CPEG 421/621: Compiler Design: Optimization of Dense Matrix Multiplication
Handout: Project Part 1
Issue Date: 3 November, 2010
Due Date: 23 November, 2010

1 Problem

Matrix Multiplication algorithms have been studied extensively. The General Matrix Multiplication (GEMM) is the core of every linear algebra library (e.g. BLAS). Studies of Matrix Multiplication (MM) focus mainly on: (1) Algorithms that decrease the naive complexity of $O(m^3)$ (e.g. Strassen's algorithm). (2) Implementations that take advantage of advanced features of computer architectures to achieve higher performance. This project is oriented towards the second area.

Our objective is to maximize the performance of the double precision dense matrix multiplication $A \times B = C$, with each matrix of size $m \times m$ (square matrices) using algorithms of running time $O(m^3)$ using the IBM Cyclops 64 (C64) many-core Architecture [1]. Operands of MM are in on-chip memory (SRAM). The serial pseudo-code is shown on figure 1.

```
0:  for i = 1 to m
1:     for j = 1 to m
2:        for k = 1 to m
3:           C[i*m+j] += A[i*m+k]*B[k*m+j]
```

Figure 1: Serial MM pseudo-code

Throughout the design process proposed, specific features of C64 the will be used to illustrate the advantages of a set of Compiler Optimizations.

2 Documents Required

This handout is also packed with the following documents:

1. Cyclops 64 Basic Instruction Set (sent by email).
2. Different Versions of code for Matrix multiplication on Cyclops 64. (sent by email).
3. Links to references to be downloaded from our repository. (at the end of the handout).

3 Procedure

3.1 Compiler, Simulator and Chip

- You need access to the CAPSL server atlantic.capsl.udel.edu. Contact the TA if you need assistance.
- Add /opt/eti-c64/local/bin to your PATH variable in order to compile and simulate programs for C64.
- The C64 compiler follows the same syntax of gcc. For example to compile a program hello.c just use:

  cyclops64-linux-elf-gcc -O3 hello.c -o hello.out
- The C64 simulator has several flags. For simulating the hello.out program you can use:

  cyclops64-linux-elf-sim –spsd –bw –icache -p80 ./hello.out

  Where –spsd means single program single data, –bw includes the crossbar contention of the chip, –icache models the instruction cache and -p80 uses 80 Processors. Notice that a C64 chip has 80 Processors and each processor has two thread units.

  Note 1: The flags –bw –icache increase the simulation time, you can avoid them for debugging purposes but NOT for reporting results. Timing results of the simulator with the flags –bw and –icache are near to timing on the real chip but highly optimized code can produce very optimistic timing results on the simulator.

  Note 2: If you need to make simulations for a different number of thread units, DO NOT change the flag -p80 because it will change the memory and crossbar mapping and produce different timing. You can include the number of threads/processors as an input of your program.

  Note 3: For additional flags you can use –help
- To run programs on the real chip you will have the help of people of ET International. Ask the instructor about the procedure.

3.2 Simple Exercise

A partially optimized code for MM on C64 will be provided, you will simulate this code for being familiar with the C64 tool chain and the MM.

1. You will receive different codes for MM, some of them have compiler optimizations. Put the folders with different codes on the capsl server atlantic.capsl.udel.edu. We will start with the folder mmPartition3. It contains 3 files mm.c, Makefile and Readme.txt
2. Read the Readme.txt file, it includes a short description of the program, how to compile it using the Makefile and how to run it.

3. Open the Makefile and verify that you will use the CPPFLAGS -DMEASURE, -DDEBUG, -DCOMPARE, -DNOSIMPLIFY.

4. Compile the program typing on your command line

   make

5. Run the simulation typing

   make sim

   The program will print debug information like the partitions, and the ID of threads finished. At the end, the resultant matrix C is printed with useful information like:

   Total: seq = 838.610355, par = 838.610355 Elapsed cycles: 311576. # Mega Floating Point Operations: 6.525. MFLOPS: 10.471.

