Performance Techniques for Future High-Performance Computers

Artur Podobas
RIKEN R-CCS, Kobe, Japan
Work performed at Matsuoka-lab, Tokyo Institute of Technology
Opinions are my own.
Overall Talk Structure

• Field-Programmable Gate-Arrays in HPC
• MACC: A Transpiler for Multi-GPUs
• Double-Precision FPUs in HPC: an Embarrassment of Riches?
What are FPGAs?

- Field-Programmable Gate-Arrays (FPGAs)
  - Architecture composed of a large number of Look-Up Tables (LUTs)
  - LUTs programmed as ”truth-tables“ and connect to each other
  - Belong to ”fine-grained“ reconfigurable architectures

Figure source: Stratix II ALM-block, Altera (Intel)
What are FPGAs?

- Field-Programmable Gate-Arrays (FPGAs)
  - Architecture composed of a large number of Look-Up Tables (LUTs)
  - LUTs programmed as ”truth-tables” and connect to each other
  - Belong to ”fine-grained” reconfigurable architectures

- Programmed using low-level languages
  - E.g. Verilog or VHDL

```vhdl
... positN_def <= (not(A_POSIT_cycle_1)+'1') when (A_POSIT_cycle_1[32-1] = '1') else A_POSIT_cycle_1;
  posit_shQ_def <= positN_cycle_2[32-2 downto 0] & '0';
  new_inputQ_def <= posit_shQ_cycle_3 when (posit_shQ_cycle_3[32-1]='0') else not (posit_shQ_cycle_3);
  partial_input_1M_def <= new_inputQ_cycle_4[32-1 downto 29];
  partial_0T_def <=
    "11" when (partial_input_1M_cycle_5 = "000") else
    "10" when (partial_input_1M_cycle_5 = "001") else
    "01" when (partial_input_1M_cycle_5 = "010") else
    "01" when (partial_input_1M_cycle_5 = "011") else
    "00" when (partial_input_1M_cycle_5 = "100") else
    "00" when (partial_input_1M_cycle_5 = "101") else
    "00" when (partial_input_1M_cycle_5 = "110") else
    "00";
  partial_input_1L_def <= new_inputQ_cycle_4[29-1 downto 26];
  ...
What are FPGAs?

• Field-Programmable Gate-Arrays (FPGAs)
  • Architecture composed of a large number of Look-Up Tables (LUTs)
  • LUTs programmed as “truth-tables” and connect to each other
  • Belong to “fine-grained” reconfigurable architectures

• Programmed using low-level languages
  • E.g. Verilog or VHDL

• Historically (and still) used for:
  • Military applications
  • Telecommunications
  • Automobile
  • Low-power consumer electronics
  • Simulations
  • High-Performance Computing?
FPGAs in High-Performance Computing

• What changed that encourage looking into FPGAs today?
FPGAs in High-Performance Computing

• What changed that encourage looking into FPGAs today?
  1. Moore’s law is ending
     • Unable to place more functionality/transistors on future chips
     • FPGAs are reconfigurable, possible resilience to end of Moore

HPC Presentation @ KTH
FPGAs in High-Performance Computing

• What changed that encourage looking into FPGAs today?
  1. Moore’s law is ending
     • Unable to place more functionality/transistors on future chips
     • FPGAs are reconfigurable, possible resilience to end of Moore
  2. Maturity in High-Level Synthesis
     • Describe functionality in abstract language
       • C/C++ (LegUp, DWARV, PANDA/BAMBU)
       • OpenCL (Xilinx, Intel)
       • Java (Maxeller)

