Opportunities for deferring application partitioning and accelerator synthesis to runtime – EU FP7 Project SAVE –

Tobias Kenter, Gavin Vaz, Heinrich Riebler, Christian Plessl

High-Performance IT Systems Group
Paderborn Center for Parallel Computing (PC²)
Paderborn University

kenter@uni-paderborn.de
Motivation

- Heterogeneous systems, from data center to personal devices

Intel Begins Shipping Xeon Chips With FPGA Accelerators

By Jeffrey Burt | Posted 2016-04-13

Intel has begun shipping a development module that features the company’s latest Xeon E5-2600 v4 “Broadwell” processors and programmable chips that will help customers drive performance while holding down power consumption.

The multichip platform is pairing the 14-nanometer Xeon E5-2600 v4 “Broadwell” processors—launched late in March—with the Arria10 field-programmable gate arrays (FPGAs) from Altera. Diane Bryant, senior vice president and general manager of Intel’s Data Center Group, on April 13 announced during a keynote address at the company’s Intel Developer Forum (IDF) in China that the modules were shipping.

Intel and other chip makers are increasingly relying on accelerators to help improve the performance and energy efficiency of their processors and speed up the workloads that run on them. Nvidia and Advanced Micro Devices offer GPU accelerators. However, the company also is now using FPGAs, which can be reprogrammed through software after they’ve been manufactured. They’re becoming more important for cloud and Web-scale environments, where workloads can change quickly.

Intel for several years had partnered with Altera to take advantage of the company’s technology. Intel now has brought the company in-house, earlier this year completing the acquisition of Altera, which the chip maker bought for $16.7 billion. Intel officials have said the goal is to eventually integrate the FPGAs onto the same die as the CPU.

Intel is not only relying on FPGAs for CPU acceleration. The company also has a growing portfolio of Xeon Phi chips, which are x86-based co-processors that can act either as accelerators or as primary chips.

In addition, Intel is partnering with eASIC to bring application specific integrated circuits (ASICs) to custom Xeons to be used in enterprise data centers and cloud environments for such workloads as data analytics and security. The goal is to offer customers a broad range of choices.

- Most software is not heterogeneous
- Code extraction, synthesis and offloading at runtime!
1 for(int x=0; x<WIDTH; x+=VL_max)
2   VL = min(VL_max, WIDTH-x)
3 for(int y=1; y<HEIGHT; y++)
4   im[x:x+VL][y] = im[x:x+VL][y] + im[x:x+VL][y-1];
Overlay Target 1: Configurable DFE

- Platform: Maxeler MAX3424A Vectis FPGA board
- Library of overlays with fixed structure and configurable nodes

LLVM IR

match control flow
match statements
## Overlay Target 1: Results

- Speedups depend on filter size and input size

<table>
<thead>
<tr>
<th>Filter size</th>
<th>Image resolution</th>
<th>576</th>
<th>720</th>
<th>1080</th>
<th>1080</th>
<th>2160</th>
<th>4320</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 × 3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5 × 5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.85</td>
</tr>
<tr>
<td>7 × 7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.55</td>
<td>3.23</td>
</tr>
<tr>
<td>9 × 9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.5</td>
<td>5.25</td>
</tr>
<tr>
<td>11 × 11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.03</td>
<td>3.66</td>
<td>7.639</td>
</tr>
<tr>
<td>13 × 13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.37</td>
<td>1.4</td>
<td>4.9</td>
<td>10.08</td>
</tr>
</tbody>
</table>

Overlay Target 2: Vector Code

- Platform: Convey HC-1, four user FPGAs
- Instruction-programmable overlay: vector processor
  - Loop extraction and vectorization

```c
//Integral column sums
for (int x=0; x<WIDTH; x++)
  for (int y=1; y<HEIGHT; y++)
    im[x][y] = im[x][y] + im[x][y-1];
```

```c
1  for(int x=0; x<WIDTH; x+=VL_max)
2     VL = min(VL_max, WIDTH-x)
3  for(int y=1; y<HEIGHT; y++)
4     im[x:x+VL][y] = im[x:x+VL][y] + im[x:x+VL][y-1];
```