   Notice that the correctness of the algorithm is checked by checksum of sequential and parallel computations, they have to be equal.

6. Edit the makefile for use only the -DMEASURE flag. Clean the folder files using the command

   make clean

   Simulate again using

   make sim

   Now you are measuring the real performance of the MM.

7. Now you will simulate a MM of 300x300 using 144 thread units. You can edit the makefile or type

   cyclops64-linux-elf-sim –spsd –bw –icache -p80 mm -p144 -n300

8. What is the maximum speed up and performance of this naive version?

9. Make a graph of performance vs number of threads (e.g. 1, 4, 9, ... , 144) for different matrix sizes (e.g. 50, 100, 300) and comment your results. Is the performance dependent of matrix size? if so, why?

10. Explain how can you make an optimal partition for using all the thread units available on C64.

    Hint: C64 has 160 threads units but 1 is used for the runtime and 3 are used for communication with other chips. The maximum number of available threads is 156.

3.3 Optimized Dense Matrix Multiplication

The optimization of MM involves several stages. The instructor will provide several codes that incrementally apply different Compiler Optimizations that you have to study, also you have to implement some of them. The characteristics of the codes are:

- It works with square matrices of arbitrary size. If on-chip memory is full, the matrices are allocated in off-chip memory and the performance will decrease.
- It is optimized for an square number of threads (1, 4, 9, ... , 144).
- It uses an optimum statical partition for load balancing.
- It uses assembly code for increasing performance.
- The performance is measured using:
  $Performance[MFLOPS] = \frac{500[MHz]*(2m^3-m^2)[FloatingPointOperations]}{\Delta t[cycles]}$

Your goal is to increase as much as possible the performance of the MM with the characteristics given in section 1. You will study the incremental performance of the following compiler optimizations using **the simulator and the real C64 chip**. Follow these steps:

1. **Assembly Implementation:** Look the folder mmNoTiling. This MM version has a naive assembly implementation of the function ProcessBlock. Read the code and identify the three nested loops.
   (a) What is the function of the code at the beginning and the end of the function saving and loading the registers? Why is it necessary?
   (b) Make a graph of performance vs number of threads (e.g. 1, 4, 9, ... , 144) for different matrix sizes (e.g. 50, 100, 300) and comment your results. Compare with the C version.

2. **Register Tiling and Register Allocation:** Look the folder mmTiling3. It includes a MM that uses an optimum register tiling following [2]. This Tiling minimize the number of memory operations maximizing the reuse of data. It will decrease the access to on-chip memory and the stalls produced by memory latencies.
   (a) Identify the points in the assembly that calculate the tiles. Write a pseudo-code that describes the control flow of the function ProcessBlock.
   (b) For C64, the optimum tile size on C is 6×6 without any spilling or use of scratchpad memory. Discuss the pros and cons of using spilling for increasing the performance of this version of MM.
   (c) Suppose the number of register available for C64 is decreased so the maximum tile size is 4×4. Modify the code for using this a maximum tile size.
   (d) Make a graph of performance vs number of threads (e.g. 1, 4, 9, ... , 144) for different matrix sizes (e.g. 50, 100, 300) using the two tilings and comment your results. Compare with previous versions.

3. **Instruction Selection:** One of the most important cases is the use of special assembly functions for Load and Store. C64 provides the instructions multiple load (*ldm RT, RA, RB*) and multiple store (*stm RT, RA, RB*) that combine several memory

operations into only one instruction. For the *ldm* instruction, starting from an address in memory contained in RA, consecutive 64-bit values in memory are loaded into consecutive registers, starting from RT through and including RB. Similarly, *stm* instruction stores 64-bit values in memory consecutively from RT through and including RB starting in the memory address contained in RA. The advantage in the use of these instructions is that the normal load instruction issues one data transfer request per element while the special one issues one request each 64-byte boundary. A tiling that uses consecutive data in memory can improve the performance with these kind of instructions [2].

Other examples are use of shifting instead of multiplications and divisions by powers of 2 and the replacement of integer division and remainder operations by other integer operations if it is possible.