```
for (int i = 0; i < 100; i++)
  A[i] = B[i] * k;
```
FPGAs in High-Performance Computing

• What changed that encourage looking into FPGAs today?
  1. Moore’s law is ending
     • Unable to place more functionality/transistors on future chips
     • FPGAs are reconfigurable, possible resilience to end of Moore
  2. Maturity in High-Level Synthesis
     • Describe functionality in abstract language
       • C/C++ (LegUp, DWARV, PANDA/BAMBU)
       • OpenCL (Vivado, Intel)
       • Java (Maxeller)
  3. More (floating-point) compute in FPGAs
     • Modern FPGAs has in order of TeraFLOP/s in compute
FPGAs in High-Performance Computing

• We wanted to know the following:
  1. What performance can we get using FPGAs on HPC workloads?
  2. What is the effort involved?
  3. How does it perform compared to CPUs or GPUs?

To this end, we chose *Stencil Computations* and the programming model Intel OpenCL SDK for OpenCL.
Stencil computations

• A very re-occurring computation pattern in High-Performance Computing
  • Weather simulations, Fluid Dynamics, Electrodynamics, etc.
  • Convolutional Neural Networks

• Iterative methods, where each element of a N-dimensional mesh is updated as a weight-sum of its neighbors

• Generally memory-bound (even for high-order stencils)
  • The larger the radius the less memory-bound it becomes
  • Generally high Byte-to-FLOP ratio
Two Gordon Bell prize winners, the Dendrite growth on TSUBAME 2.0 (left, 2012) and the Weather Climate modelling on TaihuLight (right, 2017) are examples of Stencil Computations.
Stencil Computations (cont.)

• After surveying the literature on Stencils on FPGAs, we found the following:
  • Most work target small-radius, 2D stencils
  • All related work enforce strict (and small) dimension constraints
    • E.g. the Mesh had to be at most 128 element wide (with no restrictions on height)
    • There is a loss in generality
• Our objective was to come overcome those limitations:
  • To handle higher dimensional meshes (e.g. 3D)
  • Arbitrary radius on stencils, and
  • Without any loss of generality (and hopefully performance)
The Stencil Accelerator

• We designed a Stencil accelerator:
  • A “front” that reads in data
  • A “end” that writes-back data
  • Custom processing elements serially linked in-between
    • Communicating through on-chip FIFO channels
The Stencil Accelerator: Spatial Blocking

- Neighbor cells are kept on-chip and reused
  - Avoids redundant accesses to external memory
- Stream one dimension and block others
- Blocks are overlapped
  - Avoid halo communication/synchronization
- Parameter: **block size**
  - Controls amount of redundant computation
The Stencil Accelerator: Spatial Blocking

- On-chip buffer is configured as shift register
  - Minimum on-chip memory size: $2 \times \text{rad}$ block rows for 2D and $2 \times \text{rad}$ block planes for 3D
- Computation is vectorized in the $x$ dimension
  - Parameter: vector size
  - Controls spatial parallelism and memory bandwidth utilization
Temporal Blocking

- Multiple time steps (iterations) are combined
  - External memory accesses between them are avoided
  - Scales performance beyond memory bandwidth limit
- Replicated into multiple PEs
  - Each PE works on a consecutive time-step
  - Halo size increases with number of PEs
- Parameter: **degree of temporal parallelism**
  - Equal to number of PEs
Software

• FPGA
  • Quartus and AOC v16.1.2

• GPU
  • Highly-optimized code from [1] (with temporal blocking)
  • CUDA 9.0

• Xeon/Xeon Phi
  • State-of-the-art YASK framework [2] (temporal blocking exists but is ineffective)
  • Intel Compiler 2018.1


## Benchmarks

<table>
<thead>
<tr>
<th>Radius</th>
<th>FLOP per Cell Update</th>
<th>Byte per Cell Update</th>
<th>Byte FLOP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diffusion 2D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>8</td>
<td>0.889</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>8</td>
<td>0.471</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>8</td>
<td>0.320</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
<td>8</td>
<td>0.242</td>
</tr>
<tr>
<td><strong>Diffusion 3D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>8</td>
<td>0.615</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>8</td>
<td>0.320</td>
</tr>
<tr>
<td>3</td>
<td>37</td>
<td>8</td>
<td>0.216</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>8</td>
<td>0.163</td>
</tr>
</tbody>
</table>

- No shared coefficients
- Byte per cell update with assumption of full spatial reuse
## Hardware

<table>
<thead>
<tr>
<th>Type</th>
<th>Device</th>
<th>Peak Compute Performance (GFLOP/s)</th>
<th>Peak Memory Bandwidth (GB/s)</th>
<th>Byte FLOP</th>
<th>TDP (Watt)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Stratix V GX A7</td>
