SIMD Parallel Execution on GPU from High-Level Dataflow Synthesis

Aurelien Bloch,
Simone Casale-Brunet,
Marco Mattavelli
EPFL

École Polytechnique Fédérale de Lausanne
Outline

- Introduction
- RVC-CAL model of computation
- Design methodology
  - Improved communications
  - Improved GPU partitions
  - SIMD parallelization optimization
- Experimental evaluation
- Conclusion and Future Work
Introduction

- Moore’s law is slowing down
- Explosion of multicore systems
- Scale-out by an increase in processor count is not enough anymore
- The trend is to go heterogeneous
- Heterogeneous systems are difficult to program
  - Analyze
  - Optimize
  - Portable

- Contributions and objectives
  - Computer-assisted software design for heterogeneous platforms
  - Increase resource utilization and parallelism opportunities
RVC-CAL model of computation

Actors:
- Computational kernel
- Communication via FIFO buffers
- Atomic execution of actions
- Encapsulate states variables

Actions:
- Consume/Produce tokens
- Can modify internal variables

(a) An example with five actors (i.e. Prod, CopyTokenA, CopyTokenB, PingPong and Merger).

```plaintext
actor Producer () -> int 0:
  uint counter := 0;
  p: action -> 0!counter;
  guard counter < &
  do  counter := counter + 1; end

actor CopyTokens (String name) int I -> int 0:
  o: action I!val => 0!val; end

actor PingPong () int I => int 0:
  p1: action I!val => 0!val;
  do println("PingPong(p1):" + val); end
  p2: action I!val => 0!val;
  do println("PingPong(p2):" + val); end
  schedule fam a_ppl:
    a_ppl(p1) => a_ppl;
    a_ppl(p2) => a_ppl;
  end

actor Merger () int I1, int I2 => : 
  uint counter := 0;
  m: action I1: v1, I2: v2 => 
  do println("Merger("counter")": v1 + ";" + v2);
  counter := counter + 1;
  end
```

(b) Producer.cal
(c) CopyTokens.cal
(d) PingPong.cal
(e) Merger.cal
RVC-CAL model of computation

- Direct code generation for software or hardware
- Explicit memory contention management
  - No data race
- Profiling capabilities
Design methodology - Tool flow

- CopyTokenA
- CopyTokenB
- Prod
- PingPong
- Merger
- CAL
- XCF
- BXDF
- ORCC
- Exelixi CUDA backend
- C++ CUDA
- NVCC
- BIN
Design methodology - Tool flow

MCSoC 2021 / SIMD Parallel Execution on GPU from High-Level Dataflow Synthesis

Aurelien Bloch

CAL
XCF
BXDF

ORCC
Exelixi CUDA backend

C++
CUDA

NVCC

BIN
New CPU/GPU FIFO
- pinned memory
- hardware accesses
- less overhead
Action selection and actions on GPU

Independent and long running kernel

Automatic ending detection

```c
__global__ void idct2dNS::
  action_selection(Idct2d* idct2d,
           EStatus* status,
           size_t actorIdx,
           size_t actorSize) {
  uint64_t _start = clock64();
  bool stop = false;
  do {
    status[actorIdx] = None;
    bool res1 = true;
    while (res1) {
      res1 = false;
      idct2d->status_IN = idct2d->IN->count(0);
      idct2d->status_OUT = idct2d->OUT->rooms();
      bool res2 = true;
      while (res2) {
        res2 = false;
        if(idct2d->status_IN >= 64*512 66
idct2d->isSchedulable_action()) {
          if(idct2d->status_OUT >= 64*512) {
            Ports ports;
            ports.IN =idct2d->IN->read_address(0,64*512);
            ports.OUT =idct2d->OUT->write_address();
            idct2dNS::action<<1,512,0>>(idct2d,ports);
            cudaDeviceSynchronize();
            res1 = true;
            res2 = true;
            status[actorIdx] = hasExecuted;
          }
        }
      }
    }
  }
  if (checkStatus(status, actorSize) == None) {
    stop = (clock64() - _start) > wait_period;
  } else {
    _start = clock64();
  }
  while(!stop);
}
```
Design methodology - GPU partitions

