High Performance Compiler Optimizations for Mobile Processors

Keith H. Bova

Abstract—**This paper presents an experiment that can be used as a mobile processor benchmark. The program evaluates floating point operations per second (flops) by first computing a matrix-matrix product. Source code to reproduce the experiments is provided.**

Index Terms—**Vector Processor, in-order processor, CISC, RISC, MicroBlaze, AMD, Zilinx, Intel, ZCU-102, ARM A53**

I. Introduction

For this experiment, we wrote a program in the C language that implemented the loop interchange and blocking algorithms, as outlined by John L. Hennessy and David A. Patterson [1]. Most of the software was developed using an Intel Core i5-4260U; however, it was substantially revised using a 32 bit MicroBlaze, and finished testing with an Intel i9-9880H processor. We also discuss a benchmark on the ARM A53.

II. Background

Many scientific computing problems in engineering are memory bound– and are often so large that the problem does not fit inside the cache of a processor; in these cases, the programmer must write their software in such a way that it can leverage the underlying architecture as much as possible. Hennessy and Patterson describe the loop interchange and blocking methods as modifications that can be made to nested for loops– improving their data access patterns.

For this experiment, we primarily focused on optimizing the blocking algorithm. We saw that the loop interchange program was rather straightforward to implement–and it was relatively easy to demonstrate a speedup without much modification to our code. The blocking algorithm works in a similar way as the loop interchange–except its main focus is breaking structured data into chunks so that it can better fit in the cache.

III. Stack Size and Dynamic Memory Allocation

Standard C style arrays allocate memory in the stack by default, meaning as the size of the arrays increases–so does the size of the stack. The stack-size-limit can often vary from processor to processor, and if too much memory is allocated, you can blow the stack and cause the program to segmentation fault. We ran into this problem and came up with two solutions: the first solution was to increase the size of the stack using $ulimit - s$, and the second solution was to dynamically allocate memory using $malloc$ and $free$. We applied both of these methods in our program. Hennessy and Patterson also used two-dimensional arrays in their example; however, we chose to linearize each matrix as a one dimensional vector–further improving the access patterns.

IV. Performance Portability

The program started its life as a $C++$ application, but we quickly realized that it needed to be rewritten using C to improve its portability across embedded processors. We tested its portability by running the application on a 32 bit MicroBlaze, as described by Zach Pfeffer [2]. The steps to build the design were almost identical–however, we did not build PetaLinux and chose to flash the board directly. We noticed a few limitations about the MicroBlaze: the first is that it does not easily support the standard $printf$, and the second, it does not support the $< time.h >$ library. In order to benchmark applications, we found that we needed to replace every instance of $printf$ with the Xilinx $zilprintf$; however, even this change was not enough to get time readings–as further research revealed we needed to compile the $xtimel.h$ code to enable this functionality. In addition, we needed to use $sprintf$ before each call of $printf$–as the Xilinx implementation does not officially support printing floating point numbers directly. Unfortunately we did not have time to test the final application using dynamic memory allocation on the MicroBlaze, but with these changes to the software, we expect that it should run without any substantial modification–other than the changes already outlined above. Our only concern is with the level of difficulty associated with accessing a clock in the system.

V. The AMD ZCU-102

We chose to use the ZCU-102 FPGA to construct the MicroBlaze–as it was the hardware we had available at the time. The FPGA features the ARM A53 (a four-core, RISC in-order processor), and we decided to benchmark the CPU before constructing the MicroBlaze. To achieve this, built the disk image provided by Canonical [3], following a similar method as outlined by Terry O'Neal on the Xilinx wiki [4]. We ultimately settled on flashing the image to an SD card using Balena-Etcher. Plugging the board into an ethernet port on our router worked to login using ssh; however, it was missing a DNS and could not update until first calling

```
echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.
    conf > /dev/null$.
```