<td>~200</td>
<td>26.5</td>
<td>0.133</td>
<td>40</td>
<td>2011</td>
</tr>
<tr>
<td></td>
<td>Arria 10 GX 1150</td>
<td>1,450*</td>
<td>34.1</td>
<td>0.024</td>
<td>70</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td>Stratix 10 MX 2100</td>
<td>5,940*</td>
<td>512</td>
<td>0.081</td>
<td>150</td>
<td>2018</td>
</tr>
<tr>
<td></td>
<td>Stratix 10 GX 2800</td>
<td>8,640*</td>
<td>76.8</td>
<td>0.008</td>
<td>200</td>
<td>2018</td>
</tr>
<tr>
<td>CPU</td>
<td>Xeon E5-2650 v4</td>
<td>700</td>
<td>76.8</td>
<td>0.110</td>
<td>105</td>
<td>2016</td>
</tr>
<tr>
<td></td>
<td>Xeon Phi 7210F</td>
<td>5,325</td>
<td>400</td>
<td>0.075</td>
<td>235</td>
<td>2016</td>
</tr>
<tr>
<td>GPU</td>
<td>GTX 580</td>
<td>1,580</td>
<td>192.4</td>
<td>0.122</td>
<td>244</td>
<td>2010</td>
</tr>
<tr>
<td></td>
<td>GTX 980Ti</td>
<td>6,900</td>
<td>336.6</td>
<td>0.049</td>
<td>275</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td>Tesla P100 PCI-E</td>
<td>9,300</td>
<td>720.9</td>
<td>0.078</td>
<td>250</td>
<td>2016</td>
</tr>
<tr>
<td></td>
<td>Tesla V100 SMX2</td>
<td>14,900</td>
<td>900.1</td>
<td>0.060</td>
<td>300</td>
<td>2017</td>
</tr>
</tbody>
</table>
First-Order FPGA Results

<table>
<thead>
<tr>
<th>Device</th>
<th>Kernel</th>
<th>bsize</th>
<th>par_time</th>
<th>par_vec</th>
<th>Performance (GFLOP/s)</th>
<th>Logic</th>
<th>M20K</th>
<th>DSP</th>
<th>$f_{max}$ (MHz)</th>
<th>Power (Watt)</th>
<th>Model Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stratix V</td>
<td>Diffusion 2D</td>
<td>4096</td>
<td>24</td>
<td>2</td>
<td>113.068</td>
<td>64%</td>
<td>40%</td>
<td>95%</td>
<td>303.49</td>
<td>27.889</td>
<td>87.1%</td>
</tr>
<tr>
<td></td>
<td>Hotspot 2D</td>
<td>4096</td>
<td>12</td>
<td>4</td>
<td>143.851</td>
<td>95%</td>
<td>53%</td>
<td>83%</td>
<td>231.64</td>
<td>36.103</td>
<td>87.2%</td>
</tr>
<tr>
<td></td>
<td>Diffusion 3D</td>
<td>256x256</td>
<td>4</td>
<td>8</td>
<td>100.921</td>
<td>60%</td>
<td>67%</td>
<td>91%</td>
<td>296.12</td>
<td>29.379</td>
<td>83.7%</td>
</tr>
<tr>
<td></td>
<td>Hotspot 3D</td>
<td>256x256</td>
<td>8</td>
<td>4</td>
<td>102.503</td>
<td>84%</td>
<td>100%</td>
<td>100%</td>
<td>263.08</td>
<td>37.972</td>
<td>79.4%</td>
</tr>
<tr>
<td>Arria 10</td>
<td>Diffusion 2D</td>
<td>4096</td>
<td>36</td>
<td>8</td>
<td><strong>745.487</strong></td>
<td>56%</td>
<td>65%</td>
<td>95%</td>
<td><strong>337.78</strong></td>
<td><strong>65.516</strong></td>
<td><strong>86.4%</strong></td>
</tr>
<tr>
<td></td>
<td>Hotspot 2D</td>
<td>4096</td>
<td>36</td>
<td>4</td>
<td>613.249</td>
<td>46%</td>
<td>86%</td>
<td>95%</td>
<td>333.33</td>
<td>50.349</td>
<td>86.6%</td>
</tr>
<tr>
<td></td>
<td>Diffusion 3D</td>
<td>256x256</td>
<td>12</td>
<td>16</td>
<td><strong>377.614</strong></td>
<td>60%</td>
<td>100%</td>
<td>89%</td>
<td><strong>285.71</strong></td>
<td><strong>64.409</strong></td>
<td><strong>61.4%</strong></td>
</tr>
<tr>
<td></td>
<td>Hotspot 3D</td>
<td>128x128</td>
<td>20</td>
<td>8</td>
<td>329.882</td>
<td>63%</td>
<td>100%</td>
<td>97%</td>
<td>311.11</td>
<td>69.573</td>
<td>62.4%</td>
</tr>
</tbody>
</table>

- 2D 2x faster than 3D
- Arria 10 3-4x faster than Stratix V
- 2D: Big block size $\rightarrow$ low redundancy with temporal blocking $\rightarrow$ par$_{time}$ scales better than par$_{vec}$
- 3D: Small block size $\rightarrow$ high redundancy with temporal blocking $\rightarrow$ par$_{vec}$ scales better than par$_{time}$
First-Order Diffusion 3D Comparison

- Arria 10 faster than K40c/Xeon Phi 7210F and more power efficient than 980 Ti
  - Despite over 8 times lower memory bandwidth
- Stratix 10 MX 2100 and GX 2800 likely faster than P100 and more power efficient than V100
- Temporal blocking has good scaling on FPGAs, limited scaling on GPUs, and no scaling on Xeon/Xeon Phi
Conclusion

- Modern FPGAs can compete with (highly optimized) CPU and GPU versions
  - Reaching more than 700+ GFLOP/s of performance on a Arria10
- The effort involved is however high:
  - Multiple FPGA-specific optimizations
  - Requires decent understanding of low-level hardware to know what to optimized for
  - Compiler does not always do a good job
  - Can be extremely time-consuming
    - Place-and-route failing after 12+ hours
    - Initial reports can be very misleading
  - Performance across versions vary significantly
MACC: A Transpiler for Multi-GPUs

• General Purpose Graphics Processing Units (GPUs)
  • Widespread use in HPC
    • Five of the Top10 HPC systems use GPUs
    • Programming them remains complex
  • Among the prominent options is through programming models
    • Parallelism exposed directly in the program code
    • Compiler-directive based
      • E.g. OpenACC, OpenMP, ...
    • But most models only map to a single GPU?
      • Multi-GPU support not existens?
Motivation

CPU Code

