Zynq Parrot Core Profiling for Matrix Multiplication

Introduction

With the increasing demand for computation on devices on the network edge (such as on phones and IoT devices) as well as in off the grid settings, more software and hardware solutions are being co-designed to make the computations possible [1]. This paradigm is also useful in developing countries where network connectivity can be strained and device resources may be limited [2]. Some devices in the low resource settings even lack compilers to make smart decisions on how to restructure code to run optimally on the given hardware. It becomes important therefore to incorporate techniques that help make efficient use of scarce resources and potentially without loss of performance.

In this project, we set up a measurement system in the zynq parrot implementation to capture pipeline stalls due to cache misses and notice performance differences when we implement two techniques for data reuse proposed for matrix multiplication.

Background

Black parrot is an open source processor originally funded by DARPA but currently pulls from the computer hardware community to build a modular and easy to adopt RISC-V processor with offerings for ASIC and FPGA [3]. Zynq board created by Xilinx is a programmable system on chip that has proved to be very useful among researchers and hackers alike to tinker with computer hardware development and experiments [4]. For this project, we use the Pynq-Z2 offering of the Zynq board which comes with an ARM core processor and traditional FPGA fabric logic.

Matrix multiplications underlie many of the algorithms that eventually will be run on edge compute devices. In this project, we combine the power of black parrot and the zynq board to open up measurements on the hardware level to estimate performance benefits of matrix multiplication optimizations.

Zynq Parrot Setup

Zynq Parrot is an implementation of Black Parrot to run on the Zynq board with the ARM core as a Processor Subsystem, hosting a control program, which is used to issue read/write instructions to the Black Parrot processor in the FPGA as a Programmable Logic element. We use the matmult-float beebs benchmark to test how black parrot performs with different matmult optimizations.

We made the following changes to zynq parrot.
i. Convert bind construct for collecting processor stall information from black parrot to a synthesizable construct.
ii. Instantiate the profiler in a copy of black parrot unicore minimal found in the top zynq cosim black-parrot-example directory and update the flist.vcs file to point to the copy.

iii. Update the top_zynq file to pull the counters from (ii) to the CSR registers

Matrix Multiply Optimizations & Results

In order to improve throughput and reduce resource utilization requirements for matrix multiplication, there have been a number of proposed approaches. Of these approaches, I select two that make use of the knowledge of register load patterns to improve data reuse as discussed in [5]. The standard for-loop for matrix multiplication looks like the code snippet in Figure 1.

```
void Multiply(matrix A, matrix B, matrix Res)
{
    register int Outer, Inner, Index;

    for (Outer = 0; Outer < UPPERLIMIT; Outer++)
        for (Inner = 0; Inner < UPPERLIMIT; Inner++)
        {
            Res [Outer][Inner] = ZERO;
            for (Index = 0; Index < UPPERLIMIT; Index++)
                Res[Outer][Inner] += A[Outer][Index] * B[Index][Inner];
        }
}
```

Figure 1. Typical 3 for-loop NxN matrix multiplication

i. *Spatial Locality* is based on the assumption that during the matrix multiply operations, nearby data can be reused. In particular, we reuse the value from one of the matrices throughout an inner for loop as shown below. This saves multiple data moves where the value in register r would have been stored and loaded within the innermost for-loop.

```
void Multiply(matrix A, matrix B, matrix Res)
{
    register int Outer, Inner, Index;

  for (Index = 0; Index < UPPERLIMIT; Index++) {
    for (Outer = 0; Outer < UPPERLIMIT; Outer++) {
        register float r = A[Outer][Index];
        for (Inner = 0; Inner < UPPERLIMIT; Inner++){
            Res[Outer][Inner] += r * B[Index][Inner];
        }
    }
  }
}
```

Figure 2. A rewrite of the matrix multiply for loop to take advantage of spatial locality.

