The Intel Xeon Phi

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

The Xeon Phi chip contains almost 5 billion transistors!

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

FMA stands for "Fused Multiply+Add". It allows the operation:
\[ d = a \times b + c; \]
to be performed in the same amount of time as:
\[ d = a \times b; \]

To login to rabbit:
ssh rabbit.engr.oregonstate.edu -l yourusername

Put this in your rabbit account's .cshrc:

```
setenv INTEL_LICENSE_FILE 28518@linlic.engr.oregonstate.edu
setenv ICCPATH /nfs/guille/a2/rh80apps/intel/studio.2013-sp1/composer_xe_2015/bin/
set path=( $path $ICCPATH )
source /nfs/guille/a2/rh80apps/intel/studio.2013-sp1/bin/iccvars.csh intel64
```

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
```

To run some operational tests on the Xeon Phi:

```
miccheck
```

Running ping

```
rabbit 100% ping mic0
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=2 ttl=64 time=0.385 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=4 ttl=64 time=0.230 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=6 ttl=64 time=0.261 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=7 ttl=64 time=0.259 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=8 ttl=64 time=0.256 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=9 ttl=64 time=0.252 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=10 ttl=64 time=0.249 ms
64 bytes from rabbit-mic0.engr.oregonstate.edu (172.31.1.1): icmp_seq=11 ttl=64 time=0.246 ms
```

Then activate these values like this:

```
source .cshrc
```

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

To verify that the Xeon Phi card is there:

```
ping mic0
```

To see the Xeon Phi card characteristics:

```
micinfo
```

To run some operational tests on the Xeon Phi:

```
miccheck
```

Setup

NVIDIA Titan Black

PCIe Bus

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

The Xeon Phi system

rabbit.engr.oregonstate.edu
2 E5-2630 Xeon Processors
8 Cores
64 GB of memory
2 TB of disk

OpenCL support

Xeon Phi support:
icc, icpc, libraries, drivers

31S1P Xeon Phi system

PCIe Bus

“mic0”
57 Cores
22 nm
8 GB of memory
No disk

Application support

Xeon Phi Internals

Each Xeon Phi core has:
The Xeon Phi chip contains almost 5 billion transistors!

Vector registers are 512 bits wide = 16 floats.

They can perform Fused Multiply-Add (FMA).
Theoretical performance = almost 1 TFLOPS

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

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

FMA stands for "Fused Multiply+Add". It allows the operation:

\[ d = a \times b + c; \]
to be performed in the same amount of time as:

\[ d = a \times b; \]

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

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". It allows the operation:

\[ d = a \times b + c; \]
to be performed in the same amount of time as:

\[ d = a \times b; \]
Cross-compiling and running from rabbit

To compile on rabbit for Xeon Phi:
```bash
kmp -O3 -m -openmp -align -qopt-report=3 -qopt-report-phase=vec
```

To cross-compile on rabbit for the Xeon Phi:
```bash
kmp -O3 -m -openmp -align -qopt-report=3 -qopt-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 rabbit:
```bash
miccheck
```

Gaining Access to the Cores, I

```
for( int i = 0; i < N; i++ )
float sum = 0.;
for( int i = 0; i < N; i++ )
    sum += A[i] * B[i];
```
Gaining Access to the Cores, II

#pragma omp parallel sections
#pragma omp parallel section

#pragma omp parallel sections
#pragma omp parallel section

#pragma omp task

...


### Elemental Vector Functions

```c
__declspec(vector)
float f(float x, float y)
{
    return x*y;
}
```

In my tests, this was 3x as fast as this.

### Offload Mode

```c
float vmul(float x, float y)
{
    return x*y;
}
```

For( int i = 0; i < N; i++ )
{
    C[i] = vmul(A[i], B[i]);
}

```c
#define ALLOC    alloc_if(1)
#define REUSE    alloc_if(0)
#define RETAIN   free_if(0)
#define FREE       free_if(1)
```

### Offload Mode: Persistence Between Offloads

```c
#pragma offload target(mic) in(A:length(NUMS), ALLOC, RETAIN) out(C:length(NUMS), ALLOC, FREE)
```

### Alignment

To ensure alignment, replace this

```c
float Temperature[NUMN];
```

with this:

```c
#define ALIGN64 __declspec(align(64))

ALIGN64 float Temperature[NUMN];
```

You then free memory with:

```c
_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:

float *A = new float [NUMS];
float *B = new float [NUMS];
float *C = new float [NUMS];

with this

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:

delete [] A;
delete [] B;
delete [] C;

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

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

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

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:

kmp_set_defaults( "KMP_AFFINITY=compact" );

Use the scatter-mode if you want as much core-power applied to each thread as possible.
Use the compact-mode if there is an advantage to some threads sharing a core's local memory with other threads.

Running the Volume-Integration Program

Multicore, no vectorization

Reservation System

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