Tyler Petrochko

CPSC 524 Parallel Programming Techniques

Assignment 1

Code submission

I’ve written all the code for this assignment in C, and used a Makefile to automate compilation. I have three separate files: pi.c, division.c, and vector.c:

pi.c	The main code outlined in Exercise 1. This calculates a value of pi using by integrating the function f(x) = 1.0 / (1.0 + x^2) from 0 to 1 using “mid-point” integration, and multiplying the result by 4. I measured the runtime performance in MFlop/s and verified that the calculation by verifying sin(pi) = 0.0. Running make builds four binary executables: `pi`, `pi2`, `pi3`, and `pi4`. These correspond to the same program compiled using the four different combinations of compiler options.
division.c	This is a slight modification to pi.c. It calculates a garbage value to be ignored, but does almost exclusively division floating-point operations. In this way, it serves as a good means to approximate the latency of the divide operation. Running make yields the binary file: `div`.
vector.c	The main code outlined in Exercise 2. This benchmarks the vector triad kernel: a[i] = b[i] + c[i] * d[i] in MFlop/s for varying vector lengths. I implemented all the design suggestions made in the description for Exercise 2. Running make yields the binary file: `vector`.

I used only the module: Langs/Intel/15. See the end of this document for output to the env command.

To compile everything, run:

```
$ module load Langs/Intel/15
```

```
$ make
```

To run the code for Exercise 1, run:

```
$ ./pi
```

```
$ ./pi2
$ ./pi3
```

```
$ ./pi4
$ ./div
```

And note the corresponding outputs.

To run the code for Exercise 2, run

```
$ ./vector
```

I've also included the scripts compile.sh and run.sh to automate this process.

Exercise 1: Division Performance

Using each of the four compiler options, I achieved the following runtime performances:

`-g -O0 -fno-alias -std=c99`	834 MFlop/s
`-g -O1 -fno-alias -std=c99`	834 MFlop/s
`-g -O3 -no-vec -no-simd -fno-alias -std=c99`	835 MFlop/s
`-g -O3 -xHost -fno-alias -std=c99`	1669 MFlop/s

Firstly, all of these compiler options use the no pointer aliasing option, which allows the compiler to parallelize array operations without worrying about corrupting array data due to obscure data dependencies. However, no arrays are used in this exercise and thus this option makes no substantial difference in runtime. To confirm this, I tried removing the `–fno-alias` option and saw no substantial difference in runtime performance. The `–g` flag includes debug information along with the object files. As a side effect, this also defaults the optimization level to level 0.

With the first compiler options, the `–O0` flag specifies that no optimization should be used (optimization level 0). Because `–g` was also specified, this flag is redundant. Because no optimization is employed, the compiler uses the most naïve implementation strategy and understandably yields the slowest running executable.

In this case, the second compiler options yield no substantial speed increase. The `–O1` flag disables optimizations that substantially increase binary size (like unrolling loops). As specified in the Intel man page, these options may improve performance for applications with "very large code size." My implementation is fewer than 40 lines, so it makes sense that there's no runtime benefit.

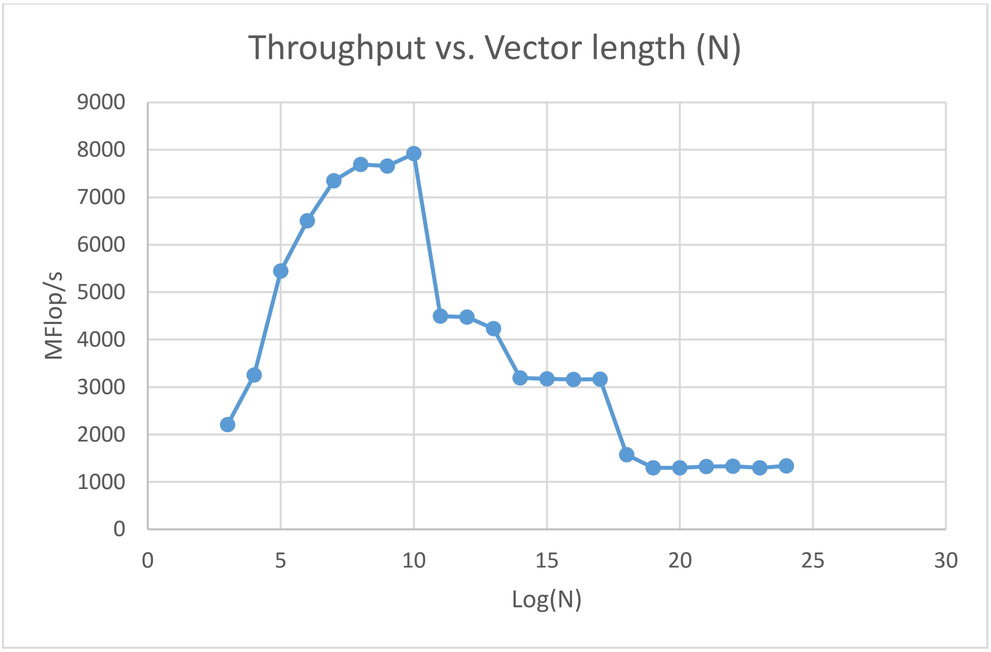
The third set of options uses the highest level of optimization, but disables automatic vectorization. The `–O3` optimization level enables "aggressive" optimizations like loop prefetching and loop transformations. The Intel man page suggests that this level of optimization is best for programs with looped floating point calculations, which describes our program well; however, the `-no-vec -no-simd` options disable vector optimizations, preventing the processor from doing batch operations on vectorized data (SIMD). This results in minimal performance benefit.

