High-Performance Domain-Specific Languages for GPU Computing

Marcel Köster, Roland Leißa, Sebastian Hack
Richard Membarth, Philipp Slusallek

Intel Visual Computing Institute
Saarland University
German Research Center for Artificial Intelligence (DFKI)
Many-Core Dilemma

Many-Core HW is everywhere – but cannot be programmed well

Intel Haswell Architecture (1.4B Transistors)  
Intel KnightsFerry (~5B Transistors)  
Nvidia Kepler (~7B Transistors)
Approaches

General-Purpose Languages

Domain-Specific Languages (embedding)
Our Approach

AnyDSL Framework

- Computer Vision DSL
- Physics DSL
- Ray Tracing DSL
- Parallel Runtime DSL
- ...
Our Approach

- DSL embedding in own host language

- Partial evaluation
- Triggered code generation
- Typesafe
Impala: A Base Language for DSL Embedding

- Impala is an imperative & functional language
- A dialect of Rust (http://rust-lang.org)
- Partial evaluation is triggered by annotation @

```rust
fn dot(n: int,
    u: [float],
    v: [float]
) -> float {
    let mut sum = 0.0f;
    foreach i in range(n) {
        sum += u[i]*v[i];
    }
    return sum;
}
```

// specialization at call-site
result = @dot(3, a, b);

// specialized code for dot-call
result = 0;
result += a[0]*b[0];
result += a[1]*b[1];
result += a[2]*b[2];
Sample DSL: Stencil Codes in Impala

- Application-specific code
  - Applies a given stencil to a single pixel
  - Compiler exposes partial evaluation through `@`

```c
fn apply_stencil(stencil: [float],
                 arr: [float],
                 index: int
     ) -> float {
    let mut sum = 0.0f;
    let half = stencil.width / 2;

    foreach i in range(-half, half+1) {
        foreach j in range(-half, half+1) {
            sum += arr[...] * stencil[...];
        }
    }

    return sum;
}
```
Sample DSL: Stencil Codes in Impala

- Application-specific code
- Applies a given stencil to a single pixel
- Compiler exposes partial evaluation through @

```rust
fn main() {
    let mut arr = ...;
    let mut out = ...;
    let a = 0.2f, b = 1.0f - 4.0f * a;
    let stencil = [
        [0.0f, b, 0.0f],
        [b, a, b],
        [0.0f, b, 0.0f]];

    foreach i in field(width, height) {
        out[i] = apply_stencil(stencil, arr, i);
    }
}
```
Sample DSL: Stencil Codes in Impala

- Application-specific code
  - Applies a given stencil to a single pixel
  - Compiler exposes partial evaluation through @

```rust
def main() {
    let mut arr = ...;
    let mut out = ...;
    let a = 0.2f, b = 1.0f - 4.0f * a;
    let stencil = [
        [0.0f, b, 0.0f],
        [b, a, b],
        [0.0f, b, 0.0f]];

    field(width, height, |i| -> void {
        out[i] = apply_stencil(stencil, arr, i);
    });
}
Sample DSL: Stencil Codes in Impala

- Application-specific code
  - Applies a given stencil to a single pixel
  - Compiler exposes partial evaluation through @

```rust
fn main() {
    let mut arr = ...;
    let mut out = ...;
    let a = 0.2f, b = 1.0f - 4.0f * a;
    let stencil = [
        [0.0f, b, 0.0f],
        [b, a, b],
        [0.0f, b, 0.0f]];

    field(width, height, |i| -> void {
        out[i] = @apply_stencil(stencil, arr, i);
    });
}
```
Mapping to Target Hardware

- Mapping and scheduling
- Exposed NVVM (Cuda) code generation

```rust
fn field(width: int, height: int, body: fn(int) -> void) -> void {
    let dim = (width, height, 1);
    let block = (32, 4, 1);

    nvvm(dim, block, || -> void {
        let tid_x = nvvm_tid_x() + nvvm_ntid_x() * nvvm_ctaid_x();
        let tid_y = nvvm_tid_y() + nvvm_ntid_y() * nvvm_ctaid_y();
        let index = tid_y * width + tid_x;
        body(index);
    });
}
```

- Additional support for
  - Vectorization, SPIR
  - C-Source, OpenCL Source
Exploiting Boundary Handling

- **Boundary handling**
  - Evaluated for all points
  - Unnecessary evaluation of conditionals
Exploiting Boundary Handling

- **Boundary handling**
  - Evaluated for all points
  - Unnecessary evaluation of conditionals

- Specialized variants for different regions

- Automatic generation of variants → Partial evaluation
Exploiting Boundary Handling (cont.)

- Specialized implementation
  - Wrap memory access in `access` function
  - Distinction of variant via `region` variable
  - Specialization discards unnecessary checks