```
cudaMalloc(&dev_m, N * sizeof(float));
cudaMalloc(&dev_p, N * sizeof(float3));
cudaMalloc(&dev_v, N * sizeof(float3));
cudaMemcpy(dev_m, m, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_p, p, N * sizeof(float3), cudaMemcpyHostToDevice);
cudaMemcpy(dev_v, v, N * sizeof(float3), cudaMemcpyHostToDevice);
```

```
for (int t = 0; t < TIME_STEP; t++) {
    kernel1<<<block_num, thread_num>>>(dev_m, dev_p, dev_v);
    kernel2<<<block_num, thread_num>>>(dev_m, dev_p, dev_v);
}
```

```
_cpu64 ideal
cpu64 *m, *p, *v;
for (int i = 0; i < N; i++) {
    m[i] = m[i] + v[i] * DT;
    p[i] = p[i] + v[i] * DT;
    v[i] = v[i] + v[i] * DT;
}
```

GPU Code

```
#pragma acc data copyin (p_x[N], p_y[N], p_z[N], m[N])
#pragma acc data copyout (v_x[N], v_y[N], v_z[N])

for (int t = 0; t < TIME_STEP; t++) {
    #pragma acc parallel loop independent
    for (int i = 0; i < N; i++) {
        p_x[i] += v_x[i] * DT;
        p_y[i] += v_y[i] * DT;
        p_z[i] += v_z[i] * DT;
    }
}
```

```
__global__ void kernel2(float *m, float3 *p, float3 *v){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int offset = tid * THREAD_SIZE;
    for (int j = 0; j < THREAD_SIZE; j++) {
        int i = offset + j;
        p[i].x += v[i].x * DT;
        p[i].y += v[i].y * DT;
        p[i].z += v[i].z * DT;
    }
}
```

HPC Presentation @ KTH
Motivation

Single-GPU Code