The last compiler options use the highest level of optimization and allow the use of native extensions to the x86 instruction set used on Intel chips via the `–xHost` flag. Most importantly, omission of the `–no-vec` flag allows automatic vectorization, an optimization technique that converts a sequential processing task (via a loop, perhaps) into a parallel one via batch vector operations. Impressively, this combination of compiler options is intelligent enough to convert my `f(x)` function into a set of inline looped instructions, and then convert the resulting task to a vector operation. This results in a huge increase in performance, as many `f(x)` computations are occurring in parallel rather than in sequential order.

To estimate the latency of the divide operation in CPU cycles, I modified the pi-computing program such that the majority of the floating-point operations were division operations. I.e. I modified `f(x)` to calculate the reciprocal (one single division operation), and remove all multiplication floating-point operations. Based on the assumption that division operations take drastically more CPU cycles than additions, we can ignore the time contribution of the addition floating-point operations.

Running this program with the first set of compilation instructions yielded about 139 MFlop/s. Assuming a 2.8 GHz processor, we know a cycle is $\frac{1\,s}{2.8*10^9} \approx 3.57 * 10^{-10}s = .357\,ns$. If we assume no vector optimizations are allowed to take place and neglect the throughput benefits of pipelining (slide 27 of lecture #2 suggests that this is a fair assumption, given pipelining only eliminates one cycle per division instruction), we can divide the time per division operation by the time per CPU cycle, yielding a latency of $\frac{\frac{1}{139*10^6}}{3.57*10^{-10}} \approx \mathbf{20.2}$ CPU cycles per division operation. This is half as many CPU cycles as listed in the lecture notes, but seems to be on the right order of magnitude.

<u>**Exercise 2: Vector Triad Performance**</u>



In the throughput plot, we notice an upward trend in throughput as the vector length approaches 2.1^{10} floats. Given a float is 4 bytes and 4 vectors are being processed simultaneously, this is about $\frac{4*4*2.1^{10}\ bytes}{1000\frac{bytes}{KB}} \approx 26.7\ KB$ of data. Given that an Intel Core i7 includes two 32-KB L1 caches, it makes sense that this yields peak performance; all of the data can comfortably reside in L1 cache, while the large vector size allows for high amounts of processing concurrency via compiler optimizations such as automatic vectorization and SIMD. At $N = 2.1^{11}$, the four vectors take about 56 KB, which makes it difficult to store all the vectors in L1 cache (given the fudge-factor of set-associative caches and overhead of other runtime data), resulting in many L1 cache misses.

These cache misses require loading data from L2 cache, which explains the sharp decrease in performance. As the vectors get larger, performance stays about constant until about $N = 2.1^{13}$, or equivalently 247 KB of floating point data. At this point, most floating point operations require data to be loaded from the 256 KB unified L2 cache. When the vectors are roughly doubled at $N = 2.1^{14}$, the 519 KB of vectorized data can no longer be stored in L2 cache and must spill into the 8 MB L3 cache. This explains the second sharp decline in throughput.

This trend occurs a third time between $N = 2.1^{17}$ and $N = 2.1^{18}$, at which the point the vector data increases from 4.8 MB to 10 MB, and data must spill from L3 cache to main memory. This explains the final decline in throughput. Beyond this point, the throughput remains constant as most floating point operations require data to be loaded from main memory.

We can approximate the access bandwidth of each level of memory using the equation $Bandwidth\left(\frac{GB}{s}\right) = \frac{GFlop}{s} * \frac{bytes}{flop}$. We assume that for $N = 2.1^{10}$, most data is loaded from L1 cache, for $N = 2.1^{12}$ most data is loaded from L2 cache, for $N = 2.1^{14}$ most data is loaded from L3 cache, and for $N = 2.1^{24}$ most data is loaded from main memory.

L1	31.6 GB/s
L2	17.9 GB/s
L3	12.8 GB/s
Main memory	5.4 GB/s

These calculations seem roughly four times the throughputs proposed in lecture #3, so it's possible these load operations are being performed in parallel.

