amsterdam
Course: Programming Multi-core and Many-core Systems
Course instructor: dr. Clemens Grelck

20th August 2018

1 Introduction

In this report we examine a sequential and vectorized parallel version of the heat dissipation over a cylinder surface problem. The model we use represents the cylinder's surface as a temperature matrix t of $N \times M$. Each point in the grid describes a temperature value on the surface and is specified as a floating point number. The cyclic dimension of the cylinder is depicted by the M columns and the N rows represent the other dimension. There is a condition for boundary points in the latter dimension: each boundary grid point is supposed to have a neighbor initialized to the same value, which stays constant over time.

Updating temperature values is an iterative process, meaning that the value at a grid point is computed based on the weighted average of the previous iteration's value, the previous iteration's values of the 4 direct neighbours and 4 diagonal neighbours. The joint weight of the 4 direct neighbours is $\frac{\sqrt{2}}{\sqrt{2}+1}$ and of the diagonal neighbours $\frac{1}{\sqrt{2}+1}$. Furthermore, there is a constant conductivity coefficient for each grid point resulting in a $N \times M$ coefficient matrix. This coefficient indicates the weight of the previous iteration's value. The simulation finishes either after a specified number of iterations or when it has sufficiently converged.

In this assignment, we examine different aspects of two main implementations of this process:

- **Sequential**: version where the algorithm is implemented in C and only automatic compiler flags are tweaked to reach the best performance. Consequently, the sequential version has evolved in different versions in a way that its *performance* becomes the best and its *readability* is not important.
- **Vectorized/SIMD** version where the same algorithm is implemented with manual attempts to gain performance through Intel SSE vectorization instead of auto-vectorization of compilers.

This report first discusses in section 2 the implementation of the two versions, including different optimizations applied, and their respective challenges/dilemmas. Then, in section 3 the conducted experiments are described and evaluated. Finally, section 4 concludes the measured results and compares them to the expected results.

2 Implementation and Optimization

In this section we describe the implementation details of the two versions of the heat dissipation problem. In each version, we describe the general approach and optimizations applied to improve the performance. Furthermore, we will go through some of the compiler optimizations that can be applied to improve performance. Respective results of each sub-section with respect to its optimization will be demonstrated in section 3.

2.1 Sequential Implementation

In this section we will briefly describe the sequential implementation of the heat dissipation problem. Next, different modifications and their respective effect on the performance are explained.

Without diving into the details of the code, the following points demonstrate the gist of our sequential implementation:

- The main flow of the program consists of three iterative loops; An outer *while* loop that holds the iteration of updates and two inner *for* loops that update each grid point in the matrix.
- Two copies of the temperature matrix are kept (both of which are stored as a contiguous memory space). One represents the current value during the update and the other one the previous values. Consequently, at the end of each iteration the matrix for previous values is replaced (pointed to) with the newly built matrix.
- For calculating intermediary and final results there is a helper function to calculate the minimum, maximum and average temperature of a given matrix.

While any code with the structure explained above would yield a correct output for this problem, their performances can be quite different. We have seen in our experiments that numerous small details can have noticeable effect on the performance. The following points will elaborate some of these optimization details.

- **Code Motion - Share common expression** While it is mostly encouraged to use readable, expressive and clear statements in programming, this rule can sometimes lead to unnecessary computation. One simple, yet frequent example of this array indexing. A *readable* way of doing this would be similar to the following snippet.

```
for (int row_idx = 0; row_idx < N; row_idx++) {
  for (int col_idx = 0; col_idx < M; col_idx++) {
     matrix[row_idx*M + col_idx] = computed_new_value;
    }
}
```

  It can be clearly seen that the computation of $rowidx * M$ in each iteration of inner loop is redundant. This snippet can be easily transformed to the following optimized version.

```
int current_row_idx;
for (int row_idx = 0; row_idx < N; row_idx++) {
  current_row_idx = row_idx*M
  for (int col_idx = 0; col_idx < M; col_idx++) {
     matrix[current_row_idx + col_idx] = computed_new_value;
```

```
        // update diff etc.
    }
}
```

