CSE 30321 – Computer Architecture I – Fall 2011

Lab 05 – Applying knowledge of underlying hardware (i) write more efficient code and (ii) to efficiently program multicore chips

Assigned: November 15, 2011 **– Due:** December 1, 2011 **– Points:** 100

1. Introduction:

At a high-level, the purpose of this lab is to further illustrate how (i) knowledge of underlying processor hardware and (ii) system-level benchmarking can lead to more efficient code – and hence faster run times for programs that are important to the end user.

- More specifically, Part 1 of this lab will focus on caching hardware. You will study how changing and "tuning" code that you write – to effectively use available cache hardware – can have a non-negligible impact on ultimate program run time (i.e. CPU time).

- Part 2 of this lab will introduce you to how multi-core computer architectures can impact system-level performance, as well as conventional "coding." More specifically, we will work with a recursive mergesort algorithm – where a dataset will be sorted on both single and N-core machines. You will see that by understanding what the underlying (multi-core) architecture looks like, it is possible to write more efficient code. You will also see that performance evaluation techniques learned in previous lectures / used in previous assignments are just as relevant to multi-core processing too.

2. Alternative Matrix Multiplies

An understanding of an underlying processor-memory architecture can actually make you a better programmer. In this problem, we'll use matrix multiply code as a case study. (Matrix multiplies are common in scientific computing, graphics transformations, etc.)

2.1 Different Matrix Multiplies:

For this problem, you will study code that will multiply the same 2 matrices together – but in two different ways. To get started, copy the "matrix_multiply" directory to your AFS space (see path below).

```
/afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/05/matrix_multiply
```

In this directory, there are 2 sub-directories: "no_blocking" and "blocking". Each sub-directory contains the following:

1. A file with c-code to perform a matrix multiply
2. An ARM processor configuration file
3. A simple script to run your program on the ARM configuration with SimpleScalar

Note, that you *do not need to run any SimpleScalar simulations for this lab*. However, I am including the configuration file and script that I used if you wish to (i) obtain any additional data points to help you answer the questions below (probably not necessary) or (ii) complete the extra credit question (this is required for the extra credit).

2.2 Questions:

Question 2.A

Trace through the code for the alternative matrix multiply (see blocking.c). What is this code doing differently than the more traditional "non-blocking" code (no_blocking.c)?

Question 2.B

In the tables below, I have summarized relevant SimpleScalar simulation data for 50x50, 150x150, and 1000x1000 matrix multiplies. Different cache sizes and different blocking factors (the variable B in the blocking.c file) are assumed. (Note that for typical scientific computing type problems – e.g. weather forecasting – (i) 1000x1000 matrices are *small* and (ii) the 1000x1000 architectural level simulations required approximately 1.5-2 days of compute time!)

50x50 matrices

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	8	1	2048	2.1706	5476205	1.00E-09	0.01189	6.38%
Blocking, B=5	256	8	1	2048	1.472	7739338	1.00E-09	0.01139	1.95%
Blocking, B=10	256	8	1	2048	1.4166	6920798	1.00E-09	0.00980	1.68%
Blocking, B=20	256	8	1	2048	1.5756	6606164	1.00E-09	0.01041	2.58%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	16	1	4096	1.952	5476205	1.00E-09	0.010690	4.83%
Blocking, B=5	256	16	1	4096	1.3339	7739338	1.00E-09	0.010324	1.13%
Blocking, B=10	256	16	1	4096	1.2444	6920798	1.00E-09	0.008612	0.59%
Blocking, B=20	256	16	1	4096	1.2912	6606164	1.00E-09	0.008530	0.93%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	32	1	8192	1.7581	5476205	1.00E-09	0.009628	3.49%
Blocking, B=5	256	32	1	8192	1.2767	7739338	1.00E-09	0.009881	0.74%
Blocking, B=10	256	32	1	8192	1.2444	6920798	1.00E-09	0.008612	0.59%
Blocking, B=20	256	32	1	8192	1.2451	6606164	1.00E-09	0.008225	0.60%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	64	1	16384	1.2195	5476205	1.00E-09	0.006678	0.51%
Blocking, B=5	256	64	1	16384	1.2258	7739338	1.00E-09	0.009487	0.40%
Blocking, B=10	256	64	1	16384	1.2172	6920798	1.00E-09	0.008424	0.37%
Blocking, B=20	256	64	1	16384	1.2198	6606164	1.00E-09	0.008058	0.39%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	128	1	32768	1.1291	5476205	1.00E-09	0.006183	0.03%
Blocking, B=5	256	128	1	32768	1.1367	7739338	1.00E-09	0.008797	0.02%
Blocking, B=10	256	128	1	32768	1.1351	6920798	1.00E-09	0.007856	0.02%
Blocking, B=20	256	128	1	32768	1.1347	6606164	1.00E-09	0.007496	0.02%

