Accelerators in Abel

Ole W. Saastad, Dr.Scient
UiO/USIT / UAV/ ITF/FI
March 28th 2014
Background, what is an accelerator?

The short explanation is that it's a device where most of the transistors are used for computation.

A calculating device more than a general multi purpose processor.

By using all transistors for calculations very high performance can be achieved.
Accelerators are not new

Why the new FPS-164 Attached Processor creates such interest

Math Co-Processor
8087

Universitetets senter for informasjonsteknologi
Top 500 supercomputers

• Of the top 500 systems, 53 now use accelerators

• 4 of the top 10 uses accelerators

• HPL performance

Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P

<table>
<thead>
<tr>
<th>Site:</th>
<th>National Super Computer Center in Guangzhou</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer:</td>
<td>NUDT</td>
</tr>
<tr>
<td>Cores:</td>
<td>3,120,609</td>
</tr>
<tr>
<td>Linpack Performance (Rmax):</td>
<td>33,862.7 TFlop/s</td>
</tr>
<tr>
<td>Theoretical Peak (Rpeak):</td>
<td>54,902.4 TFlop/s</td>
</tr>
<tr>
<td>Power:</td>
<td>17,808.09 kW</td>
</tr>
<tr>
<td>Memory:</td>
<td>1,024,609 GB</td>
</tr>
<tr>
<td>Interconnect:</td>
<td>TH Express-2</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Kylin Linux</td>
</tr>
<tr>
<td>Compiler:</td>
<td>gcc</td>
</tr>
<tr>
<td>Math Library:</td>
<td>Intel MKL-11.0.6</td>
</tr>
<tr>
<td>MPI:</td>
<td>MPICH2 with a customized GLEX channel</td>
</tr>
</tbody>
</table>
Benchmark – user fortran code

MxM offloading
Fortran 90 code, double prec.

Host procs
Co-processor

Performance [Gflops]
Memory footprint matrices

0  5  10  15  20  25  30  35  40
2288 MiB  5149 MiB  5859 MiB  6614 MiB
Accelerators in Abel

- NVIDIA K20x
  - 16 nodes with two each
  - 32 GPUs in total

- Intel Xeon Phi, 5110P
  - 4 nodes with two each
  - 8 MIC systems in total
NVIDIA Kepler K20
Kepler K20, processor GK110

Kepler GK110 Block Diagram

Architecture
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3
Kepler K20 architecture
Kepler K20 architecture
Kepler K20 architecture
# Kepler K20 Performance

<table>
<thead>
<tr>
<th>Feature</th>
<th>Tesla K20X</th>
<th>Tesla K20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak Double Precision</td>
<td>1.31 TF</td>
<td>1.17 TF</td>
</tr>
<tr>
<td>Peak Single Precision</td>
<td>3.95 TF</td>
<td>3.52 TF</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>250 GB/s</td>
<td>208 GB/s</td>
</tr>
<tr>
<td>Memory size</td>
<td>6 GB</td>
<td>5 GB</td>
</tr>
</tbody>
</table>

Universitetets senter for informasjonsteknologi
GPU performance K20Xm

DGEMM performance GPU vs. CPU
Tesla K20X vs Intel SB

Double precision 64 bit,
1 Tflops/s

Single precision 32 bit,
2.6 Tflops/s
Accelerators hype or production?

**VISIBILITY**

- Peak of Inflated Expectations
- Plateau of Productivity
- Roundabout of Repackaging
- Slope of Enlightenment
- Swamp of Continued Use
- Trash Heap of Failures

**TIME**
Exploiting the GPUs

• Pre compiled applications
  – NAMD, MrBayes, Beagle, LAMMPS etc

• CUDA libraries
  – BLAS, Sparse matrices, FFT

• Compiler supporting accelerator directives
  – PGI support accelerator directives
NAMD 2.8 and 2.9

• GPU enabled
• Easy to run

charmrun namd2 +idlepoll +p 2 ++local +devices 0,1 input.inp

Speedup : 122/39 = 3.1x
LAMMPS

- GPU enabled
- Easy to run

```
mpirun lmp_cuda.double.x -sf gpu -c off -v g 2 -v x 128 -v y 128 -v z 128 -v t 1000 in.lj.gpu
```

Speedup 720/250 = 2.9x
Running applications with GPUs

Example using lammps:

#SBATCH --job-name=lammps --account=proj --nodes=2
#SBATCH --ntasks-per-node=8 --mem-per-cpu=7800M
#SBATCH --partition=accel --gres=gpu:2 --time=01:00:00

