Application acceleration with SIMD Extensions: "Hotspot" analysis

José Sarmento
MEEC
Instituto Superior Técnico
Lisbon, Portugal
jose.sarmento@tecnico.ulisboa.pt

José Serafim
MEEC
Instituto Superior Técnico
Lisbon, Portugal
jose.serafim@tecnico.ulisboa.pt

Pedro Lopes
MEEC
Instituto Superior Técnico
Lisbon, Portugal
pedrolopes1998@tecnico.ulisboa.pt

***Abstract*—The focal point of this project is the demonstration of improvement of a processor's performance using SIMD extensions so that we can accelerate kernels from a benchmark called *Hotspot*. The assignment was accomplished using the *gem5* simulator, a modular platform for computer-system architecture research that encompasses system-level architecture as well as processor microarchitecture. Initially, the original app's performance was studied and profiled. To accelerate this benchmark, *Arm SVE (Scalable Vector Extension)* was used. A comparison was established between the original benchmark and the developed solution, showing a considerable improvement in the processor's performance while assuring the expected output.**

***Index Terms*—Hotspot, processor, optimization, SVE, vectorization**

I. Introduction

This assignment targets the *Hotspot* benchmark which can be labeled as a transient temperature computing application. *Hotspot* is used to estimate processor temperature based on an architectural floorplan and simulated power measurements. The app solves differential equations iteratively for each block. Each output element represents the average temperature of the corresponding area of the chip. The benchmark converts the heat transfer differential equations to difference equations and solves the difference equations by iterating.

In terms of optimization/acceleration strategies, some aspects are identified as main time dispensing factors such as the high number of for loops iterations and their size. Alternatively, the fact that *for* loops are developed to execute first through rows is important and preserved because of the contiguous memory address space.

Taking this into account, the following optimization strategy was developed:

- *Profiling:* Study and analysis of the benchmark and its functions. Time measurement of functions and segments of code in order to identify which loops take more time to execute and that are worthy of an acceleration process.
- *Vectorization:* performing the vectorization method taught in the Advanced Computer Architectures class [1] [2].
- *Loop Unrolling:* performing the loop unrolling procedure taught in class [2].

The designed accelerated benchmark's code is located in the Cuda2 machine, group07/Hotspot/source folder.

II. Target application

The benchmark starts by allocating necessary space for the temperature and power arrays leading to the input of grid data from the respective data files (function *read_input*). This grid data is represented by a grid model where the matrix is divided in chunks of size 16×16. This app involves two major functions (where one of them calls the other):

- *compute_tran_temp()* - consists in a transient solver driver routine. Simply converts the heat transfer differential equations to difference equations and solves these by iterating.
- *single_iteration()* - consists in a single iteration of the transient solver in the grid model. Therefore, it advances a solution for the discretized difference equations by one time step.

The function *single_iteration()* is expressed in two big *for* loops. However, depending on the current 'position' in the matrix, only one loop will be executed since one of them focuses on outer chunks and the other one on internal chunks. Targeting an optimization of the application, a *profiling* was elaborated and the time spent in each function fraction and loops is represented in the table I. The program was executed with the Arm A7 CPU model and an input data size of 128, simulated under *gem5.opt*.

TABLE I
Benchmark's processing time before optimization

	w/o SVE [μs]	**% of function total time**
Transient temperature computing	788	100
All edge/corner chunks	175	22.21
All inner chunks	609	77.28
Average edge/corner chunks	6.25	0.79
Average inner chunks	16.92	2.15

Evaluating these loops, we can conclude that the *for* loop responsible for the inner chunks takes most of the time ($\approx$ 77.28% of total time).

Taking all this into account, the *for* loop that processes internal chunks (Code 1.) is to be accelerated.

Through Amdhal's Law (eq. 1), we can obtain a "ceiling" for the function speedup, Code 1. Inner chunk code.

```
for ( r = r_start; r < r_start + BLOCK_SIZE_R; ++r ) {
    for ( c = c_start; c < c_start + BLOCK_SIZE_C; ++c ) {
        result[r*col+c] = temp[r*col+c] +
        (    Cap_1 * ( power[r*col+c] +
             (temp[(r+1)*col+c] + temp[(r-1)*col+c]
                 - 2.f*temp[r*col+c])    * Ry_1 +
             (temp[r*col+c+1] + temp[r*col+c-1]
                 - 2.f*temp[r*col+c])    * Rx_1 +
             (amb_temp - temp[r*col+c]) * Rz_1        )
        );
    }
}
```

$$S = \frac{1}{1 - f + \frac{f}{s}} \tag{1}$$

where f is the fraction of the system we want to optimize ($f \approx 0.77$), and s the fraction speedup. To obtain the theoretical maximum function speedup S, we simply need to imply a maximum fraction speedup (i.e., $s \rightarrow \infty$), which results into $S \approx 4.40$. This is the result that this work will strive for this input size.