Furthermore, since the expression $current\_row\_idx + col\_idx$ will be used again at the end of the inner loop to calculate the *diff*, it is computationally more efficient to store it once and re-use it again.

```
int current_row_idx, current_cell_index;
for (int row_idx = 0; row_idx < N; row_idx++) {
  current_row_idx = row_idx*M
  for (int col_idx = 0; col_idx < M; col_idx++) {
     current_cell_index = current_row_idx + col_idx
     matrix[current_cell_index] = computed_new_value;

     // update diff etc.
     int diff = matrix[current_cell_index] - o_matrix[current_cell_index]
  }
}
```

- **Avoiding Optimization Blockers** An ideal loop is one that does *exactly* the same set of operations a finite number of times. One common programming structure that is against this rule, despite being used frequently, is conditional statements inside the loop. In the heat simulation, this can happen due to the fact that the memory access pattern of updating first and last column is different than the rest. More precisely, since the first column is adjacent to the last one, the first row must be fetched as the neighbor index. While duplicating the first column after the last one is a fix for this problem, an even more optimized, yet slightly *uglier*, solution is to duplicate the code of cell update of the first and last column to the outside of the loop and do it separately. The following code demonstrates this point.

```
for (int row_idx = 0; row_idx < N; row_idx++) {
  for (int col_idx = 1; col_idx < M-1; col_idx++) {
     matrix[row_idx*M + col_idx] = computed_new_value;
  }
  // do cell update for col_idx = 0
  matrix[row_idx*M] = computed_new_value;
  // do cell update for col_idx = M-1
  matrix[row_idx*M + M-1] = computed_new_value;
}
```

Furthermore, all function calls, despite making the code more readable, are removed from the main loop and their return expression values are used in-place. Note that an alternative way for large functions is to define them as $inline function$.

- **Reduction in Strength** (using bitwise arithmetic) To reduce computation time it is common

practice to use less expensive operations whenever possible. For example, a bit operation is cheaper than a floating point multiplication. For that reason the calculation of the absolute difference between new and old value at every point in the matrix was optimized using a bitwise *AND* operation as displayed in the listing below. The *IEEE 754 double-precision binary floating-point format* states that the first bit of a *double* represents the sign of the number. With the help of a bitmask and the bitwise *AND* operation this bit can be set to zero, which results in the absolute value of that number.

```
union Abs {
  double d;
  long i;
};
const long abs_bitmask = ~0x8000000000000000;

// calculate absolute diff between new and old value
abs_diff.d = t_surface_index - temp_index;
abs_diff.i = abs_bitmask & abs_diff.i;
```

- **Memory Access** Aside from memory access patterns, one should note that accessing it is also more expensive via a complex structure (*i.e.* an array) compared to a simple structure like normal variables. Consequently, all memory accesses are also reduced to the minimum possible. In all of the above code snippets, there is a major flaw because the new value written immediately to $matrix[rowidx * M + colidx]$ is read again at the end of the loop to be compared with the old value to calculate the *diff*. Hence, a better way would be to store the new value in a variable, re-use it as many times as needed and finally write it to the memory at the very end of the loop. The fixed version of the above snippet would then be:

```
int current_row_idx, current_cell_index, temp_value;
for (int row_idx = 0; row_idx < N; row_idx++) {
  current_row_idx = row_idx*M
  for (int col_idx = 0; col_idx < M; col_idx++) {
     current_cell_index = current_row_idx + col_idx
     temp_value = computed_new_value;

       // update diff etc.
       int diff = temp_value - o_matrix[current_cell_index]

       // write to memory once
       matrix[current_cell_index] = temp_value
    }
}
```

Furthermore, there was an attempt to reduce memory calls to the big matrix *t_surface* and therefore possibly decrease cache misses by prefetching the previous, current and next row. Unfortunately, this turned out to be slower due to the fact that the data needed to be copied.

```
double this_row[M];
double row_up[M];
double row_down[M];

// for every row
memcpy(this_row, t_surface+this_row_start_idx, sizeof(double)*M);
memcpy(row_up, t_surface+row_up_start_idx, sizeof(double)*M);
memcpy(row_down, t_surface+row_down_start_idx, sizeof(double)*M);
```

