Design of Runtime System for Task-based FPGA Programming

Programming Environment Research Team
Jinpil Lee
Agenda

- Task offloading to FPGA using task
- Batch generation in runtime
Background

- Writing computing kernels for FPGAs
  - OpenCL/OpenMP/OpenACC/DPC++/DSL
- How to control the generated kernel?
  - OpenCL event
  - OpenACC kernel
  - OpenMP target
    generates FPGA kernels
- XcalableMP task

assumption
1. optimized libraries using FPGAs
   (e.g. BLAS for FPGA)
2. APIs are called by CPUs
3. inner-node parallelism
   (inter-node work distribution is the future work)
Motivated Example

- In some OpenMP task program, tasks like yellow part will appear
  - dominant computation time
  - can be executed in parallel by OpenMP task
- Host can offload these tasks asynchronously
  - overlap between CPU and FPGA

```c
#pragma omp task depend ...
{
  ...
  calc_on_fpga(event);
  do {
    #pragma omp taskyield
    status = checkStatus(event);
  } while (status != done);
  ...
}
```

```c
#pragma xmp task depend async on dev(FPGA) ...
{
  calc(event);
}
```

```c
#pragma xmp variant(FPGA:calc)
{
  calc_on_fpga(event);
}
```
Implementation

- Prepared independent queue for each FPGA instance
  - parallel execution
  - cl_command_queue... like CUDA stream object
- Asynchronous task execution
  - Argobots

### Hardware configuration

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2660 v4 x 2</td>
</tr>
<tr>
<td>Host DRAM</td>
<td>DDR4-2400 16GB x 4</td>
</tr>
<tr>
<td>FPGA Board</td>
<td>BittWare A10PL4 (Intel Arria10 GX1150) PCIe Gen3 x8</td>
</tr>
<tr>
<td>FPGA DRAM</td>
<td>DDR4-2133 4GB x 2</td>
</tr>
</tbody>
</table>

### Software configuration

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>CentOS 7.3 x64</td>
</tr>
<tr>
<td>Host Compiler</td>
<td>Intel C Compiler 18.1</td>
</tr>
<tr>
<td>Thread library</td>
<td>Argobots 1.0b1</td>
</tr>
<tr>
<td>FPGA compiler</td>
<td>Intel FPGA SDK for OpenCL 17.12.304</td>
</tr>
</tbody>
</table>
SIMD vs Instance

<table>
<thead>
<tr>
<th>SIMD width</th>
<th>num. of instances</th>
<th>Fmax (MHz)</th>
<th>DSPs (%)</th>
<th>BRAM (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S16</td>
<td>16</td>
<td>164.71(1.0)</td>
<td>69</td>
<td>60</td>
</tr>
<tr>
<td>S8</td>
<td>8</td>
<td>179.59(1.09)</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>S4</td>
<td>4</td>
<td>213.94(1.29)</td>
<td>70</td>
<td>90</td>
</tr>
</tbody>
</table>

Lower is Better

Higher is Better

![Graph 1](graph1.png)

![Graph 2](graph2.png)
Parallelism on FPGAs

Multiple Instances with
- Independent task queue
  - like multi-core CPUs
  - can use XMP/OMP task

- Single task queue
  - like GPUs
  - internal hardware scheduler
  - single task will use the whole device
  - requires parallelism inside a task
Batched Kernel API

- batch: gathering multiple calculation kernels into a single task
- implementation depends on the target hardware
  - CPU/GPU implementation exploits its parallelism
  - assign kernels onto computing cores
- needs code modification

**BLAS API**

```c
void calc_dgemms(const int num_kernels,
                 const int n, double **A, double **B,
                 double **C, double *DMONE,
                 double *DONE) {
    for (int i = 0; i < num_kernels; i++) {
        cublasDgemm(handle, CUBLAS_OP_N,
                    CUBLAS_OP_N, n, n, n,
                    &DMONE, A[i], n, B[i], n,
                    &DONE, C[i], n);
        cudaDeviceSynchronize();
    }
}
```