III. Parallelization approach

After choosing the function to parallelize, we started parallelizing it by using Arm SVE. The function we choose works with a matrix, and has two 'for' loops, one for the rows and one for the columns. We decided to parallelize the loop of the columns, because it was the loop with the most instructions, and it will also accelerate the rows loop.

We set our unrolling factor to 4, which means for each iteration of a row loop, we will compute 4 rows in parallel.

When using the SVE extension, we parallelized it with a vector length of eight, by loading 32 elements per register. The matrix was divided in chunks of 16x16 elements each; in our optimized loop we needed to process every chunk except the ones on the borders of the matrix. By loading 32 consecutive elements on a register, we were loading the first row of two chunks (16 elements per chunk). We kept processing 32 elements at a time from the same line, until we reached the before last chunk (the last chunk is a border chunk, which is processed in a different part of the code), and with that we could optimize the memory access, because all the elements in a line were stored in consecutive memory positions. It was possible to do this because for every possible size of matrix, the number of chunks we needed to process per row was divisible by two. After processing the first complete row (except the border chunks), we continue to the second row, and so on until we've processed all the non-border chunks.

The code can be observed in the linked repository.

IV. Results

The simulations were ran under *gem5*; all runs were executed with with the same number of iterations (1) and SVE vector length (1024-bit), and with two different CPU models and various input matrices sizes. The program was first compiled with the original C code only, then with the optimized fraction in SVE; everything was compiled with the `O2` optimization flag.

A slight variation was noticed in different runs with the exact same parameters; to mitigate this variance, a script was developed so that, for each set of parameters, it would run the program 100 times and output the median (except for sizes 512 and 1024, which 100 iterations would take too long; for comparison, one run under *gem5.opt* and input size 64 takes 44 seconds; an input size of 512 takes 33 minutes; 1024 takes 133 minutes).

The speedup for each component is obtained by calculating the division of the original over the optimized time.

The results can be seen on tables (II) and (III).

TABLE II
Vectorizable fraction time results

Execution specifications		**Fraction time**		
CPU model	*Input size*	*w/o SVE* [μs]	*w/ SVE* [μs]	**Fraction Speedup**
Arm A7	64	21	3	**7**
	128	610	90	**6.78**
	256	1872	401	**4.67**
	512	10234	1792	**5.71**
	1024	44261	7678	**5.76**
Arm A15	64	9	2	**4.5**
	128	172	31	**5.55**
	256	919	218	**4.22**
	512	4921	1030	**4.78**
	1024	23018	4931	**4.67**

TABLE III
Function time results

Execution specifications		**Function time**			
CPU model	*Input size*	*w/o SVE* [μs]	*w/ SVE* [μs]	**Function Speedup**	*% of* S_{max}
Arm A7	64	68	46	**1.48**	32.99
	128	785	266	**2.95**	65.76
	256	2239	743	**3.01**	49.34
	512	11078	2570	**4.31**	70.65
	1024	46228	9471	**4.88**	79.99
Arm A15	64	41	32	**1.28**	20.98
	128	275	133	**2.07**	33.93
	256	1124	417	**2.7**	44.26
	512	5399	1468	**3.68**	60.32
	1024	24229	5924	**4.09**	67.04

From the results, we can see from the get-go a significant time improvement with the optimized code.

The function speedup is higher with an increasing input size, as the number of inner chunks grows relative to the outer chunks. For input size 64, the ratio inner/outer chunks is 1:3; for 1024, it's approximately 48:3. As our function speedup increases, so does the theoretical maximum for each input size;

We can see that of the two CPU models, the Arm A15 is the fastest: this is because it is an out-of-order processor, which means that it will run instructions depending on the availability of input data and execution units, instead of the original code order. This is also why we see smaller speedups with this CPU, as it is the CPU itself that controls instructions flow and optimization.

V. Conclusion

The SVE has revolutionized the ARM architecture, by allowing us to have a more effective way to accelerate applications that were not available with NEON.

By completing this assignment we realized that we're able to obtain a full function speedup of 4.3 by parallelizing the code with the Arm SVE extension. That speedup allowed us to realize how the vectorization and loop unrolling can be very important at a larger scale, to accelerate machines, programs or processors.

It would be possible to adapt our program to use a bigger vector length, and by that, processing a higher number of elements at a time, what possibly would increase our speedup even more.

Further optimization could be established by integrating software prefetching in the Assembly code. Another possible route would be to increase the SVE vector length to 2048-bits and in the same loop process 4 chunk rows in a first part then 2 chunk rows in the other.

References

[1] F. Petrogalli, "A sneak peek into SVE and VLA programming"
[2] J. Domingos and P. Tomás, "SIMD processing with Arm NEON/SVE"