. /cluster/bin/jobsetup
module load lammps/2013.08.16
module load cuda/5.0

EXE=lmp_cuda.double.x
OPT="-sf gpu -c off -v g 2 -v x 128 -v y 128 -v z 128 -v t 1000"
INPUT=in.lj.gpu

mpirun $EXE $OPT < $INPUT
CUDA libraries – easy access

• Precompiled, just linking
  – BLAS
  – Sparse
  – FFT
  – Random
  – Some extras, ref. doc.
CUDA libraries

From fortran 90:

\begin{verbatim}
call cublas_dgemm('n', 'n', N, N, N, alpha, a, N, b, N, beta, c, N)
\end{verbatim}

Same syntax as standard dgemm

Compile and link:

\begin{verbatim}
gfortran -o dgemmdriver.x -L/usr/local/cuda/lib64 /usr/local/cuda/lib64/fortran_thunking.o -lcublas dgemmdriver.f90
\end{verbatim}

Interfaces hides the cuda syntax.
CUDA libraries

Performance in Gflops/s

<table>
<thead>
<tr>
<th>N</th>
<th>Footprint MB</th>
<th>CUDA BLAS</th>
<th>MKL BLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>91</td>
<td>3.29</td>
<td>34.14</td>
</tr>
<tr>
<td>4000</td>
<td>366</td>
<td>24.94</td>
<td>61.96</td>
</tr>
<tr>
<td>8000</td>
<td>1464</td>
<td>159.7</td>
<td>71.44</td>
</tr>
<tr>
<td>12000</td>
<td>3295</td>
<td>345.55</td>
<td>72.09</td>
</tr>
<tr>
<td>15000</td>
<td>5149</td>
<td>482.15</td>
<td>72.56</td>
</tr>
</tbody>
</table>

DGEMM performance GPU vs. CPU

Speedup 482/73 = 6.6x
Open ACC – very easy to get started

OpenACC

Directives for Accelerators

OpenACC Accelerates Science

Weather Prediction

Chemistry Research

Fuel Efficiency

Oil Exploration

COSMO (Physics)

GAMESS CCSD

S3D

ELAN

4.2x

3.1x

2.2x

3.2x

Linux GCC Compiler to Support GPU Accelerators

Open Source

OpenACC in GCC by Mentor Graphics & Samsung

Pervasive Impact

Free to all Linux users

Mainstream

Most Widely Used HPC Compiler

"Incorporating OpenACC into GCC is an excellent example of open source and open standards working together to make accelerated computing broadly accessible to all Linux developers."

Oscar Hernandez
Oak Ridge National Laboratories

Universitetets senter for informasjonsteknologi
Open accelerator initiative info

- www.openacc-standard.org
- www.pgroup.com
- en.wikipedia.org/wiki/OpenACC
- developer.nvidia.com/openacc
Open ACCelerator initiative

Directives inserted into your old code

```plaintext
Program myscience
    ... serial code ...
    !$acc parallel loop
    do k = 1,n1
        do i = 1,n2
            ... parallel code ...
            enddo
        enddo
    !$acc end parallel loop
End Program myscience
```

Compiled Hint

CPU

GPU

Your original code
Compiler supporting OpenACC

• Portland (PGI), pgcc, pgfortran, pgf90
  – Installed on Abel

• CAPS HMPP
  – Not installed on Abel
  – Commercial, rather expensive

• GCC (soon)
  – in version 5.0
Compiler supporting OpenACC

fortran 90 code:

SUBROUTINE DGEMM_acc

!$acc region
   DO 90 J = 1,N
      IF (BETA.EQ.ZERO) THEN
         DO 50 I = 1,M
            C(I,J) = ZERO
      50   CONTINUE

90 CONTINUE

!$acc end region

Compile and link :

pgfortran -o dgemm-test.x -ta=nvidia,kepler dgemm.f dgemmtest.f90
Running accelerated code

dgemm

Performance in Gflops/s

<table>
<thead>
<tr>
<th>N</th>
<th>Footprint MB</th>
<th>PGI Accel</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>91</td>
<td>2.33</td>
<td>2.51</td>
</tr>
<tr>
<td>4000</td>
<td>366</td>
<td>9.48</td>
<td>2.17</td>
</tr>
<tr>
<td>8000</td>
<td>1464</td>
<td>14.03</td>
<td>2.2</td>
</tr>
<tr>
<td>12000</td>
<td>3295</td>
<td>14.09</td>
<td>2.2</td>
</tr>
<tr>
<td>15000</td>
<td>5149</td>
<td>11.85</td>
<td>1.79</td>
</tr>
</tbody>
</table>

Accelerated F77 code vs plain F77
Portland accelerator directives, dgemm

Accelerated F77 code vs plain F77
Portland accelerator directives, sgemm

Universitetets senter for informasjonsteknologi
CUDA stands for «Compute Unified Device Architecture». CUDA is a parallel computing architecture and C based programming language for general purpose computing on NVIDIA GPU's.