150x150 matrices

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	8	1	2048	2.4874	150417309	1.00E-09	0.374148	8.65%
Blocking, B=5	256	8	1	2048	1.4875	214237398	1.00E-09	0.31868	2.12%
Blocking, B=10	256	8	1	2048	1.3935	191480928	1.00E-09	0.26683	1.60%
Blocking, B=20	256	8	1	2048	1.4812	181279044	1.00E-09	0.26851	2.14%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	16	1	4096	1.9273	150417309	1.00E-09	0.289899	4.96%
Blocking, B=5	256	16	1	4096	1.3606	214237398	1.00E-09	0.291491	1.34%
Blocking, B=10	256	16	1	4096	1.2853	191480928	1.00E-09	0.246110	0.94%
Blocking, B=20	256	16	1	4096	1.2813	181279044	1.00E-09	0.232273	0.93%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	32	1	8192	2.0758	150417309	1.00E-09	0.312236	5.50%
Blocking, B=5	256	32	1	8192	1.3202	214237398	1.00E-09	0.282836	1.03%
Blocking, B=10	256	32	1	8192	1.2482	191480928	1.00E-09	0.239006	0.67%
Blocking, B=20	256	32	1	8192	1.2348	181279044	1.00E-09	0.223843	0.60%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	64	1	16384	1.7749	150417309	1.00E-09	0.266976	3.39%
Blocking, B=5	256	64	1	16384	1.2765	214237398	1.00E-09	0.273474	0.71%
Blocking, B=10	256	64	1	16384	1.2204	191480928	1.00E-09	0.233683	0.46%
Blocking, B=20	256	64	1	16384	1.2084	181279044	1.00E-09	0.219058	0.41%

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	256	128	1	32768	1.3281	150417309	1.00E-09	0.199769	0.90%
Blocking, B=5	256	128	1	32768	1.2791	214237398	1.00E-09	0.274031	0.59%
Blocking, B=10	256	128	1	32768	1.2234	191480928	1.00E-09	0.234258	0.38%
Blocking, B = 20	256	128	1	32768	1.2348	181279044	1.00E-09	0.223843	0.60%

1000x1000 matrices

Algorithm	Sets in $	Bytes/set	Blocks/set	$ size (bytes)	CPI	# instructions	Clock period	CPU time	$ miss rate
No Blocking	16	32	32	16384	2.7982	58126096440	1.00E-09	162.648443	10.24%
Blocking, B = 20	16	32	32	16384	1.4875	68253137391	1.00E-09	101.52654	0.12%

Given the provided data:

- Explain how blocking can positively and negatively impact performance.
- Explain when you should consider using the blocking technique.
  - Selectively quantify performance gains/losses to help support your answer.

Question 2.C (**Extra credit – not required)**
Consider other configurations (cache size and associativity, blocking factors, matrix sizes, etc.) other than those provided in the tables above might impact CPU time. Your response to this question must be organized as follows:

1. Explain *why* you chose a particular configuration.
2. Explain how you expect CPU time to be affected – and why you expect it to change in a certain way. (You should aim for additional performance *improvement.*)
3. Use SimpleScalar simulations to quantify CPU time.
4. Revisit your predictions and discuss the outcomes. Did they hold? Or did you see different results? Why do you think this is the case?

Remember, to compile the matrix multiply (or any) C-code so that we can study it within the SimpleScalar environment, you can simply type:

```
>        arm-gcc blocking.c –o blocking -static
```

To run this program in the SimpleScalar environment (i.e. to see how efficiently the StrongARM configuration can perform this matrix multiply), you can simply use the script provided OR type:

```
>        sim-outorder –config <configuration file name>.cfg <cross-compiled executable name>
```

Finally, remember that 1000x1000 matrix multiply simulations can take 1-2 days to run. 150x150 or 50x50 matrix multiplies generally take just a few minutes.

3. A Real Life Example:

The focus of my research is on computational devices beyond the transistor – which may ultimately be limited by physics, cost, and manufacturing-related issues. My research efforts have primarily targeted computational systems where magnetic elements with nanometer feature sizes are used to both process and store binary information.