VI. High-Performance-Conjugate-Gradient

After logging into the ZCU-102 using ssh, we built the Mantevo HPCG benchmark [5] using spack, as outlined by AMD on their website[6]. We did not build it with OpenMP support, and instead built it using OpenMPI. We ran the benchmark, varying the number of cores from one to four, and generated a graph of the following:

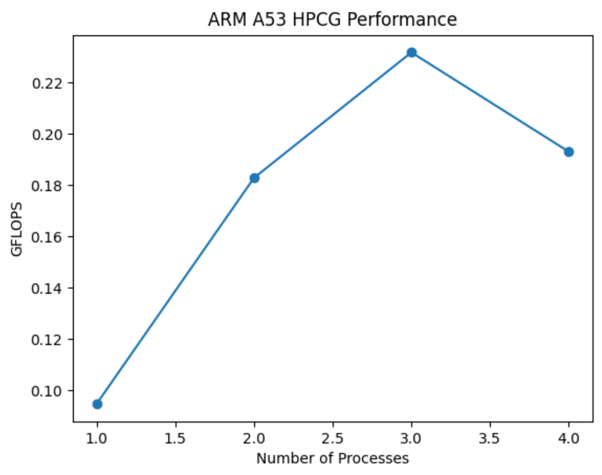

Fig. 1. HPCG Performance on the ARM A53 using the ZCU-102

After discussing with Dr. Andrew Targhetta, we come to the conclusion that the HPCG benchmark is likely communication bound–or the problem size was too small (which would imply communication bound in this case) [7].

VII. Matrix-Matrix Products Using Blocking

As mentioned earlier, in a linux shell, we can get the size of our stack by running the $ulimit - s$ command in the terminal. To calculate the upper limit of the size of our arrays that can fit in the stack, we must first take the value returned from $ulimit - s$, and divide it by the size of our data using the following equation:

$$N = \sqrt{\frac{U_{limit} * 1024}{3 * size(T)}} \tag{1}$$

If we are using $T = float = 4bytes$, and have an upper limit of 8192 kilo-bytes, then we can calculate that a size greater than 836 (without dynamic memory allocation) will cause us to blow the stack. Our dense matrix-matrix product contained two operations–a multiply and an add–and was executed inside three loops of length N. We can express this mathematically as:

$$flops = \frac{2 * N^3}{t} \tag{2}$$

VIII. Results

We found that we could compile our code without errors using both gcc and g++. For this experiment, we defined a size of 512, a block size of 32, and $T = float$. Using gcc and varying the compiler optimizations, we observed the following:

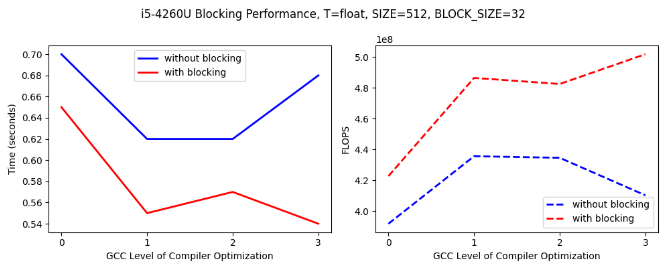

Fig. 2. Blocking before and after, varying the gcc compiler optimizations for floating point precision

We repeated the experiment–this time using double precision instead of float32. We observed the following:

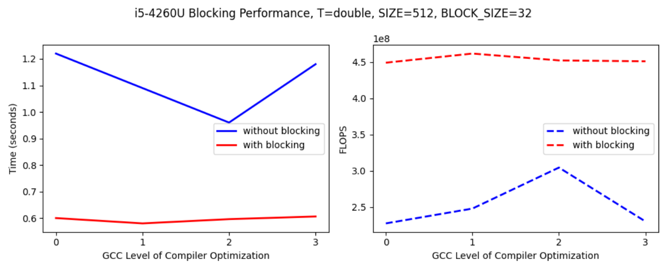

Fig. 3. Blocking before and after, varying the gcc compiler optimizations for double precision