**Batched BLAS API**

```c
void calc_dgemms_batched(const int num_kernels,
                         const int n, double **A, double **B, double **C,
                         double *DMONE, double *DONE) {
    cublasDgemmBatched(handle, CUBLAS_OP_N,
                       CUBLAS_OP_N, n, n, n, &DMONE,
                       A, n, B, n,
                       &DONE, C, n, num_kernels);
    cudaDeviceSynchronize();
}
```
**Code Translation from OMP**

- static vs dynamic
  - static
    - no runtime overhead
    - covers easy cases
  - dynamic
    - simple
    - runtime overhead
    - irregular patterns

```c
for (int i = k+1; i < num_tiles; i++) {
    for (int j = k+1; j < i; j++) {
#pragma omp task depend(in:A[k][i],A[k][j]) depend(out:A[i][i])
    dgemm(A[k][i], A[k][j], A[j][i], tile_size, tile_size);
    }
#pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
    dsyrk(A[k][i], A[i][i], tile_size, tile_size);
}
```

```c
for (int i = k+1; i < num_tiles; i++) {
    #pragma omp task depend(in:A[k][i],A[k][K+1:i]) depend(out:A[K+1:i][i])
    dgemm_batch(A_ptr_array, ...);
    #pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
    dsyrk(A[k][i], A[i][i], tile_size, tile_size);
}
void cholesky_decomposition(double* A[nt][nt], ...) {
    for (int k = 0; k < nt; k++) {
        void *args[3] = {A[k][k], &tile_size, &tile_size};
        void *out_data[1] = {A[k][k]};
        task_create(0, potrf, 3, args, 0, NULL, 1, out_data);
        for (int i = k + 1; i < nt; i++) {
            void *args[4] = {A[k][k], A[k][i], &tile_size, &tile_size};
            void *in_data[1] = {A[k][k]};
            void *out_data[1] = {A[k][i]};
            task_create(0, trsm, 4, args, 1, in_data, 1, out_data);
        }
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
                void *args[5] = {A[k][i], A[k][j], A[j][i], &tile_size, &tile_size};
                void *in_data[2] = {A[k][i], A[k][j]};
                void *out_data[1] = {A[j][i]};
                task_create(1, gemm, 5, args, 2, in_data, 1, out_data);
            }
        }
    }
}
Task Runtime

- non-batch tasks use normal OpenMP runtime
- batch task
  - task with batch ID
  - put into batch queue
  - merged when the same kind of task exists
Task Runtime (cont'd)

- task scheduling algorithm uses the task queue
  - (batch) task is not ready for task scheduling when in the batch queue
- batch tasks are moved to the task queue
  - when the task queue is empty (ready for scheduling)
  - delaying scheduling to gather more kernels into a batch

![Diagram of task runtime]

- Normal task
- Batch task
- Task queue
- Batch queue
Evaluation

- DGEMM Loop simple, shows the ideal performance
- Blocked Cholesky Decomposition to see the runtime overhead
- single thread, all serialized

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel (R) Xeon (R) CPU E5-2680 8 cores (2 sockets), 2.70 GHz</td>
</tr>
<tr>
<td></td>
<td>Intel Compiler 18.0.3</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla K20</td>
</tr>
<tr>
<td></td>
<td>CUDA 9.1 with cuBLAS</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4 64 GB</td>
</tr>
</tbody>
</table>
Result: Cholesky Decomposition

- block (tile) size: 128, # Blocks: 4096 $\rightarrow$ 36% perf improvement
- little improvement with small matrices
- not consistent with the previous result ...
Performance Breakdown

- system: overall runtime overhead (batch creation, scheduling)
- huge performance improvement & overhead with small matrices
- trade off between batch efficiency and task overhead?

Performance Ratio

<table>
<thead>
<tr>
<th></th>
<th>ser 4096-32</th>
<th>batch 4096-32</th>
<th>ser 4096-64</th>
<th>batch 4096-64</th>
<th>ser 4096-128</th>
<th>batch 4096-128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Ratio</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>math</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dgemm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dgemm batch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Future Plan

- Work distribution among FPGA nodes
  - data distribution
  - scheduling tasks among nodes by dependency

- Remote Procedure Call from Fugaku to ESSPER
  - by XcalableMP?