2.2 Compiler Vectorization

Aside from code optimizations, compiler options are also important to reach a good performance. In this report, aside from general optimizations, we will focus on compiler flags that enable *Auto Vectorization*. Using these flags, the compiler will automatically try to identify possible candidates for vectorization and convert them to SIMD code. Since vectorization is a pure functionality of the CPU hardware, it is dependent on the CPU vendor and generation. The DAS4 nodes that have been used for this experiment are equipped with *Intel Xeon Processor E5620*. This generation of CPUs supports up to the *SSE* variant of vectorization. In *SSE*, each register can store up to 128 bits of data which can be equal to two *double* or four $float$ values. Furthermore, the version of the compiler is also important since each behave differently. While the DAS4 nodes are loaded with *gcc4* by default, we have also included *gcc6* and reproduced some tests with this newer version. Another benchmark is the comparison of different compilers. While *gcc* might be the most renowned or well-known C compiler, another variant is Intel's *icc*. Since the vectorization operation is CPU dependent and is implemented in the hardware by the vendor, a portion of the tests regarding auto-vectorization of the code have also been compiled and using *icc*.

Both compilers will not apply optimizations on-fly without any instructions. Instead, compilers flags are used to enable different levels of optimization. Due to the high number of flags, the standard *-Ox* is used to enable groups of optimization flags. The full list of flags that are enabled in each optimization level are listed in [1]. All tests are executed with all four possible levels of optimization.

One of the optimization that will be enabled by *-O2* (and higher) levels of *gcc* is *ftree-loop-vectorize* and *-ftree-slp-vectorize*. These flags will attempt to perform the same operation as we will attempt to do in our SIMD implementation (see section 2.3). Furthermore, albeit these flags are used on the sequential versions, the *SIMD* version is always compiled with *-fno-tree-vectorize*. This flag will disable auto-vectorizing loops. Hence, we can clearly compare the vectorization performed by the compiler and by our own implementation.

The reader should also note that the mentioned flags are only regarding loop vectorization. Our examination of the assembly codes of both *gcc* and *icc* have shown that despite both compilers fail at auto-vectorizing the loops of the sequential version, various SSE assembly commands are added

by the compiler to all versions. As an example, the following log shows a portion of the assembly code generated by *gcc-5 -O2 -S -std=c99* compile flags.

```
L21:
   movapd  %xmm8, %xmm4
   jmp  L12
```

This clearly shows that *movapd* is used in the assembly code of the compiler and it is an Intel specific intrinsic command for loading two double values into a register.

2.3 SIMD Implementation

As described in the previous section the tests were conducted on the DAS4, which supports *SSE* (128 bits). Since the computation of heat dissipation uses double precision floating point numbers, a register can fit 2 double values.

The SIMD implementation is based on the sequential version. Therefore, the basic application architecture is the same as explained in section 2.1. The only change is that the inner *for* loop advances 2 instead of 1 ($col+=2$). This is possible due to the fact that one SIMD register can fit 2 double values and the implementation computes both at the same time.

Figure 1 displays this behaviour. First, the two values at the current index position are loaded into the register $v2$. Then the final values of the direct neighbours and diagonal neighbours of each of the points are calculated and loaded into respective registers $v3$ and $v4$. As an intermediate step registers $v2$ and $v3$ are added together, the result is added to $v4$ and the overall result gets stored in $v1$. All these operations are executed in parallel for both points in the matrix. Similarly to the sequential version, the respective absolute difference between old and new value is calculated using a bitwise *AND NOT* operation.

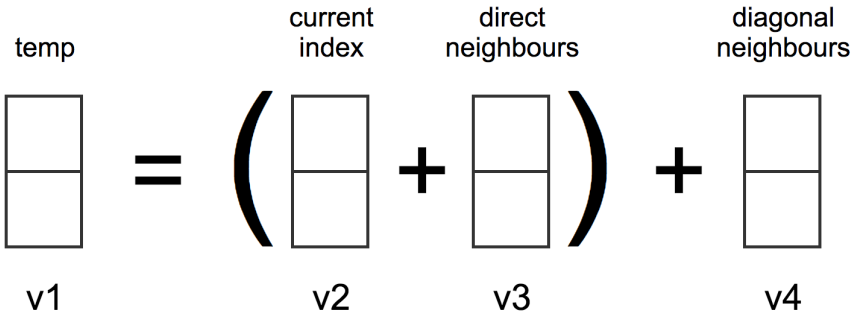

Figure 1: High-level view of the SIMD implementation within one iteration in the inner *for* loop