[Kenter, Vaz, Plessl, *Partitioning and Vectorizing Binary Applications for a Reconfigurable Vector Computer*, ARC ‘14]
Overlay Target 2: Results

- Speedups depend on input size and arithmetic intensity

![Graph showing speedups for different resolutions and algorithms.](image)
Offloading decisions

• Static offloading decisions?
  – Profiling with **representative profiling** data
  – Trip counts depending on **constant** bounds

• What else?
  – Iteration space often deterministic at **loop entry**

```c
loadImage(&width, &height);
...
for(int x=0; x<width; x++)
  for(int y=1; y<height; y++)
    im[x][y] = im[x][y] + im[x][y-1];
```

//Integral column sums
for(int x=0; x<WIDTH; x++)
  for(int y=1; y<HEIGHT; y++)
  im[x][y] = im[x][y] + im[x][y-1];
Offloading decisions in application code

Compile Time

Application in LLVM IR

Compilation Toolflow

Partition Pass

Runtime Code Injection

LLVM (CPU) Backend

Convey Compiler

Convey Assembler + Linker

HeterogeneousExecutable

CPU Code

Coprocessor Code

Run Time

Heterogeneous Executable

Execution on CPU

offloading decision

run on CPU

run on coprocessor

Migrate required data back to CPU memory

Migrate required data to coprocessor memory

Cohesive shared virtual memory

Host

Coprocessor
Offloading decisions

- Trip counts depending on **constant** bounds

```c
//Integral column sums
for(int x=0; x<WIDTH; x++)
    for(int y=1; y<HEIGHT; y++)
        im[x][y] = im[x][y] + im[x][y-1];
```

- Iteration space often deterministic at **loop entry**

```c
loadImage(&width, &height);
...
for(int x=0; x<WIDTH; x++)
    for(int y=1; y<HEIGHT; y++)
        im[x][y] = im[x][y] + im[x][y-1];
```

- Iteration space can be **unknown** at loop entry (e.g. data driven)

```c
do
    px = read(x);
    x++;
while (px != eof)
```

- **How frequent are these patterns?**

Tobias Kenter, Paderborn University
Potential of Decisions at Runtime 1

- Three benchmarks
  - SPEC CPU 2006, MiBench, MediaBench I
- \(~16,000\) loops executed at least once
- Classified into three groups
  - RT: deterministic trip count at loop entry
  - CT: deterministic trip count at compile-time
  - NC: typically data-driven loop exit

Potential of Decisions at Runtime 2

- For offloading, loops need high trip counts
- ~ 92 billion loop iterations counted

<table>
<thead>
<tr>
<th>SPEC CPU 2006</th>
<th>MiBench</th>
<th>MediaBench I</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Bar chart" /></td>
<td><img src="image.png" alt="Bar chart" /></td>
<td><img src="image.png" alt="Bar chart" /></td>
<td><img src="image.png" alt="Bar chart" /></td>
</tr>
</tbody>
</table>

Loop classification

Loop execution frequency

---

Tobias Kenter, Paderborn University
Breakdown by applications
Decisions on system level

**OBSERVE**
- application
- heartbeat monitor
- resource utilization
- monitoring infrastructure

**DECIDE**
- policies

**ACT**
- actions
  - core assignment
  - DVFS
  - priority modification

- actuating elements

Monitoring infrastructure observed data

Requirements & goals (system and apps)

Tobias Kenter, Paderborn University
Summary + outlook

• Summary
  – Accelerator synthesis
    ▪ Overlay architectures for fast synthesis and reconfiguration
  – Runtime decisions
    ▪ Deterministic decisions at loop entries often possible

• Outlook
  – Machine learning for decisions
  – Impact of system level decisions
Thank you!