- Programming from scratch, special syntax for GPU
- Works only with NVIDIA
CUDA - NVIDIA

CUDA C

Standard C Code

```c
void saxpy_serial(int n,
    float a,
    float *x,
    float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);

Parallel C Code

```c
__global__
void saxpy_parallel(int n,
    float a,
    float *x,
    float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);

Programming with CUDA

// Run kernel
dim3 blockDim(16, 16, 1);
dim3 gridDim((width + blockDim.x - 1)/ blockDim.x, (height + blockDim.y - 1) / blockDim.y, 1);
kernel<<< gridDim, blockDim, 0 >>>(d_data, height, width);

__global__ void kernel(float* odata, int height, int width)
{
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        float c = tex2D(tex, x, y);
        odata[y*width+x] = c;
    }
}
CUDA 6.0 the new version
OpenCL™ is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software.

OpenCL 2.0
OpenCL 2.0 is the latest significant evolution of the OpenCL standard, designed to further simplify cross-platform programming while enabling a rich range of algorithms and programming patterns to be easily accelerated. As the foundation for those increased capabilities, OpenCL 2.0 defines an enhanced execution model and a subset of the C and C++11 memory model, synchronization and atomic operations.

OpenCL 1.2
- The OpenCL 1.2 specification and header files are available in the Khronos Registry
- The OpenCL 1.2 Quick Reference card (View online)
- The OpenCL 1.2 Online Manual
- The OpenCL C/C++ Wrapper 1.2 Reference Card (View online)

OpenCL 1.1
- The OpenCL 1.1 specification and header files are available in the Khronos Registry
- The OpenCL 1.1 Quick Reference card (View online)
- The OpenCL 1.1 Online Manual

OpenCL 1.0
- The OpenCL 1.0 specification and header files are available in the Khronos Registry
- The OpenCL 1.0 Quick Reference card (View online)
- The OpenCL 1.0 Online Manual
OpenCL language

• Open Compute Language
  – Support for a range of processors incl x86-64

• An open standard supported by a multiple of vendors

• Complexity comparable to CUDA

• Performance comparable to CUDA
OpenCL

// create the compute kernel
    kernel = clCreateKernel(program, "fft1D_1024", NULL);

// set the args values
    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
    clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16, NULL);
    clSetKernelArg(kernel, 3, sizeof(float)*(local_work_size[0]+1)*16, NULL);

__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy) {

    int tid = get_local_id(0);
    int blockIdx = get_group_id(0) * 1024 + tid;
    float2 data[16];

    // starting index of data to/from global memory
    in = in + blockIdx;  out = out + blockIdx;
    globalLoads(data, in, 64); // coalesced global reads
    fftRadix16Pass(data);     // in-place radix-16 pass
    twiddleFactorMul(data, tid, 1024, 0);

    // local shuffle using local memory
    localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
    fftRadix16Pass(data);     // in-place radix-16 pass
    twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication

    localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));

    // four radix-4 function calls
    fftRadix4Pass(data);      // radix-4 function number 1
    fftRadix4Pass(data + 4);  // radix-4 function number 2
    fftRadix4Pass(data + 8);  // radix-4 function number 3
    fftRadix4Pass(data + 12); // radix-4 function number 4
Running jobs / SLURM - GPUs

• Request both GPUs
  – Qlogin --account xx --partition=accel
    --gres=gpu:2 --nodes=1 --ntasks-per-node=8

• #SBATCH --nodes=1 --ntasks-per-node=8
  – Reserve all resources for your job

• #SBATCH --partition=accel --gres=gpu:2
Intel Xeon Phi – MIC architecture
Outstanding performance

Theoretical performance:

Clock frequency 1.05 GHz
60 cores (x60)
8 dp entry wide Vector (x8)
FMA instruction (x2)