ii. *Temporal Locality* assumes that data that was recently used will be used again. In the block implementation, matrices are broken into small blocks of size Block by Block such that data for the block fits into the cache and used in computing a partial sum as in Figure 3.

```
// Block
void Multiply(matrix A, matrix B, matrix Res)
{
    register int Outer, Inner, Index, i1, j1, k1;
    int Block = 5;

  for (Outer = 0; Outer < UPPERLIMIT; Outer+=Block) {
    for (Inner = 0; Inner < UPPERLIMIT; Inner+=Block) {
        for (Index = 0; Index < UPPERLIMIT; Index+=Block) {
          // BxB mini matrix multiplications
            for (i1 = Outer; i1 < Outer+Block; i1++)
              for (j1 = Inner; j1 < Inner+Block; j1++)
                for (k1 = Index; k1 < Index+Block; k1++)
                  Res[i1*UPPERLIMIT + j1] += A[i1*UPPERLIMIT  + k1] * B[k1*UPPERLIMIT + j1];
        }
    }
  }
}
```

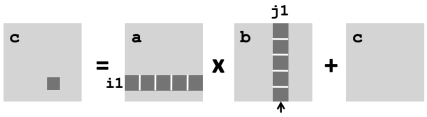

Figure 3. Block matrix multiplications to take advantage of temporal locality.

I tested these both in verilator simulation and on the FPGA board. In verilator, taking advantage of spatial locality led to fewer icache misses, as a result of data reuse from matrix B as shown in figure 1. There was however a worsening effect due to more cache replacements for matrix A.

Table 1. Verilator icache and dcache miss/rollback results for running float matrix multiplications at size 10,15 and 20. Default refers to the 3 for loops for matrix multiplication, temporal uses a block size of 1.

Default (matmult-float)

Beeb	# Icache miss	#Icache rollback	#dcache miss	#dcache rollback
10x10	548	125	1418	284
15x15	551	129	2081	440
20x20	552	139	3017	656

Spatial (matmult-float)

Beeb	# Icache miss	#Icache rollback	#dcache miss	#dcache rollback
10x10	503	128	1440	284
15x15	511	132	2091	436
20x20	516	131	3035	652

Temporal (matmult-float)

Beeb	# Icache miss	#Icache rollback	#dcache miss	#dcache rollback
10x10	538	133	1433	284
15x15	551	137	2072	432
20x20	550	137	3005	652

At the default 512 cache size, matrices of size 1000x1000 started to take much longer to run the for loops but were still not seeing cache misses to test the benefits of running the spatial and temporal locality algorithms. I therefore ended up changing the cache size to 8 and 1 set before being able to test the changes in reasonable time. The change is also in line with testing performance on tiny devices with limited memory.

```
localparam bp_proc_param_s bp_unicore_l1_tiny_override_p =
  '{icache_sets         : 8 // 64
    ,icache_assoc       : 1
    ,icache_block_width : 8 // 512
    ,icache_fill_width  : 8 // 512
    ,dcache_sets        : 1 // 64
    ,dcache_assoc       : 1
    ,dcache_block_width : 8 // 512
    ,dcache_fill_width  : 8 // 512
    ,default : "inv"
    };
`bp_aviary_derive_cfg(bp_unicore_l1_tiny_cfg_p
                      ,bp_unicore_l1_tiny_override_p
                      ,bp_unicore_cfg_p
                      );
```

Figure 4. Changing black parrot cache size from 512 (in blue comment) to 8 and cache sets from 64 to 1.