```c
#pragma acc data\n  copyout(x[0:N]) present(y)
#pragma acc kernels
for (int i = 0; i < N; i++)
  x[i] = y[i] * y[i];
```

+ Concurrent Execution
+ Data Transfer
+ Loop Division

Multi-GPU Code

```c
numgpuses = acc_get_num_devices(DEVICE_TYPE);
#pragma omp parallel num_threads(numgpuses)
{
  int tnum = omp_get_thread_num();
  int sz = N / numgpuses;
  int lb = sz * tnum; int ub = lb + sz;
  acc_set_device_num(tnum, DEVICE_TYPE);
#pragma acc data copyout(x[lb:sz]) present(y)
#pragma acc kernels
  for (int i = lb; i < ub; i++)
    x[i] = y[i] * y[i];
}
```
MACC

• Our proposal is MACC: A source-to-source transpiler allowing OpenACC codebase to leverage multi-GPU execution
  • Automatic with no manual effort
  • Increase maintenance and portability
  • No invasive changes to the programming model
  • Automatic data management between GPUs
    • With support for GPU-to-GPU transfers through NVLINK
MACC: Overview

1. Makes abbreviated notations flattened
   • #parallel loop → #parallel + #loop, #kernels copy(...) → #data copy(...) + #kernels

2. Replaces #kernels by using #parallel and #loop
   • We employed a basic loop-carried dependency checker

3. Iterative data-flow analysis for runtime detection of array regions
   • Collects array indexes, extracting variable representations

4. Converts #parallel, #data and #update, for each

Single-GPU Code (OpenACC)

Multi-GPU Code (OpenACC w/ OpenMP)

OpenACC + OpenMP Compiler

Multi-GPU Binary

Input

Output

MACC
Example transformations

```c
if(/* sections are changed */) {
    /* recalculate sections */
    #pragma omp parallel num_threads(NUMGPUS)
    {
        int tnum = omp_get_thread_num();
        set_gpu_num(tnum);
        set_data_section(/* ... */);
    }
    #pragma omp barrier
    #pragma acc parallel
    { /* Splitted Loop */ }
}
```

