Empirical Roofline Tool Developer Guide

Terry J. Ligocki
Leonid Oliker
Brian Van Straalen
Sam Williams
Nicholas Wright
Matthew Cordery
Wyatt Spears
Linda Lo

November 5, 2014

Contents

Chapter 1

Overview

The Roofline model of computational performance has been shown to be a useful simplification of the complexity of processor and memory systems. The goal of this work is to provide a tool that will allow people writing scientific codes and libraries to generate the Roofline graph for the computers they are using.

To achieve this an number of tasks need to completed:

- Understand the details and nuances of the generation of the Roofline model on various computer architectures and operating systems.
- Write the codes that will be used to characterize these hardware/software systems.
- Customize configuration tool(s) to automatically determine hardware/software system being tested so that the Roofline codes can be compiled and run correctly.
- Produce output in a standard form that can be utilized by other performance tools, e.g., Tau.
- Create database(s) to hold the performance information and make them accessible to a wide audience.

Chapter 2

Development

This chapter gives the details of how the Empirical Roofline Tool will be developed, tested, and released. This includes incorporating suggests and general feedback at all stages.

Much of the initial groundwork has been completed. We have experimented with a number of microkernels on a variety of machines. From these experiments, weve been able to determine which codes give accurate bandwidth and GFLOP values which can be used to generate Roofline graphs for each machine.

In addition, weve been able to structure the microkernels so they can be run with combinations of MPI processes, OpenMP threads, and GPU threads. This allows us to characterize the hardware with a variety of different forms and combinations of parallelism.

Other variations include the use of different compilers and compiler flags, the use of difference runtime environments and environment variables, and microkernels that express possible parallelism more explicitly.

For a given experiment, the current microkernels are run with a wide range of array sizes and a wide range of trials per array size. This allows the benchmarks to map out a fairly large parameter space for each microkernel. Currently, all this data is output for subsequent postprocessing.

The postprocessing consists for averaging multiple runs and extracting bandwidth or GFLOP numbers. In the case of bandwidth number, there are usually a number of plateaus which correspo0nd to the array fitting in each of the various portions of the memory hierarchy.

2.1 Next Steps

The next steps that need to be completed are:

1. Take the experimental codes that have been successful on all the machine architectures and use them to design a single framework that will be used by the Empirical Roofline Tool to generate bandwidth and GFLOP values.
2. Define what raw data will be output by the unified framework in (1), what postprocessing will be applied to the raw data, what the processed data will be, and how it will be represented/externalized.
3. Determine what parameters will be automatically configured and which will be specified by the user.
4. Develop the configuration system for the parameters that will be automatically set such that it will work on all the target systems.
5. Develop a (simple) UI to gather the parameters that will be specified by the user.
6. Test and debug the entire system on the target machines: Hopper, Edison, Mira, Titan (CPU), Babbage, Dirac, Titan (CPU+GPU).
7. Make the Empirical Roofline Tool available to other members of the Roofline Toolkit effort along with the results gathered in (6).
8. Based on what is learned and the data collected, expand the set of microkernels, vary the data access patterns, and expand the unified framework to collect a larger range of data spanning more of the multidimensional problem space.
9. Expand the configuration and testing to as many architectures and systems as possible.
10. Fix and update the tool as needed, document, package, and release.

2.2 Schedule

Here is the proposed schedule for the development outlined above:

Task	Scheduled Completion Date	Actual Completion Date
Basic investigation and experiments	End of Summer 2014	End of Summer 2014
Define data output format for all data - raw and processed	Sept. 8	Sept. 29
Determine automatic and user specified parameters		
Understand and combine experimental codes	Sept. 8	Sept. 29
Develop the configuration system	Oct. 1	Oct. 8
Develop a (simple) UI to gather user parameters		
Test and debug on target architectures	Nov. 1	N/A
Release Empirical Roofline Tool internally		
Expand coverage of microkernels, data access patterns, etc.	Dec. 1	N/A
Expand configuration and testing		
Fix and update the tool as needed, document, package, and release	Jan. 1	N/A

2.3 SC 2014

For SC 2014 the following needs to be completed. Text in gray has been completed:

1. Minimal Requirements:
    - The ERT runs on Edison using flat MPI and produces the expected roofline (which may not be optimal).

- The ERT runs on a Linux SMP "node" using pure OpenMP and produces the expected roofline.
- A basic website is set up with some information about the ERT including a user's manual and code that can be downloaded.
- Include in the user's manual appendices the rooflines we've generated for various architectures/machines.
- Include in the user's manaul examples of running the ERT, the plots it produces, etc.
- "Spec" values for bandwidth and gflop rate are added to the configuration file and propagated through to the final JSON output. We may provide a capability/option of plotting them on the roofline.
- Add the bandwidth and gflop rate numbers to the roofline.