Aside from the *SIMD* pattern explained above, we have also tried to vectorized the addition of 8 neighbors of each cell. This modification is not included in the main branch of our codebase because all benchmarks immediately showed that using only 128 bit registers, this pattern is not efficient. In other words, the time spent to load all neighbors into registers two by two, add them together and storing them in the original place is more significant than the time saved while doing two additions at the same time. This argument can be further justified by noting that addition itself is not a computationally expensive operation, hence vectorizing it with a high cost of load/store operations is inefficient.

3 Evaluation

This section first describes the experimental setup and the procedure of taking measurements. Then, the results using the presented setup are displayed and analysed.

3.1 Experimental setup

All the experiments were conducted on the DAS4 VU cluster. Each node consists of a dual quad-core, 2x hyperthreading processor with 2.4GHz, 24 GB memory and 2*1TB HDDs interconnected with Gigabit Ethernet and InfiniBand. Measured was only the time of the actual computation without eventual initial time for reading and preprocessing. Furthermore, floating point operations per second (FLOPs) are calculated based on the measured time and the size of the problem.

Each run was executed five times, the output was compared to the sequential version to make sure it is correct and the resulting time and FLOPs are the fastest of the five runs. The exact parameters for each run are specified with the test results.

3.2 Results

In this section we will go through the experiments conducted and evaluate them. We will start by examining *gcc* and *icc* for various input sizes. Next we will examine two different versions of *gcc* and *icc* and compare them together. Finally, we will benchmark our own implementation of loop vectorization.

Table 1 shows the result of our optimized sequential version with various input sizes. As seen in the header, both compilers are used with the maximum optimization flag. This ensures that the compiler is attempting to vectorize the code and, as mentioned before, use various SSE intrinsics. As can be seen in the table, the operation count per flop in different sizes is almost equal, which shows that different memory mappings does not have a significant effect on the outcome. Furthermore, if we go into the details, it can be seen that only for **relatively large square matrices** *icc* is performing slightly better. In all other cases, *gcc* is the dominant with a small margin.

		seq gcc -O3		seq icc -O3	
N	**M**	**Time in s**	**GFLOPs**	**Time in s**	**GFLOPs**
100	100	0.0371	1.63	0.0380	1.59
1000	1000	4.1878	1.44	4.0682	1.49
2000	2000	17.2260	1.40	16.2505	1.49
100	50	0.0188	1.61	0.0194	1.56
100	200	0.0755	1.60	0.0784	1.54
100	1000	0.3771	1.60	0.3896	1.55
100	2000	0.7571	1.60	0.7824	1.55
200	100	0.0767	1.58	0.0800	1.51
1000	100	0.3833	1.58	0.3957	1.53
2000	100	0.7692	1.57	0.7852	1.54

Table 1: Result set using *gcc4* and the following parameters: $I = 500$, $E = 0.01$

		seq gcc -O0		seq gcc -O1		seq gcc -O2		seq gcc -O3	
N	**M**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**
2000	2000	66.2905	0.365	26.6646	0.908	17.2801	1.40	17.2260	1.40
100	2000	3.1644	0.382	1.2345	0.98	0.7605	1.59	0.7571	1.60
2000	100	3.2582	0.371	1.2401	0.976	0.7705	1.57	0.7692	1.57

Table 2: Result set using *gcc4* and the following parameters: $I = 500$, $E = 0.01$

Furthermore, Table 2, 3 and 4 demonstrate a more comprehensive comparison of all three compiler that we have used. The goal of these measures is to see which performs better with auto-optimization. Multiple facts can be inferred from these tables:

- *gcc6* performs better than *gcc4*. The difference becomes more significant in columns that show the execution with a high optimization level.
- Both row and column wise matrices have almost the same result in all three compilers. Once more, we can argue that as long as the matrices are stored in a contiguous memory space, their shape has a small impact.
- Similar to the conclusion in Table 1, we can see that *icc* can only outperform *gcc4* in large square matrices. The same argument does not hold for *gcc6* and it is clearly the best in terms of optimization flags.