```c
#pragma acc parallel
{ /* ... */ }
```

```c
#pragma acc data\copy(x[0:N])
{ /* ... */ }
```

```c
#pragma omp parallel num_threads(NUMGPUS)
{ 
    copyin_routine(omp_get_thread_num(),x,0,N);
} 
{ /* ... */ }
#pragma omp parallel num_threads(NUMGPUS)
{ 
    copyout_routine(omp_get_thread_num(),x);
} 
```
Multi-GPU execution is enabled when the kernel’s all DEF sections don’t overlap among GPUs

- The switch between single/multi-GPU execution is performed at runtime involving communications

(A) Direct Communications

(B-1) GPU-to-Host Comms removing duplications

(B-2) Host-to-GPU Comms
Performance and Evaluation

• Implemented MACC prototype through XCodeML/C
• Evaluated the performance of several benchmarks
  • Using MACC
  • Using MPI+OpenACC (manual)
  • Using NVIDIA’s Unified Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Intel Xeon E5-2680 V4 (Broadwell-EP 14core) x 2</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>NVIDIA P100 (16GB HBM2@732GB/s) x 4</td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
<td>PGI Compiler 17.10</td>
</tr>
<tr>
<td><strong>CUDA</strong></td>
<td>CUDA 9.0</td>
</tr>
<tr>
<td><strong>NVLink</strong></td>
<td>GPU0 $\leftrightarrow$ GPU2, GPU1 $\leftrightarrow$ GPU3: 40GB/s (one-way)</td>
</tr>
<tr>
<td></td>
<td>Others:</td>
</tr>
<tr>
<td></td>
<td>20GB/s (one-way)</td>
</tr>
</tbody>
</table>

TSUBAME3 (Tokyo Tech)
Himeno Benchmark (19-point Stencil)

Size: \((i, j, k) = (256 \times 256 \times 512)\), Halo Communication (approx. \(255 \times 511 \times 8\) bytes)

- MACC (w/ NVLink) achieved \(3.36\times\) speedup (32.1% performance increase compared to no NVLink)
- Unified Memory, that uses NVLINK, is slightly better than MACC (w/o NVLink)
MACC (w/ NVLink) gained the highest performance (40.9% performance increase compared to no NVLink)

Unified Memory degraded the performance due to memory thrashing (frequent page fault & migration)

MPI version (limited to proc=$n^2$) had low performances due to redundant communications
Conclusion

- We built an OpenACC transpiler to use multi-GPU automatically
  - Not invasive to the source code
  - Communications are generated based on upper/lower-bounds of array accesses
- 3.36× speedup with stencil, and 2.16× speedup with NPB-CG when using four GPUs
- GPU-to-GPU communication via NVLink improved the performances
- Future work:
  - More analysis (affine and non-affine program analysis)
  - Work-sharing optimization (temporality, fine distribution)
  - Combining with task-based system
  - More accelerators, Heterogeneous computing
Double-Precision in Modern FPUs: An Embarrassment of Riches?

• Among the (uncontended) wisdom in HPC is the need for double-precision arithmetic
  • Reflected in TOP500
  • Reflect in modern architectures
  • We wanted to re-evaluated and challenge question that view

• The questions we want to (re-)evaluate is:
  1. How much do HPC workloads actually depend on FP64 instructions?
  2. How well do our HPC workload utilize FP64 instructions?
  3. Are our architectures well- or ill-balanced w.r.t. FP64, FP32, etc.?
  4. Can we empirically evaluate the impact of a different FP64 distribution?
Double-Precision in Modern FPUs: An Embarrassment of Riches?

• Can we find two systems that are architecturally very similar but with a different floating-point compute distributions?
Double-Precision in Modern FPUs: An Embarrassment of Riches?

• Can we find two systems that are architecturally very similar but with a different floating-point compute distributions?
  • Turns out, we can on the Xeon PHI family of systems
• Intel Knight’s Landing and Mill
  • Two many-core architectures
  • Difference in the silicon re-distribution
    • KNM has more single-precision performance
    • KNL has more double-precision performance
• If we truly need the (embarrassingly) large amount of FP64, the this should materialize in a large performance difference between the two architectures

<table>
<thead>
<tr>
<th>Feature</th>
<th>KNL</th>
<th>KNM</th>
<th>Broadwell-EP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>7210F</td>
<td>7295</td>
<td>2x E5-2650v4</td>
</tr>
<tr>
<td># {Cores} (HT)</td>
<td>64 (4x)</td>
<td>72 (4x)</td>
<td>24 (2x)</td>
</tr>
<tr>
<td>Base Frequency</td>
<td>1.3 GHz</td>
<td>1.5 GHz</td>
<td>2.2 GHz</td>
</tr>
<tr>
<td>Max Turbo Freq.</td>
<td>1.4 GHz</td>
<td>1.6 GHz</td>
<td>2.9 GHz</td>
</tr>
<tr>
<td>CPU Mode</td>
<td>Quadrant</td>
<td>Quadrant</td>