Questions?
Potential of Decisions at Runtime 2

- For offloading, loops need high trip counts
- ~ 92 billion loop iterations counted

- Significant potential for decisions at runtime

[Vaz, Riebler, Kenter, Plessl, Potential and Methods for Embedding Dynamic Offloading Decisions into Application Code, Computers & Electrical Engineering '16]
Inserting the runtime decision

fun_A

Block_0
Block_1
Block_2
Block_3
Block_4
Block_5
Block_6
Block_7
Block_8
Block_9

fun_A

Block_0
Runtime Check
Block_0
Block_1
Block_2
Block_3
Block_4
Block_5
Block_6
Block_7
Block_8
Block_9

fun_B

Migrate data to the coprocessor

Offloading Decision

Yes

No

Migrate data back to the CPU

Host Function

Coprocessor Function

Gavin Vaz, University of Paderborn
Offloading Decision at Runtime

Tobias Kenter, Paderborn University
Speedup compared to always on coprocessor

Beneficial to execute on the CPU as offloading overheads are high

Speedups of \textbf{up to 10x} can be achieved depending on the loop iteration space.

Tobias Kenter, Paderborn University
Speedup compared to always CPU

Beneficial to execute on the coprocessor as offloading overheads can be amortized

Speedups of up to 3x can be achieved by executing code on the coprocessor instead of the CPU
Deferring Accelerator Offloading Decisions to Application Runtime

Gavin Vaz, Heinrich Riebler, Tobias Kenter, Christian Plessl

Custom Computing Group
Computer Engineering Group
University of Paderborn

{gavin.vaz - heinrich.riebler - kenter - christian.plessl}@uni-paderborn.de
Agenda

- How are hotspots offloaded?
- What we investigate
- Inserting the runtime decisions
- Case Study and Evaluation
- Conclusion & Outlook
How are hotspots offloaded?

1. Identify suitable hotspots
   - What can be run on the coprocessor?
   - What are computational intensive parts of an application?
   - Can we benefit from offloading them to the coprocessor?

2. Generate coprocessor code

3. Modify the host application to call the offloaded code
Identify suitable hotspots

Can be executed on the coprocessor

Potential hotspots

Blocks to offload

Tobias Kenter, Paderborn University
Generate coprocessor code

Host Function

Coprocessor Function

fun_A

fun_A

fun_B

Block_0

Block_1

Block_2

Block_3

Block_4

Block_5

Block_6

Block_7

Block_8

Block_9

The image part with relationship ID rId5 was not found in the file.
Identify suitable hotspots

Can be executed on the coprocessor

Potential hotspots

Blocks to offload
Identify suitable hotspots

Can be executed on the coprocessor

Potential hotspots

Blocks to offload

const int WIDTH = 1024;
void heavyFunc(int height)
    for(int i=0; i<WIDTH; i++)
        for(int j=0; j<height; j++)
            for(int k=50; k<(WIDTH+height)/2; k+=2 )
                // Computation

Tobias Kenter, Paderborn University
Identify suitable hotspots

Can be executed on the coprocessor

Potential hotspots

Blocks to offload

Aggressive

const int WIDTH = 1024;
void heavyFunc(int height)
    for(int i=0; i<WIDTH; i++)
        for(int j=0; j<height; j++)
            for(int k=50; k<(WIDTH+height)/2; k+=2)
                // Computation

Tobias Kenter, Paderborn University
Identify suitable hotspots

Can be executed on the coprocessor

Potential hotspots

Blocks to offload

The image part with relationship ID rId4 was not found in the file.

Experience

Aggressive

Conservative

const int WIDTH = 1024;
void heavyFunc(int height)
  for(int i=0; i<WIDTH; i++)
    for(int j=0; j<height; j++)
      for(int k=50; k<(WIDTH+height)/2; k+=2 )
        // Computation

Tobias Kenter, Paderborn University
What we investigate

• To what extent does this problem effect real world applications?

• Is it possible to take better decisions at runtime instead of at compile time?