(a) Modify the mmTiling3 code for a better instruction selection using the 6×6 tiling.

(b) Make a graph of performance vs number of threads (e.g. 1, 4, 9, ... , 144) for different matrix sizes (e.g. 50, 100, 300) using that version and comment your results. Compare with previous versions.

4. **Instruction Scheduling:** The correct interleaving of independent instructions to alleviate stalls. Data dependencies can stall the execution of the current instruction waiting for the result of one issued previously. We want to hide or amortize the cost of critical instructions that increase the total computation time executing other instructions that do not share variables or resources. The most common example involves interleaving memory instructions with data instructions but there are other cases: multiple integer operations can be executed while one floating point operation like multiplication is computed [2].

   (a) Modify the previous version of the code for a better instruction scheduling.

   (b) Make a graph of performance vs number of threads (e.g. 1, 4, 9, ... , 144) for different matrix sizes (e.g. 50, 100, 300) using that version and comment your results. Compare with previous versions.

5. **Data Prefetching and Loop Unrolling:** The Instruction Scheduling may hide only partially the latencies of fetching the operands from on-chip memory to registers. To eliminate stalls due to latency, prefetching operands into registers using loop unrolling in the calculation of the tile can improve significantly the performance. It also implies new register allocation given the increase of register pressure. Be aware of the trade offs between the data prefetching and the register tiling, you may need to change the tile in order to take advantage of these techniques [3].

   (a) Modify the previous version of the code for making Data Prefetching Unrolling once the loop that calculates the tile.

(b) Make a graph of performance vs number of threads (e.g. 1, 4, 9, ... , 144) for different matrix sizes (e.g. 50, 100, 300) using that version and comment your results. Compare with previous versions.

Useful information about MM on C64 can be found in [4, 2, 3]

3.4 What to hand in

- A report with all the experiences, answer of questions, design choices and analysis made for each incremental optimization with graphs of Performance vs Number of Thread Units for different matrix sizes for each case.
- The programs designed with their corresponding makefile.

Team Work and Project Management

You need to establish a team methodology as in a real world design project. It is mandatory that you elect a project team leader whose function, among other things, is to call the design meetings. In your first project review meeting, you should partition the work so each team member should have a clear responsibility and a fair portion of the workload. You should also have a plan for weekly design review meetings (expected 4-5 times in total).

Ethics for Team Work

Although discussion of the project in general terms is to be expected, members of different groups should not exchange source code or project implementation details. Also, it is expected that the contributions of each member of group will be clearly detailed, so care should be taken to make sure that each person contributes a fair amount. Therefore, your team strategy should be geared to prevent bottlenecks where group progress is entirely dependent on one person. A well-considered strategy and a clearly defined interface may mitigate such difficulties.

References

[1] M. Denneau and H. S. Warren Jr., "64-bit Cyclops: Principles of Operation," IBM Watson Research Center, Yorktown Heights, NY, Tech. Rep., April 2005.

[2] E. Garcia, I. E. Venetis, R. Khan, and G. Gao, "Optimized Dense Matrix Multiplication on a Many-Core Architecture," in *Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010), Part II*, ser. Lecture Notes in Computer Science, vol. 6272. Ischia, Italy: Springer, 2010, pp. 316–327. [Online]. Available: http://www.capsl.udel.edu/pub/doc/papers/ElkinGarcia-EuroPar2010.pdf

[3] E. Garcia, R. Khan, K. Livingston, I. E. Venetis, and G. Gao, "Dynamic percolation - mapping dense matrix multiplication on a many-core architecture," *CAPSL Technical Memo 98*, June 2010. [Online]. Available: ftp://ftp.capsl.udel.edu/pub/doc/memos/memo098.pdf

[4] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao, "Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences," in *12th International European Conference on Parallel Processing (Euro-Par 2006)*, Dresden, Germany, Aug. 2006, pp. 134–144. [Online]. Available: http://www.capsl.udel.edu/pub/doc/papers/EuroPar-2006.pdf