<td>N/A</td>
</tr>
<tr>
<td>TDP</td>
<td>230 W</td>
<td>320 W</td>
<td>210 W</td>
</tr>
<tr>
<td>DRAM Size</td>
<td>96 GiB</td>
<td>96 GiB</td>
<td>256 GiB</td>
</tr>
<tr>
<td>DDR3 BW</td>
<td>71 GB/s</td>
<td>88 GB/s</td>
<td>122 GB/s</td>
</tr>
<tr>
<td>MCDRAM Size</td>
<td>16 GiB</td>
<td>16 GiB</td>
<td>N/A</td>
</tr>
<tr>
<td>DDR4 BW</td>
<td>439 GB/s</td>
<td>430 GB/s</td>
<td>N/A</td>
</tr>
<tr>
<td>MCDRAM Mode</td>
<td>Cache</td>
<td>Cache</td>
<td>N/A</td>
</tr>
<tr>
<td>LLC Size</td>
<td>32 MiB</td>
<td>36 MiB</td>
<td>60 MiB</td>
</tr>
<tr>
<td>Inst. Set Extension</td>
<td>AVX-512</td>
<td>AVX-512</td>
<td>AVX2</td>
</tr>
<tr>
<td>FP32 Peak Perf.</td>
<td>5,324 Gflop/s</td>
<td>13,824 Gflop/s</td>
<td>1,382 Gflop/s</td>
</tr>
<tr>
<td>FP64 Peak Perf.</td>
<td>2,662 Gflop/s</td>
<td>1,728 Gflop/s</td>
<td>691 Gflop/s</td>
</tr>
</tbody>
</table>
Double-Precision in Modern FPUs: An Embarrassment of Riches?

• We benchmarked and evaluated both architectures:
  • 19 mini-Applications from two well-known suites
    1. ECP Proxy Applications (used in procuring CORAL machine)
    2. Post-K MiniApps (used in procuring Post-K)
    3. HPL and HPCG for sanity testing
  • Performance analysis tools
    1. GNU Perf
    2. Intel SDE
    3. Intel PCM
    4. Intel Vtune
    5. Valgrind/Heap-track
  • Each applications’ parameter optimized by hand
    1. OpenMP threads vs MPI ranks
    2. All fit inside MCDRAM
  • Several month long process, which also served as great introduction to HPC benchmarking to students
Double-Precision in Modern FPUs: An Embarrassment of Riches?
Double-Precision in Modern FPUs: An Embarrassment of Riches?
Conclusion

• Performance between KNM and KNL is not as big as expected
  • The large difference in peak DP not materialized in performance
• Might not need the excessive amount of DP performance
  • Silicon better spent somewhere else
    • Increased b/w
    • Larger caches
    • Mixed- or hybrid-precision
• Exciting to evaluate upcoming architectures
  • E.g. ARM A64FX

Systematic and Open-source Framework for benchmarking
• https://gitlab.com/domke/PAsudy
Summary

• Summarized three fronts of HPC computing
  • FPGA and reconfigurable compute accelerators
  • Compilers for HPC infrastructure
  • Benchmark and (Re-)Evaluation of HPC applications

• Several new challenges and opportunities with the end of Moore’s law
  • Coarse-Grained Reconfigurable Architectures / Overlay architectures
  • Neuromorphic architectures
  • (Quantum?)

• Really exciting times to be in the field
Publications related to the presentation

2. High-Performance High-Order Stencil Computation on FPGAs Using OpenCL (IPDPS RAW 2018)
3. Accelerating Posit-based Computations using FPGAs and OpenCL (CONGA 2018, Invited Talk)
   (URL: https://www.youtube.com/watch?v=j8JNiWMAaU0&t=8s)
4. MACC: An OpenACC Transpiler for Automatic Multi-GPU Use (SCAsia 2018)
5. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL (FPGA 2018)
Acknowledgements

- **Colleagues**: Hamid Reza Zohouri, Jens Domke, Kazuaki Matsumura, Mohamed Wahib, Haoyu Zhang, Keita Yashima, Toshiki Tsuchikawa, Yohei Tsuji, Naoya Maruyama, Satoshi Matsuoka (+ many more)

- **Funding**: JSPS Postdoc fellowship, JSPS KAKENHI, JST CREST-AI, JST CREST BIGDATA
• All types of general-purpose processor legacy-software optimizations for HPC,
• Changes to (collective) communication algorithms or implementations to enable the use of different numerical methods (for example: Lagrangian vs. Eulerian),
• Accelerating of pre-/post-processing in a scientific workflows or axillary tools used in HPC environments,
• Improved maintainability and performance through the use of existing production libraries,
• Revisiting and applying modern compiler (flag) techniques, performance analysis tools, moderate usage of OpenMP pragmas, etc., for performance gains,
• Manual code refactoring, such as loop transformations or changing data structures, to acknowledge the shifting ratio in memory vs. compute capabilities of modern architectures, and
• Using mixed or adaptive precision wherever possible.

Final submission Deadline: April 18, 2019
https://refac-ws.gitlab.io/2019/
• The first RIKEN workshop on neuromorphic computing and applications:
  • Several excellent talks
    • 3 keynote speakers
    • 12 invited talks from academia and industry
  • A full day tutorial on Loihi

March 11, 2019 (Mon) – March 13, 2019 (Wed)
https://usability-research.r-ccs.riken.jp/r-wonc19/