We see from the above results that the blocking algorithm outperformed the baseline in every metric, and from our experiments, the optimal gcc compiler optimizations for single precision is 3, and 2 for doubles. Finally, we repeat the above experiment, using $T = double$ and second level compiler optimization, sweeping the matrix size from 256, to 512, 1024, and finally to 2048. We observe the following:

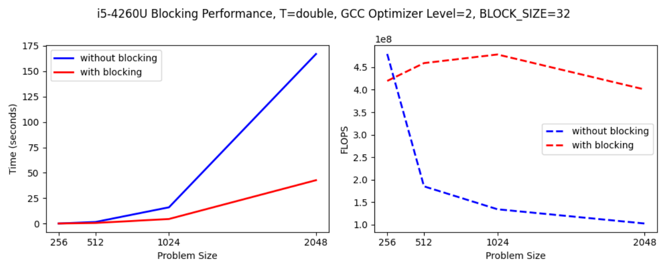

Fig. 4. Blocking before and after, varying the problem size for double precision

We see from the above plot that blocking is only effective for problems that don't fit in the cache. If we represent our cache size as C, the byte-size as B and the problem size as N, we can express the relationship with the following equation:

$$3BN^2 = C \tag{3}$$

Using algebra we can rewrite this relationship as:

$$N = \sqrt{\frac{C}{3B}} \tag{4}$$

Using the datasheet for the Intel i5-4260U, we see that it has a cache size of 3MB [8]. Plugging this value back into the equation, and using a size of 8 bytes for a double, we can rewrite the equation as:

$$N = \sqrt{\frac{3 * 1024^2}{3 * 8}} = 256 * \sqrt{2} \approx 362 \tag{5}$$

According to the above equation, we would expect the minimum problem size where blocking would be effective to occur where $N > 362$, which is consistent with our results.

IX. Conclusion

We took a three-sample average for each experiment–which was enough to get consistent results; however, in practice it would be best to use more samples. We see cache optimizations substantially affect the runtime of an application, and blocking is an effective strategy if your problem does not fit inside the cache; if it does fit in the cache, the extra loop-overhead actually slows down the application.

In addition, testing on the MicroBlaze and the A53 tremendously helped designing a lightweight application that worked across different compilers. When building a MicroBlaze, it is important to note that it can break an existing operating system on the A53; care must be taken so that both instances of the processors can run simultaneously on the FPGA.

Finally, we found that it helps to write an application under the assumption that it will not fit in the cache–then optimize it to fit in the cache using the methods described above. While the processor is fetching data from memory, the program is still able to execute without the pipeline grinding to a complete halt. We intend to continue research with this program on the A53 and the MicroBlaze.

References

[1] *Computer Architecture A Quantitative Approach.* John L. Hennessy & David A. Patterson

[2] *Create a MicroBlaze.* Zach Pfeffer [Online]. Available: https://www.css-techhelp.com/post/create-a-microblaze-test-the-uart-in-sdk-and-boot-linux-using-2019-1-vivado-and-petalinux-tools

[3] *Install Ubuntu on AMD.* Canonical [Online]. Available: https://ubuntu.com/download/amd

[4] *Building and Runnung the Ubuntu Desktop From Sources.* Terry O'Neal [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841937/Zynq+UltraScale+MPSoC+Ubuntu+part+2+-+Building+and+Running+the+Ubuntu+Desktop+From+Sources

[5] *Mantevo HPCCG.* Michael A. Heroux [Online]. Available: https://github.com/Mantevo/HPCCG

[6] *Build HPCG Using Spack.* AMD [Online]. https://www.amd.com/en/developer/zen-software-studio/applications/spack/hpcg-benchmark.html

[7] *Conversations with Dr. Andrew Targhetta.*

[8] *Intel i5-4260U Datasheet.* Intel [Online]. https://www.intel.com/content/www/us/en/products/sku/75030/intel-core-i54260u-processor-3m-cache-up-to-2-70-ghz/specifications.html