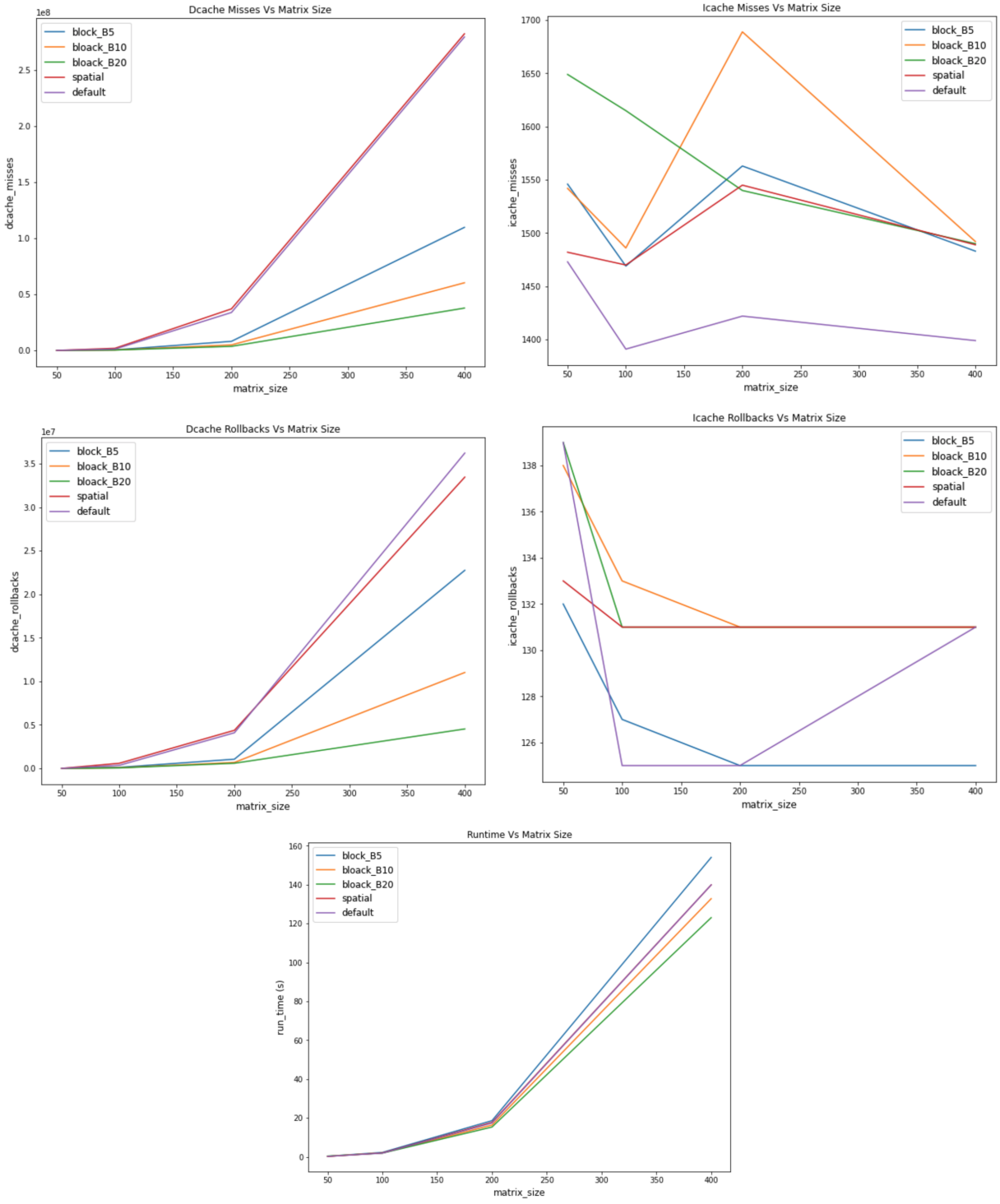

Figure 5. Graphs of Data cache and Instruction cache misses and rollbacks as well as time taken by algorithm type and matrix size

As seen from figure 5, using blocking gave the most benefits on the FPGA regarding Dcache misses and block size 20 of the options we tried performed the best. Effects on Icache are not very well defined. For spatial locality, we didn't see much benefits but that could be because our

current cache size ends up causing more misses in the matrix we don't fix a spot in (i.e. the matrix B in Figure 2) and so a bigger cache is needed to see the benefits.

Conclusion

With increasing need for compute intensive algorithms on tiny devices, hardware/software codesign becomes more important than ever before. In this project, we took a look at one such codesign and the potential gains in simulation and on an FPGA board. For future projects, we hope to dive deeper into that hardware side of things to learn of further optimization techniques that can assist in writing hardware aware algorithms.

References:

[1] Shi, Weisong, et al. "Edge computing: Vision and challenges." *IEEE internet of things journal* 3.5 (2016): 637-646.

[2] MTSHALI, Mxolisi, et al. "Edge Computing for Emerging Markets Addressing African Needs." *2019 IST-Africa Week Conference (IST-Africa)*. IEEE, 2019.

[3] Petrisko, Daniel, et al. "Blackparrot: An agile open-source risc-v multicore for accelerator socs." *IEEE Micro* 40.4 (2020): 93-102.

[4] Crockett, Louise Helen, et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[5] http://www.cs.cmu.edu/afs/cs/academic/class/15213-s19/www/lectures/12-cache-memories.pdf