The Intel Xeon Phi

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

Oregon State University
Computer Graphics
mjb – April 24, 2017

1

Setup

Xeon system
rabbit.engr.oregonstate.edu
2 ES-2630 Xeon Processors
8 Cores
64 GB of memory
2 TB of disk

Xeon Phi support:
ic, icpc, libraries, drivers

31S1P Xeon Phi system

PCIe Bus

“mic0”

57 Cores
22 nm
8 GB of memory
No disk
Application support

NVIDIA Titan Black

PCIe Bus

15 SMs
2880 CUDA cores
6 GB of memory
OpenGL support
OpenCL support

1 core for Linux +
56 cores * 4 hyperthreads/core =
224 hyperthreads for you to use

6/14/2017

2

Fused Multiply-Add

Many scientific and engineering computations take the form:
D = A + (B*C);

A “normal” multiply-add would handle this as:
tmp = B*C;
D = A + tmp;

A “fused” multiply-add does it all at once, that is, when the low-order bits of B*C are ready, they are immediately added into the low-order bits of A at the same time the higher-order bits of B*C are being multiplied.

Consider a Base 10 example: 789 + ( 123*456 )

123
\_x 456
738
615
492
+ 789
56,877

Can start adding the 9 the moment the 8 is produced!

Note: “Normal” A+(B*C) ≠ “FMA” A+(B*C)

3

Xeon Phi Internals

Each Xeon Phi core has:

Instruction Decode
32K L1 Instruction Cache

Scalar Unit
Vector Unit

Scalar Registers
Vector Registers

32K L1 Data Cache

512K L2 Cache / Core

Ring Bus

Vector registers are 512 bits wide = 16 floats.
They can perform Fused Multiply-Add (FMA).
Theoretical performance = almost 1 TFLOPS

The Xeon Phi chip contains almost 5B transistors!

Cache is 8-way set associative.
Cache lines are 64 bytes.

Assembly instructions are executed in-order.
Thus, hyperthreading is important!

Getting to rabbit and setting up your account

To verify that the Xeon Phi card is there:
ping mic0

To see the Xeon Phi card characteristics:
micinfo

Getting to rabbit and setting up your account

To run some operational tests on the Xeon Phi:
miccheck

Note: “Normal” A+(B*C) ≠ “FMA” A+(B*C)

5

Xeon Phi Peak Performance

Clock freq x # cores x # vector lanes x 2 FMA / 2 cycles to decode =
1.091 GHz x 56 x 16 x 2 / 2 =
0.98 TFLOPS

FMA stands for “Fused Multiply-Add”.

To login to rabbit:
lsb rabbit.engr.oregonstate.edu -you

Put this in your rabbit account's .cshrc:

source /nfs/guille/a2/rh80apps/intel/studio.2013-sp1/bin/iccvars.csh intel64

Then activate these values like this:
sourc .cshrc

(These will be activated automatically the next time you login.)

To see the Xeon Phi card characteristics:
micinfo

To run some operational tests on the Xeon Phi:
miccheck

6

Note: “Normal” A+(B*C) ≠ “FMA” A+(B*C)
Running ping

PING rabbit-mic0.engr.oregonstate.edu (172.31.1.1) 56(84) bytes of data.
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=1 ttl=64 time=290 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=5 ttl=64 time=0.225 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=3 ttl=64 time=0.242 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=2 ttl=64 time=0.385 ms

Test 6 (mic0): Check running flash version is correct ... pass
Test 5 (mic0): Check ras daemon is available in device ... pass
Test 3: Check mpssd daemon is running ... Pass
Test 2: Check number of devices driver sees in the system ... pass
Test 1: Check mic driver is loaded ... pass
Test 0: Check number of devices the OS sees in the system ... pass

Cross-compiling and running from rabbit

To compile on rabbit for rabbit:
kpc -o try trycpp -m-openmp -align -oapl-report3 -oapl-report-phase=vec

To cross-compile on the Xeon Phi:
kpc -mic -o try trycpp -m-openmp -align -oapl-report3 -oapl-report-phase=vec

Note: the summary of vectorization success or failure is in a *.optvec file

To execute on the Xeon Phi, type this on:
microsvecindex try
Gaining Access to the Cores, I

```c
float sum = 0.;
#pragma omp parallel for reduction(+:sum)
for( int i = 0; i < N; i++ )
    sum += A[i] * B[i];
```