1.05*60*8*2 = 1008 Gflops/s

1 Tflops/s on a single PCIe card
MIC architecture

- 60 physical cores, x86-64 in order execution
- 240 hardware threads
- 512 bits wide vector unit (8x64 bit or 16x32 bit floats)
- 8 GiB GDDR5 main memory in 4 banks
- Cache Coherent memory (directory based, TD)
- Limited hardware prefetch
- Software prefetch important
MIC architecture
Simple to program – X86-64 arch.
8 x double vector unit and FMA

Floating Point (FP)
- 64-bit
- 52-bit

SSE/AVX 128
- AVX-256: 2/4
- MIC-512: 4/8
- xmm: 8/16

Fused Multiply-Add (FMA)

**Example**

\[ A \times B = \text{Product} + C \]

(retain all digits)

- **Key benefits:**
  1. Increased FP compute density
  2. Improved numeric accuracy
  3. Benefits vector and scalar workloads

- **Important notes:**
  - \( A = \pm B \times C \pm D \); IEEE-754r compliant (only round is at the end)
  - Either C or D can come from memory
  - True 4 operand

- **Intel® MIC**
  - Vector size: 512bit
  - Data types:
    - 32 and 64 bit integers
    - 32 and 64bit floats
    - (some support for 16 bits floats)
  - VL: 8,16
  - Sample: 32 bit float
Vector and FMA for M x M

Matrix multiplication

Typical line to compute $A = B \times C + D$

Easy to map to FMA and vector since:

$A_1 = B_1 \times C_1 + D_1$
$A_2 = B_2 \times C_2 + D_2$
$A_3 = B_3 \times C_3 + D_3$

.. $A_8 = B_8 \times C_8 + D_8$

All this in one instruction VFMADDPD!

do $i=$iminloc,imaxloc
  $u_{old}(i,j,in)=u(i,in)+(flux(i-2,in)-flux(i-1,in)) \times dtdx$
end do

Universitetets senter for informasjonsteknologi
Benchmarks – Matmul using MKL

MKL dgemm automatic offload

Two SB processors, One Phi card

Percent offloaded to mic

Universitetets senter for informasjonsteknologi
Benchmark – user fortran code

MxM offloading

Fortran 90 code, double prec.

Performance [Gflops]

Host procs
Co-processor

Memory footprint matrices

2288 MiB 5149 MiB 5859 MiB 6614 MiB
Accelerators hype or production?
Easy to program hard fully exploit

- Same source code – *no changes, same compiler*
- 60 physical cores – *one vector unit per core*
- 240 hardware threads – *at least 120 is needed for fp work*
- 8/16 number wide vector unit - *try to fill it all the time*
- Fused Multiply add instruction – *when can you use this*
- Cache Coherent memory – *nice but has a cost*
- OpenMP – *threads – cc-memory*
- MPI – *uses shared memory communication*
Easy to program - native

- Compile using Intel compilers
  - icc -mmic -openmp
  - ifort -mmic -openmp
  - Other flags are like for Sandy Bridge

- Compile on the host node and launch at MIC node
Easy to program - offload

• Use MKL calls

• Compile using Intel compilers

• Set flags to use offload
  – export MKL_MIC_ENABLE=1
  – export OFFLOAD_DEVICES=0,1
  – export MIC_KMP_AFFINITY=explicit,granularity=fine
Experience - native

Stream Memory bandwidth Benchmark.

Vector update
Experience - native

Xeon Phi vs. single Sandy Bridge

NPB openmp

NASA Parallel Benchmark
Serial, threaded (OpenMP) and MPI versions.
Run natively on Xeon Phi.

Hard to beat Sandy Bridge :(}

Universitetets senter for informasjonsteknologi
Experience – offload user function

MxM offloading
Fortran 90 code, double prec.

![Graph showing performance [Gflops] for different memory footprints (2288 MiB, 5149 MiB, 5859 MiB, 6614 MiB) for Host procs and Co-processor.](image)
Running jobs on the Xeon Phis

- Request both MICs
  - `Qlogin --account xx --partition=accel
    --gres=mic:2 --nodes=1 --ntasks-per-node=16`

- `#SBATCH --partition=accel --gres=mic:2`

- `#SBATCH --nodes=1 --ntasks-per-node=16`
  - Reserve all resources for your job
Running jobs on the Xeon Phis

• This is still an evaluation resource

• Log in to the host, like ssh c19-20 (17,18,19 & 20)
• Log onto one of the Phis, ssh mic0 / mic1
• The software are available at /phi
• A user directory is at /phi/users/<username>
  – Request one to be made for you
• The /phi directory is also mounted at the host