Table 5 and 6 enumerate the results of our manually vectorized implementation. All of the numbers in this table are extraction with executions that are compiled with *-ftree-no-vectorize*, which disables compiler vectorization of loops. Similar to the sequential version, it can be seen that the dimension of the matrix is not significant in the results. Furthermore, it can be seen that our vectorized version, compiled with *gcc6*, with the rest of the optimizations being enabled, can outperform *gcc4* and *icc*. Albeit, this argument becomes significant in optimization levels of

		seq gcc-6 -O0		seq gcc-6 -O1		seq gcc-6 -O2		seq gcc-6 -O3	
N	**M**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**
2000	2000	65.0182	0.372	24.0632	1.01	15.4555	1.57	15.4538	1.57
100	2000	3.1531	0.384	1.1088	1.09	0.6712	1.80	0.6709	1.80
2000	100	3.1492	0.384	1.1111	1.09	0.6823	1.77	0.6820	1.77

Table 3: Result set using *gcc6* and the following parameters: $I = 500$, $E = 0.01$

		seq icc -O0		seq icc -O1		seq icc -O2		seq icc -O3	
N	**M**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**
2000	2000	65.6362	0.369	17.4026	1.39	16.2783	1.49	16.2505	1.49
100	2000	3.1906	0.379	0.8399	1.44	0.7831	1.55	0.7824	1.55
2000	100	3.1897	0.379	0.8556	1.41	0.7845	1.54	0.7852	1.54

Table 4: Result set using *icc* and the following parameters: $I = 500$, $E = 0.01$

-O1 and higher. As an example, *gcc4* and *icc* reach a maximum GFLOPs rate of 1.40 and 1.49 respectively in the sequential version with 2000 by 2000 input, while the vectorized version peaks at 1.76.

4 Conclusion

The previous section presented and discussed experimental results. The most important findings are:

- There is a noticeable difference between the choice of compiler versions. More significantly, optimization flags can have a significant effect in the performance.
- Recent versions of compilers do a better job at optimizing the code. While our vectorized version could outperform *gcc4* and *icc*, *gcc6* still produced a faster executable for the sequential version.
- Our failed attempt at vectorizing a sequence of additions shows that especially for problems like heat dissipation, where the structure of operations is relatively complex, vectorization should be applied with caution to prevent negative speedup.

SSE intrinsics can be game-changing feature for performance-critical applications. They are

		vec gcc -O0		vec gcc -O1		vec gcc -O2		vec gcc -O3	
N	**M**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**
2000	2000	88.22	0.274	22.79	1.06	17.65	1.37	17.24	1.37
100	2000	4.24	0.285	1.03	1.17	1.18	1.02	1.18	1.02
2000	100	4.26	0.284	1.02	1.18	0.77	1.55	1.22	1.04

Table 5: Result set using *gcc4* and the following parameters: $I = 500$, $E = 0.01$

		vec gcc-6 -O0		vec gcc-6 -O1		vec gcc-6 -O2		vec gcc-6 -O3	
N	**M**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**	**Time**	**GFLOPs**
2000	2000	87.45	0.277	18.61	1.30	13.64	1.77	13.75	1.76
100	2000	4.22	0.287	0.83	1.44	0.58	2.08	0.58	2.06
2000	100	4.24	0.285	0.83	1.44	0.58	2.08	0.57	2.07

Table 6: Result set using *gcc6* and the following parameters: $I = 500$, $E = 0.01$

integrated with the hardware at a such deep level that their potential speedup is pure without any overheads, compared to other parallelism options (*i.e. threads*). Nonetheless, one should not forget their purpose, **doing a relatively simple, iterative task**. If the body of the loop is complicated, the computations are conditional and the memory access pattern is in a way that data cannot be readily loaded into registers, we cannot expect the ideal speedup with the order of 2. As an example, the speedup that we gained compared to the sequential version is at most has .3 order of magnitude. This can be explained by arguing that aside from the complexity of the cell update loop, data alignment in this assignment is not ideal for SSE intrinsics.

References

[1] Optimization Options - GCC https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[2] Intel Intrinsics Guide https://software.intel.com/sites/landingpage/IntrinsicsGuide/