- Action selection and actions on GPU
- Independent and long running kernel
- Automatic ending detection

```c
__global__ void idct2dNS:
    action_selection(Idct2d* idct2d,
                     EStatus* status,
                     size_t actorIdx,
                     size_t actorSize) {

    uint64_t _start = clock64();
    bool stop = false;
    do {
        status[actorIdx] = None;
        bool res1 = true;
        while (res1) {
            res1 = false;
            idct2d->status_IN = idct2d->IN->count(0);
            idct2d->status_OUT = idct2d->OUT->rooms();
            bool res2 = true;
            while (res2) {
                res2 = false;
                if(idct2d->status_IN >= 64*512 &&
                   idct2d->isSchedulable_action()) {
                    if(idct2d->status_OUT >= 64*512) {
                        Ports ports;
                        ports.IN = idct2d->IN->read_address(0, 64*512);
                        ports.OUT = idct2d->OUT->write_address();
                        idct2dNS:action<<1, 512, 0>>(idct2d, ports);
                        cudaDeviceSynchronize();
                        res1 = true;
                        res2 = true;
                        status[actorIdx] = hasExecuted;
                    }
                }
            }
        }
    }
    if (checkStatus(status, actorSize) == None) {
        stop = (clock64() - _start) > wait_period;
    } else {
        _start = clock64();
    }
    } while(!stop);
```


- **Action selection and actions on GPU**
- **Independent and long running kernel**
- **Automatic ending detection**

```c
__global__ void idct2dNS::
action_selection(Idct2d* idct2d,
EStatus* status,
size_t actorIdx,
size_t actorSize) {

uint64_t _start = clock64();
bool stop = false;
do {
    status[actorIdx] = None;
    bool res1 = true;
    while (res1) {
        res1 = false;
        idct2d->status_IN = idct2d->IN->count(0);
        idct2d->status_OUT = idct2d->OUT->rooms();
        bool res2 = true;
        while (res2) {
            res2 = false;
            if(idct2d->status_IN >= 64*512 &&
               idct2d->isSchedulable_action()) {
                if(idct2d->status_OUT >= 64*512) {
                    Ports ports;
                    ports.IN =idct2d->IN->read_address(0,64*512);
                    ports.OUT =idct2d->OUT->write_address();
                    idct2dNS::action<<1,512,0>>(idct2d,ports);
                    cudaDeviceSynchronize();
                    res1 = true;
                    res2 = true;
                    status[actorIdx] = hasExecuted;
                }
            }
        }
        if (!CheckStatus(status, actorSize) == None) {
            stop = (clock64() - _start) > wait_period;
        } else {
            _start = clock64();
        }
    }
} while(!stop);
```
Design methodology - SIMD parallelization

- Dynamic parallelism for SIMD parallelization

```c
__global__ void idct2dNS::
action_selection(Idct2d* idct2d,
EStatus* status,
size_t actorIdx,
size_t actorSize) {
uint64_t _start = clock64();
bool stop = false;
do {
    status[actorIdx] = None;
    bool res1 = true;
    while (res1) {
        res1 = false;
        idct2d->status_IN = idct2d->IN->count(0);
        idct2d->status_OUT = idct2d->OUT->rooms();
        bool res2 = true;
        while (res2) {
            res2 = false;
            if(idct2d->status_IN >= 64*512 &&
              idct2d->isSchedulable_action()) {
                if(idct2d->status_OUT >= 64*512) {
                    Ports ports;
                    ports.IN =idct2d->IN->read_address(0,64*512);
                    ports.OUT =idct2d->OUT->write_address();
                    idct2dNS::action<<1,512,0>>(idct2d,ports);
                    cudaDeviceSynchronize();
                    res1 = true;
                    res2 = true;
                    status[actorIdx] = hasExecuted;
                }
            }
        } }
    if (checkStatus(status, actorSize) == None) {
        stop = (clock64() - _start) > wait_period;
    } else {
        _start = clock64();
    }
} while(!stop);
```