• What are the overheads associated with such a runtime decision?
## Analysis of loop trip count

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># loops</th>
<th>Trip count known at</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td><strong>SPEC CPU’06</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- C/C++</td>
<td>26,131</td>
<td>3,196 (12%)</td>
</tr>
<tr>
<td>- F/F90</td>
<td>43,890</td>
<td>4,065 (9%)</td>
</tr>
<tr>
<td>- Total</td>
<td>70,021</td>
<td>7,261 (10%)</td>
</tr>
<tr>
<td><strong>MiBench</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- C</td>
<td>5,080</td>
<td>617 (12%)</td>
</tr>
<tr>
<td><strong>MediaBench I</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- C</td>
<td>683</td>
<td>182 (27%)</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>75,784</td>
<td>8,060 (11%)</td>
</tr>
</tbody>
</table>

Tobias Kenter, Paderborn University
Inserting the runtime decision

Host Function

Coprocessor Function

fun_A

Block_0
Block_1
Block_2
Block_3
Block_4
Block_5
Block_6
Block_7
Block_8
Block_9

fun_A

Block_0
Runtime Check
Block_1
Block_2
Block_3
Block_4
Block_5
Block_6
Block_7
Block_8
Block_9

Offloading Decision

Migrate data to the coprocessor

Yes

No

Migrate data back to the CPU

Tobias Kenter, Paderborn University
Inserting the runtime decision

Insert code at compile time that performs the decision at runtime
How to compute the trip count at runtime?

- LLVM’s *ScalarEvolution* (SCEV) analysis pass
  - Generates the trip count expression

```c
const int WIDTH = 1024;
void heavyFunc(int height)
    for(int i=0; i<WIDTH; i++)
        for(int j=0; j<height; j++)
            for(int k=50; k<(WIDTH+height)/2; k+=2 )
                // Computation
```

Tobias Kenter, Paderborn University
How to compute the trip count at runtime?

- LLVM’s *ScalarEvolution* (SCEV) analysis pass
  - Generates the trip count expression

```c
const int WIDTH = 1024;
void heavyFunc(int height)
    for(int i=0; i<WIDTH; i++)
        for(int j=0; j<height; j++)
            for(int k=50; k<(WIDTH+height)/2; k+=2 )
                // Computation
```

- Generate LLVM IR that evaluates this expression at runtime
  - LLVM’s SCEVExpander Class

```c
%add         = add i32 %WIDTH, %height
%div         = div i32 %add, 2
%smax        = smax i32 50, %div
%sub         = sub i32 %smax, 50
%trip_count  = div i32 %sub, 2
```
How to compute the trip count at runtime?

- **LLVM’s ScalarEvolution (SCEV) analysis pass**
  - Generates the trip count expression

  ```
  const int WIDTH = 1024;
  void heavyFunc(int height)
      for(int i=0; i<WIDTH; i++)
          for(int j=0; j<height; j++)
              for(int k=50; k<(WIDTH+height)/2; k+=2 )
                  // Computation
  ```

- **Generate LLVM IR that evaluates this expression at runtime**
  - LLVM’s SCEVExpander Class

  ```
  %add = add i32 %WIDTH, %height
  %div = div i32 %ADD, 2
  %smax = smax i32 50, %div
  %sub = sub i32 %smax, 50
  %trip_count = div i32 %sub, 2
  ```

Negligible overhead (less than 1%)
How can we decide when to offload?

- **Empirical:** analysis of last results and projection to next values
  - Sample runs with different input data and measurements
  - Easy to use, but needs initialization/learning

- **Analytical:** accurate estimation model and exhaustive search function
  - Analytically create good abstraction model of influencing values and build search function to guide to optimal solution
  - Accurate, but slow (or inaccurate, and fast)

- **Heuristic:** satisfactory solution based on assumptions and probability models
  - Predefine assumptions to reduce search space and select best solution for given computation time budget
  - Fast, but inconsistent/unsteady
Case Study
Convey HC-1: platform

- 2 socket mainboard
- Xeon CPU
- FPGA coprocessor
  - 4 Virtex 5 LX330 FPGAs
- 80GB/s memory bandwidth
  - scatter/gather RAMs for 8 byte accesses
- Common cache-coherent virtual address space
  - physically distinct locations
- FPGA-based vector personality
Toolflow