2. Second Level Requirements:
   - The ERT runs on Edison with MPI+OpenMP over a range of MPI processes and threads and produces a near optimal roofline.
   - Add a working set size or a range of sizes to the bandwidth and gflop rate labels on the roofline.
3. Third Level Requirements:
   - The ERT runs on Babbage in native mode using the MIC with pure OpenMP and gets the expected roofline.
4. Fourth Level Requirements:
   - The ERT runs on Mira and gets the expected roofline.

2.4 List Of Work To Do

This is a list of work that could be done to improve the ERT. The work has been divided into several sections to help organize the tasks. Of course, some work spans multiple sections and some falls in no specific section.

For the former, one section was chosen. For the latter, there is the "Miscellaneous" section. The goal is to make new specific sections as needed when any section gets too large.

Also, completed work will not be deleted. Initially, it will be "grayed out" in place and, eventually, it will be move to the end of the section or to a separate section for completed work.

2.4.1 Internal Tool Development

2.4.2 Additional Tool Functionality And Options

2.4.3 Validation And Verification Of The Tool

2.4.4 Documentation

2.4.5 Releasing The Tool

Chapter 3

Usage

The Empirical Roofline Tool, ERT, is designed to allow a user to specific a variety of input parameters and their ranges (when applicable), compile and run test kernels, and produce a variety of processed output including a roofline plot for the system and configuration used.

See the Empirical Roofline Tool User's Manual.

Chapter 4

Future Work

Appendix A

Miscellaneous Details

This is a colleciton of miscellaneous details gathered from Sam Williams or found while developing and testing the Roofline Toolkit.

A.1 Compiler Flags

A.1.1 Intel Compiler

```
icc -O3 -fno-alias -fno-fnalias -openmp $ARCH

$ARCH is "-msse3" (Hopper)
         "-xAVX"  (Edison)
         "-mmic"  (Babbage/MIC/Xeon Phi/Knights Corner/KNC/...)
```

A.1.2 GCC

```
gcc -O3 -fopenmp -fno-strict-aliasing
```

A.1.3 BlueGeneQ

```
mpixlc_r -O5 -qsmp=omp:noauto ...
```

A.2 Environment Variables

A.2.1 Linux/Intel Compiler

```
setenv KMP_AFFINITY compact
                    scatter
                    balanced (MIC only)
```

Also, `KMP_AFFINITY` has verbose and granularity options so you can see what you're doing and how the hardware multithreading is changing your view of the machine, for example:

```
setenv KMP_AFFINITY=granularity=fine,compact,1,0
```

A.2.2 Linux/GCC

```
setenv GOMP_CPU_AFFINITY ???
numactl [options] -- command {arguments}
```

A.3 Running

A.3.1 Hopper

```
aprun -n 4 -d 6 -N 4 -S 1 -ss -cc numa_node <executable>
```

A.3.2 Edison

```
aprun -n 2 -d 12 -N 2 -S 1 -ss -cc numa_node <executable>
```

A.3.3 Babbage

```
setenv OMP_NUM_THREADS 240
mpirun.mic -n 1 -host <hostname> -ppn 1 <executable>
```

A.3.4 BlueGeneQ

```
qsub -t 00:10:00 -n 1 -A BGQtools_esp \
     --proccount 1 \
     --mode c1 \
     --env BG_SHAREDMEMSIZE=32MB:PAMID_VERBOSE=1:\
           BG_COREDUMPDISABLED=1:BG_SMP_FAST_WAKEUP=YES:\
           BG_THREADLAYOUT=1:OMP_PROC_BIND=TRUE:\
           OMP_NUM_THREADS=64:OMP_WAIT_POLICY=active\
     <executable>
```

You could also run 8 processes per node:

```
qsub -t 00:10:00 -n 1 -A BGQtools_esp \
     --proccount 8 \
     --mode c8 \
     --env BG_SHAREDMEMSIZE=32MB:PAMID_VERBOSE=1:\
           BG_COREDUMPDISABLED=1:BG_SMP_FAST_WAKEUP=YES:\
           BG_THREADLAYOUT=2:OMP_PROC_BIND=TRUE:\
           OMP_NUM_THREADS=8:OMP_WAIT_POLICY=active\
     <executable>
```

Where `-n` is the number of nodes, `--proccount` is the number of processes, and `--mode` is the number of processes per node.

```
setenv BG_THREADLAYOUT 2 <==> setenv KMP_AFFINITY compact
setenv BG_THREADLAYOUT 1 <==> setenv KMP_AFFINITY scatter
```

They have a balanced option but it's a kludge.

A.4 Maximum FLOP Rate

This is computed a priori with a formula resembling:

```
A * CPU-frequency * cores * M
```

```
A = 1 (simple CPU/FPU)
    2 (FMA, +/*)

M =  1 (simple CPU)
     2 (SSE)
     4 (AVX/QPX)
     8 (MIC)
    64 (K20 - 192/3)
```

To achieve this on some (most?) current architecture instruction level parallelism, ILP, (implicit or explicit) may be needed. This is necessary to make sure the internal pipelines feeding the arithmetic units stay filled.