Gaining Access to the Cores, II

```c
#pragma omp parallel sections
#pragma omp section
. . .
#pragma omp section
. . .
#pragma omp task
. . .
```

Gaining Access to the Vector Units

```c
#pragma omp simd
for( int i = 0; i < N; i++ )
    C[i] = A[i] * B[i];
#pragma omp parallel for simd
for( int i = 0; i < N; i++ )
    C[i] = A[i] * B[i];
```

Turning Off All Vectorization

```c
icpc -mmic -o try try.cpp -lm -openmp -align -qopt-report=3 -qopt-report-phase=vec
micnativeloadex try
```

Vectorizing Conditionals

```c
#pragma omp simd
for( int i = 0; i < N; i++ )
{
    if( D[i] == 0 )
        C[i] = A[i] * B[i];
    else
        C[i] = A[i] + B[i];
}
```

Reducing a Vector

```c
float f = __sec_reduce_add(A[0:N]);
float f = __sec_reduce_mul(A[0:N]);
float f = __sec_reduce_max(A[0:N]);
float f = __sec_reduce_min(A[0:N]);
int i = __sec_reduce_max_ind(A[0:N]);
int i = __sec_reduce_min_ind(A[0:N]);
boolean b = __sec_reduce_all_zero(A[0:N]);
boolean b = __sec_reduce_all_nonzero(A[0:N]);
boolean b = __sec_reduce_any_zero(A[0:N]);
boolean b = __sec_reduce_any_nonzero(A[0:N]);
```

You must specify the array length. An argument of `A[0]` will throw a compiler error.
Reducing a Vector

float sum = 0.;
for (int i = 0; i < N; i++)
{
    sum += A[i];
}

In my tests, this was the same speed as this.

float sum = __sec_reduce_add(A[0:N]);

Offload Mode

You don't need to do anything special with the compile line:

kpc -o try try.cpp -lm -openmp -align -qopt-report=3 -qopt-report-phase=vec

Try

Alignment

To ensure alignment, replace this

float Temperature[NUMN];

with this:

#define ALIGN64 __declspec(align(64))

* * *

ALIGN64 float Temperature[NUMN];

Alignment

To ensure alignment, replace this

float *A = (float *) malloc(NUMS*sizeof(float));
float *B = (float *) malloc(NUMS*sizeof(float));
float *C = (float *) malloc(NUMS*sizeof(float));

with this

float *A = (float *)__mm_malloc(NUMS*sizeof(float), 64);
float *B = (float *)__mm_malloc(NUMS*sizeof(float), 64);
float *C = (float *)__mm_malloc(NUMS*sizeof(float), 64);

You then free memory with:

__mm_free(A);
__mm_free(B);
__mm_free(C);
Alignment

If you want to ensure alignment, but still want to use C++'s new and delete, replace this:

```c++
float *A = new float [NUMS];
float *B = new float [NUMS];
float *C = new float [NUMS];
```

with this:

```c++
float *pa = (float *) _mm_malloc( NUMS*sizeof(float), 64 );
float *pb = (float *) _mm_malloc( NUMS*sizeof(float), 64 );
float *pc = (float *) _mm_malloc( NUMS*sizeof(float), 64 );
```

float *A = new(pa) float [NUMS];
float *B = new(pb) float [NUMS];
float *C = new(pc) float [NUMS];

You then free memory with:

```c++
delete [] A;
delete [] B;
delete [] C;
```

As You Create More and More Threads, On Which Cores Do They End Up?

If you want them spread out onto as many cores as possible, execute this:

```c++
kmp_set_defaults( "KMP_AFFINITY=scatter" );
```

If you want them packed onto the first core until it has 4, than onto the second core until it has 4, etc., execute this:

```c++
kmp_set_defaults( "KMP_AFFINITY=compact" );
```

An advantage of using new and delete instead of malloc is that they allow you to use C++ constructors and destructors.

Running the Volume-Integration Program

# of Threads

MegaHeights per Second

# of Divisions

Multicore, no vectorization

Reservation System

https://secure.engr.oregonstate.edu/engr/resources/bailey