Environment

```
MKLROOT=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mkl

MANPATH=/opt/moab/share/man:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.
2.164/man/en_US:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugg
er/gdb/intel64/share/man/:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.
164/debugger/gdb/intel64_mic/share/man/:/usr/share/man:/opt/moab/share/man:

GDB_HOST=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger/gdb/
intel64_mic/bin/gdb-ia-mic

HOSTNAME=compute-33-1.local

INTEL_LICENSE_FILE=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/lic
enses:/opt/intel/licenses:/home/apps/fas/Licenses/intel_site.lic

IPPROOT=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp

SHELL=/bin/bash

TERM=xterm

HISTSIZE=1000

SSH_CLIENT=10.191.63.252 47029 22

GDBSERVER_MIC=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger
/gdb/target/mic/bin/gdbserver

PERL5LIB=/opt/moab/lib/perl5:/opt/moab/lib/perl5

LIBRARY_PATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp/../co
mpiler/lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp
/lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/compiler/
lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mkl/lib/in
tel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/tbb/lib/intel64/
gcc4.4

FPATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mkl/include

OLDPWD=/lustre/home/client/fas/cpsc424/thp8
```

```
QTDIR=/usr/lib64/qt-3.3

QTINC=/usr/lib64/qt-3.3/include

SSH_TTY=/dev/pts/9

MIC_LD_LIBRARY_PATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mp
irt/lib/mic:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp/lib/mi
c:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/compiler/lib/mic:/ho
me/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mkl/lib/mic:/opt/intel/mi
c/coi/device-linux-
release/lib:/opt/intel/mic/myo/lib:/home/apps/fas/Langs/Intel/2015_update2/composer_x
e_2015.2.164/tbb/lib/mic

ANT_HOME=/opt/rocks

USER=thp8

LD_LIBRARY_PATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mpirt/
lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp/../com
piler/lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp/
lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp/tools/
intel64/perfsys:/opt/intel/mic/coi/host-linux-
release/lib:/opt/intel/mic/myo/lib:/home/apps/fas/Langs/Intel/2015_update2/composer_x
e_2015.2.164/compiler/lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe
_2015.2.164/mkl/lib/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.
2.164/tbb/lib/intel64/gcc4.4:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015
.2.164/debugger/ipt/intel64/lib

ROCKS_ROOT=/opt/rocks

MIC_LIBRARY_PATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/compi
ler/lib/mic:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mpirt/lib/
mic:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/tbb/lib/mic

CPATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/ipp/include:/hom
e/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mkl/include:/home/apps/fas
/Langs/Intel/2015_update2/composer_xe_2015.2.164/tbb/include

NLSPATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/compiler/lib/i
ntel64/locale/%l_%t/%N:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164
/ipp/lib/intel64/locale/%l_%t/%N:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_
2015.2.164/mkl/lib/intel64/locale/%l_%t/%N:/home/apps/fas/Langs/Intel/2015_update2/co
mposer_xe_2015.2.164/debugger/gdb/intel64_mic/share/locale/%l_%t/%N:/home/apps/fas/La
ngs/Intel/2015_update2/composer_xe_2015.2.164/debugger/gdb/intel64/share/locale/%l_%t
/%N

YHPC_COMPILER=Intel

PATH=/opt/moab/bin:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/bin
/intel64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mpirt/bin/int
el64:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger/gdb/inte
l64_mic/bin:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger/g
db/intel64/bin:/home/apps/fas/Modules:/usr/lib64/qt-
3.3/bin:/opt/moab/bin:/usr/local/bin:/bin:/usr/bin:/usr/java/latest/bin:/opt/rocks/bi
```

```
n:/opt/rocks/sbin:/home/apps/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/fas/cpsc424/th
p8/bin

MAIL=/var/spool/mail/thp8

YHPC_COMPILER_MINOR=164

TBBROOT=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/tbb

PWD=/lustre/home/client/fas/cpsc424/thp8/as1

F90=ifort

JAVA_HOME=/usr/java/latest

YHPC_COMPILER_MAJOR=2

_LMFILES_=/home/apps/fas/Modules/Base/yale_hpc:/home/apps/fas/Modules/Langs/Intel/15

LANG=en_US.iso885915

DOMAIN=omega

GDB_CROSS=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger/gdb
/intel64_mic/bin/gdb-mic

MOABHOMEDIR=/opt/moab

MODULEPATH=/home/apps/fas/Modules

KDEDIRS=/usr

LOADEDMODULES=Base/yale_hpc:Langs/Intel/15

YHPC_COMPILER_RELEASE=2015

F77=ifort

HISTCONTROL=ignoredups

SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass

CXX=icpc

MPM_LAUNCHER=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger/
mpm/bin/start_mpm.sh

INTEL_PYTHONHOME=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debug
ger/python/intel64/

HOME=/home/fas/cpsc424/thp8

SHLVL=2

FC=ifort

LOGNAME=thp8

CVS_RSH=ssh

QTLIB=/usr/lib64/qt-3.3/lib
```

```
SSH_CONNECTION=10.191.63.252 47029 10.191.12.33 22

MODULESHOME=/usr/share/Modules

LESSOPEN=||/usr/bin/lesspipe.sh %s

arch=intel64

INFOPATH=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/debugger/gdb/
intel64/share/info/:/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/de
bugger/gdb/intel64_mic/share/info/

CC=icc

INCLUDE=/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/mkl/include

G_BROKEN_FILENAMES=1

BASH_FUNC_module()=() {  eval `/usr/bin/modulecmd bash $*`

}

_=/bin/env
```