Aurelien Bloch
Dynamic parallelism for SIMD parallelization

```c
__global__ void idct2dNS::
    action_selection(Idct2d* idct2d,
                     EStatus* status,
                     size_t actorIdx,
                     size_t actorSize) {

    uint64_t _start = clock64();
    bool stop = false;
    do {
        status[actorIdx] = None;
        bool res1 = true;
        while (res1) {
            res1 = false;
            idct2d->status_IN = idct2d->IN->count(0);
            idct2d->status_OUT = idct2d->OUT->rooms();
            bool res2 = true;
            while (res2) {
                res2 = false;
                if (idct2d->status_IN >= 64*512 && idct2d->isSchedulable_action()) {
                    if (idct2d->status_OUT >= 64*512) {
                        Ports ports;
                        ports.IN = idct2d->IN->read_address(0, 64*512);
                        ports.OUT = idct2d->OUT->write_address();
                        idct2dNS::action<<1, 512, 0>>(idct2d, ports);
                        cudaDeviceSynchronize();
                        res1 = true;
                        res2 = true;
                        status[actorIdx] = hasExecuted;
                    }
                }
            }
        }
    } while (!stop);
}
```
Dynamic parallelism for SIMD parallelization

```
__global__ void idct2dNS::
    action_selection(Idct2d* idct2d,
        EStatus* status,
        size_t actorIdx,
        size_t actorSize) {

    uint64_t _start = clock64();
    bool stop = false;
    do {
        status[actorIdx] = None;
        bool res1 = true;
        while (res1) {
            res1 = false;
            idct2d->status_IN = idct2d->IN->count(0);
            idct2d->status_OUT = idct2d->OUT->rooms();
            bool res2 = true;
            while (res2) {
                res2 = false;
                if(idct2d->status_IN >= 64*512 &&
                    idct2d->isSchedulable_action()) {
                    if(idct2d->status_OUT >= 64*512) {
                        Ports ports;
                        ports.IN = idct2d->IN->read_address(0, 64*512);
                        ports.OUT = idct2d->OUT->write_address();
                        idct2dNS::action<<1,512,0>>(idct2d, ports);
                        cudaDeviceSynchronize();
                        res1 = true;
                        res2 = true;
                        status[actorIdx] = hasExecuted;
                    } else {
                        _start = clock64();
                    }
                }
            }
        }
    } while(!stop);
```
Experimental evaluation - Setup

- Nvidia GeForce GTX 1660 SUPER GPU
  - 6 GB of memory
- Intel Skylake i5-6600 CPU
  - 16 GB of DDR4 RAM
- CUDA 11.3.1
Experimental evaluation - SIMD parallelization

<table>
<thead>
<tr>
<th>Speedup</th>
<th>CPU</th>
<th>Parallel GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Experimental evaluation - SIMD parallelization

<table>
<thead>
<tr>
<th></th>
<th>Seq GPU</th>
<th>Parallel GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame rate [image/sec]</td>
<td>0.24</td>
<td>2.32</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>9.67</td>
</tr>
</tbody>
</table>
Experimental evaluation - CPU/GPU communications

Aurelien Bloch
Experimental evaluation - CPU/GPU communications

Speedup

<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
<th>m5</th>
<th>m6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Aurelien Bloch
Conclusion:

- New FIFO for fast communications between the CPU/GPU partitions
- New GPU partitions to reduce CPU utilization
- Methodology for using Cuda's *dynamic parallelism* to implement intra-kernel parallel execution of actions.
- The results show the related performance improvements

Future Work:

- Automatic classification and flagging of actors and actions
- Runtime for automatic action parallelization