Why is this good? In conventional electronics, most computation is charge-based. A power supply maintains state, and thousands of electrons (*each* dissipating ~1.66×10^{-19} J of energy) are needed to perform a single function (when you consider a chip with a billions transistors on it, this adds up quickly!). Alternatively, nanomagnet logic (NML) devices can process information in a cellular-automata like architecture, could dissipate less than 1.66×10^{-19} J per switching event for a *gate* operation, will retain state without power, and are intrinsically radiation hard.

Thus, looking up to applications, NML has the potential to mitigate increasing chip-level power densities that are currently exacerbated by device scaling, could help to improve battery life in mobile information processing systems, and may operate in environments where transistor-based logic and memory cannot.

To design and study magnetic logic systems, both my graduate students and I do lots and lots of physical-level simulations. In particular, we use a simulation suite called "OOMMF" – or the "Objected Oriented MicroMagnetic Framework". OOMMF simulations give us an idea of how an ensemble of magnets (which forms a magnetic logic gate or circuit) might perform when "clocked" after being subjected to new inputs. These simulations are critical for experimental design (there is excellent correlation between simulation results and experimental results), as well as for projecting how well NML might ultimately perform and scale.

The current version of OOMMF used by our group is multi-threaded, and the user can specify the number of threads that a given simulation will use – i.e. 8 cores could be used to simulate 1 magnetic ensemble, or 8 cores could be used to simulate 8 different magnet ensembles simultaneously. As you might expect, presumably the single threaded simulations would have longer execution times.

While the correlation between simulation and experimental results is strong, one drawback to this simulation-based approach to design is that the simulations themselves can be quite time consuming (i.e. 2-3 days are sometimes needed to see how just a 10-15 magnet ensemble will respond to just 1 input). Moreover, because of the rather complex magnetization dynamics, any changes to the material system being studied, magnet geometries, etc. necessitate an entirely new simulation.

At present, there are 7 graduate students in my research group. Four students (+ two faculty members) do micromagnetic simulations. Thus, it is not uncommon for the group to have between 300-500 simulations running simultaneously.

Not surprisingly, we need significant computational resources to manage this simulation overhead. We currently use machines managed by Notre Dame's Center for Research Computing (CRC). While access to these machines is good, there are times when our jobs will wait for 24-48 hours in a queue (as other research groups also have access to these resources).

There are times when we would like to have instantaneous access to computing resources (e.g. to meet a paper or proposal deadline, support experimental work, etc. As such, as part of a recent proposal to the Department of Defense to investigate magnetic logic systems, we included a line item in our budget that would allow for the purchase of compute nodes that are managed by the CRC, but where our research group would receive priority scheduling on said nodes[1].

We considered two compute server configurations that were available for purchase:

1. A server with 2, AMD 6134 chips (8 cores/chip @ 2.3GHz) and 24GB DDR3 RAM at $1,746.37
2. A server with 2, Intel E5620 chips (4 cores/chip @ 2.4GHz) and 24GB DDR3 RAM at $1,848.83

Given only a cursory glance, the choice of server would be somewhat obvious:

- The server with an AMD chipset offers 2X more cores that the server with an Intel chipset
- The server with an AMD chipset has a marginally lower clock rate than the server with an Intel chipset
- The server with an AMD chipset has the same amount and type of memory as the server with an Intel chipset

However, after consulting with CRC personnel, we learned that code used by certain research groups can perform significantly better on the Intel serves (even given the reduced number of cores). To determine which compute server best meets our needs, the CRC recommended running a set of representative simulations on different servers – where different compilers were also used to compile the simulation engine.

The results of the test simulations are summarized in the table below. Note that in all cases, jobs were submitted to maximize CPU usage. Thus, for the 4-core chip Intel server (with two chips), 8 single core jobs were submitted simultaneously. Similarly, for the 8-core chip AMD server (with two chips), 16 single core jobs were submitted simultaneously. (This was done to ensure that there was not an imbalance regarding main memory usage.)

[1] i.e. if we are not running jobs, other users' jobs can run on these nodes; however, if we submit jobs, our jobs will be scheduled and the jobs running on our nodes will be returned to the queue.