Tobias Kenter, Paderborn University
Evaluation

- Test set consists of characteristic loops based on stereo matching kernels
- The loop iteration counts corresponds to popular image formats

<table>
<thead>
<tr>
<th>Image formats</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVD</td>
<td>720x480</td>
</tr>
<tr>
<td>HD DVD</td>
<td>1280x720</td>
</tr>
<tr>
<td>Blu-Ray</td>
<td>1920x1080</td>
</tr>
<tr>
<td>4K</td>
<td>4096x2160</td>
</tr>
<tr>
<td>UHDTVD</td>
<td>7680x4320</td>
</tr>
</tbody>
</table>

- Speedup compared to always executing on the CPU
- Speedup compared to always executing on coprocessor
Speedup compared to always on coprocessor

Speedups of up to 10x can be achieved depending on the loop iteration space.

Beneficial to execute on the CPU as offloading overheads are high.
Speedup compared to always CPU

Beneficial to execute on the coprocessor as offloading overheads can be amortized

Speedups of \textbf{up to 3x} can be achieved by executing code on the coprocessor instead of the CPU
Conclusion & Outlook

• Conclusion

  – Our runtime approach is superior to the approaches that execute code only on the CPU or only on the FPGA

  – By moving the decision from compile time to runtime, we achieve speedups that were previously not achievable by using the traditional approach

  – We achieve a speedup of up to 3x compared to a “CPU only” approach and up to 10x compared to “coprocessor only”

  – The overhead of the runtime decision was negligible (less than 1%)

• Outlook

  – Can this result in significant energy savings?
  – Move in the direction of JIT compilation and offloading
Thank you!

Questions?

Gavin Vaz
gavin.vaz@uni-paderborn.de
Backup
Data Migration

- Just transfer what is required
- Buffers are used for image processing (constant size for different images)
LLVM

• Based on the LLVM infrastructure
• We assume that the LLVM IR is available
Our contributions

Convey infrastructure

LLVM infrastructure
Our contributions
Convey infrastructure
LLVM infrastructure

Tobias Kenter, Paderborn University
Runtime decisions for offloading

- Iteration count often not constant – is offloading worth it?
- Profiling?
  - proper input data?
  - time for profiling?

```cpp
//Integral column sums
for(int x=0; x<width; x++)
    for(int y=1; y<height; y++)
        im[x][y] = im[x][y] + im[x][y-1];
```

- At runtime we could know…
- ScalarEvolution::getBackedgeTakenCount(myLoop)
  - constant: static decision
  - could not compute: no vectorization
  - single value or expression: runtime decision

```cpp
if(width < THRESHOLD)
    columnSumsHost(im);
else
    columnSumsCoprocessor(im);
```
Test setup

- Vectorizable loop structures e.g. from stereo matching
  - structure: 1d, 2d, 3d loops
  - data: 1d, 2d, 3d arrays

- Systematic variation
  - dependency: inner, outer, (fully independent)
  - data layout:
    - favorable = vectors in continuous memory locations
    - transposed
  - multidimensional data structures
    - arrays allowing multidimensional offsets
    - pointer structures requiring a memory access for each dimension

- Dimensions: 5000 iterations of vectorizable loop

- Comparison
  - our toolflow, heterogeneous executable
  - LLVM backend, pure CPU version
  - (Convey vectorizing compiler, few pragmas, heterogeneous executable)
Minimal effort vectorization

• Existing Convey compiler infrastructure
  – C/C++ code annotated with pragmas
    ▪ mark coprocessor regions
    ▪ give dependency hints
    ▪ move data
  – limited in supported features
    ▪ no 2d+ pointer arrays
    ▪ only inner-loop vectorization

• Our goals
  – fully automated
    ▪ handle region detection, vectorization, data movement
  – use LLVM compiler infrastructure
    ▪ starting with LLVM IR allows different input formats

➢ Show that minimal effort, reasonable speedup is possible!

Tobias Kenter, Paderborn University