	Server Type	Intel - 8 Total Cores		AMD - 16 Total Cores	
	# Cores used per job	1	4	1	4
g++ Compiler	Average time per job (Hours:Minutes:Seconds)	3:45:53	1:21:55	4:41:00	2:06:51
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:31:45	10:55:20	4:41:00	8:27:25
Intel Compiler (icpc)	Average time per job (Hours:Minutes:Seconds)	3:47:00	1:18:30	4:34:38	1:51:30
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:34:00	10:28:00	4:34:38	7:26:00
PGI Compiler (pgCC)	Average time per job (Hours:Minutes:Seconds)	6:44:37	1:44:00	5:40:45	2:31:15
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	13:29:15	13:52:00	5:40:45	10:05:00
open64 Compiler (openCC)	Average time per job (Hours:Minutes:Seconds)	3:50:23	1:22:00	5:10:37	2:01:15
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:40:45	10:56:00	5:10:37	8:05:00

Given our typical usage patterns, a $25,000 budget, and the statistics summarized in the above table answer the questions below. You should justify your answer quantitatively:

- What servers (chipset + number of servers) would you recommend buying and why?
- Does the choice of compiler have a significant impact on your results?

4. Parallel Mergesort:

A challenge of multi-core computing is figuring out how to best make use of the multiple processors cores. It is not enough to simply split a program into N equal chunks for N processors. The separate sub-problems solved by each CPU core usually must communicate with each other to coordinate their work and produce the final solution. Though the product of the number of cores, and the computational power per core, may be much greater for a multicore than a single core of the same size, this inter-core communication is often very costly and can affect the design of the algorithms involved.

We will look at a very simple application of a multicore microprocessor, to solve a sorting problem with a modified mergesort algorithm. Essentially, this parallel mergesort will (1) split the input into equally-sized chunks for each core, and then (2) after each core sorts its partial list, these lists will be combined (merged) into the final answer.

Roughly, parallel mergesort looks like this:

```
Parallel_Mergesort(S)
{
      $S_1$, $S_2$, ..., $S_n$   = Partition(S, number_of_cores)

      for i = 1 to n:
                 Dispatch_Core(i, $S_i$)

      Wait_for_Cores_to_Finish()

      Sorted_$S_1$, Sorted_$S_2$, ..., Sorted_$S_n$ = Receive_From_Cores()

      return merge(Sorted_$S_1$, Sorted_$S_2$, ..., Sorted_$S_n$)
}

Core_Mergesort(sublist)
{
      if (length(sublist == 1)) : return sublist

      $S_1$, $S_2$ = Split(sublist)

      $S_1$ = Core_Mergesort($S_1$)
      $S_2$ = Core_Mergesort($S_2$)

      return Merge($S_1$, $S_2$)
}
```

(Note that Core_Mergesort is essentially the same function that you wrote in Lab 02 – and that code would be part of a multi-core implementation.)

For simplicity, presume the multi-core machines can do everything in parallel until the final merge step. All of the CPUs have the exact same instruction set and have the same rate of execution for each core, and the clock rates for all machines/cores are equal. However, the time to transfer the sorted sublists back to the master routine, etc. is given in clock cycles per list element – and does depend on the number of cores.

This code will run on an 8-core chip in 1 of 4 possible modes:

- Mode 1:
    - 1 core used.
- Mode 2:
    - 2 cores used.
    - Time to message between CPUs: 60 CCs per element must be accounted for
- Mode 3:
    - 4 cores used.
    - Time to message between CPUs: 100 CCs per element must be accounted for
- Mode 4:
    - 8 cores used.
    - Time to message between CPUs: 140 CCs per element must be accounted for

You should assume that on each core:

- the Split(L) function takes 2 CC per element in L
- the Merge(L1, L2) function takes 15 CC per element in L1 or L2.

Thus, there are N splits and N elements merged per core.

Finally, recall that the depth of recursion for a mergesort will be logarithmic (base 2) – specifically, for a power-of-two input size 2^N, there will be N+1 levels of recursion in `Core_Mergesort` (since the recursion bottoms out at N = 0, not 1.

The above information is all that you need to estimate run times.

Assume that you want to sort an N element list – where N is a power of 2, is greater than or equal to 8, and less than or equal to 4,194,304 (i.e. 4 megs). For what sized lists are Modes 1-4 most efficient? What do these results suggest if you're writing / compiling code that can be parallelized?

(Hint: write an expression for run time and graph it.)

5. <u>What</u> <u>to</u> <u>turn</u> <u>in</u>:

A **typed** lab report that contains answers to the questions in Sections 2-4.