```plaintext
fn access(arr: [float],
          region : int,
          idx: int
) -> float {
    if (region == 0) idx = ... // bh lower;
    if (region == 1) idx = ... // bh upper;
    return arr[idx];
}
```
Exploiting Boundary Handling (cont.)

- Specialized implementation
- Application code

```rust
def main() {
    let mut arr = ...;
    let mut out = ...;
    let stencil = [0.25f, 0.5f, 0.25f];

    foreach i in field(0, arr.size) {
        out[i] = @apply_stencil(stencil, arr, i, region);
    }
}
```
Exploiting Boundary Handling (cont.)

- Specialized implementation
- Application code

```rust
fn main() {
    let mut arr = ...;
    let mut out = ...;
    let stencil = [0.25f, 0.5f, 0.25f];

    let offset = stencil.size / 2;
    //       left    right              center
    let L = [0, arr.size - offset, offset];
    let U = [offset, arr.size, arr.size - offset];

    foreach region in range(0, 3) {
        foreach i in field(L[region], U[region]) {
            out[i] += @apply_stencil(stencil, arr, i, region);
        }
    }
}
```
Exploiting Boundary Handling (cont.)

- Specialized implementation
- Application code

```rust
fn main() {
    let mut arr = ...;
    let mut out = ...;
    let stencil = [0.25f, 0.5f, 0.25f];

    let offset = stencil.size / 2;

    // left    right              center
    let L = [0, arr.size - offset, offset];
    let U = [offset, arr.size, arr.size - offset];

    foreach region in @range(0, 3) {
        foreach i in field(L[region], U[region]) {
            out[i] += apply_stencil(stencil, arr, i, region);
        }
    }
}
```
Exploiting Boundary Handling (cont.)

Specialized implementation

Specialized code

```rust
...  
foreach i in field(0, 1) {                                  // left region
    let mut sum = 0;
    sum += arr[bh_lower(i - 1)] * 0.25f;
    sum += arr[bh_lower(i + 0)] * 0.5f;
    sum += arr[bh_lower(i + 1)] * 0.25f;
    out[i] = sum;
}

foreach i in field(arr.size - 1, arr.size) {    // right region
    ...
}

foreach i in field(1, arr.size - 1) {            // center region
    ...
}
...
**Jacobi Kernel Evaluation (NVIDIA only)**

- Specialized implementation for given stencil

<table>
<thead>
<tr>
<th></th>
<th>GTX 580</th>
<th>GTX 680</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA (hand-specialized)</td>
<td>0.32</td>
<td>0.35</td>
</tr>
<tr>
<td>Impala (specialized)</td>
<td>0.32</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Image of 2048x2048, runtime in ms

- **Cuda version**
  - Stencil hardcoded
  - Default blocking

- **Impala version**
  - Automatically specialized (@)
  - Default blocking
Jacobi Kernel Evaluation (NVIDIA only)

- Hardware-specific mapping (Cuda)

<table>
<thead>
<tr>
<th></th>
<th>GTX 580</th>
<th>GTX 680</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA (hand-specialized)</td>
<td>0.32</td>
<td>0.35</td>
</tr>
<tr>
<td>CUDA (hand-tuned)</td>
<td>0.26</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Image of 2048x2048, runtime in ms

- GTX 580
  - Blocking: 32x4
  - Unroll (iteration space): 128x
  - No textures

- GTX 680
  - Blocking: 32x4
  - Unroll (iteration space): 2x
  - Linear 1D texture memory
Jacobi Kernel Evaluation (NVIDIA only)

Hardware-specific mapping (Impala)

<table>
<thead>
<tr>
<th></th>
<th>GTX 580</th>
<th>GTX 680</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA (hand-tuned)</td>
<td>0.26</td>
<td>0.23</td>
</tr>
<tr>
<td>Impala (+ tuned)</td>
<td>0.24</td>
<td>0.23</td>
</tr>
</tbody>
</table>

```c
fn field(width: int, height: int, body: fn(int, int) -> void) -> void {
    let unroll = N;
    let dim = (width, height / unroll, 1);
    let block = (32, 4, 1);

    nvvm(dim, block, || -> void {
        let tid_x = nvvm_tid_x() + nvvm_ntid_x() * nvvm_ctaid_x();
        let tid_y = nvvm_tid_y() + nvvm_ntid_y() * nvvm_ctaid_y() * unroll;
        @iterate(0, unroll+1, |i| -> void {
            body(tid_x, tid_y + i * nvvm_ntid_y());
        });
    });
}
```
Conclusion & Future Work

- AnyDSL Framework
  - Novel code-refinement concept
  - Explicit control over
    - Partial evaluation
    - Target code-generation functionality (NVVM, SPIR, ...)

- Sample DSL for Stencil Codes
  - Embedded in Impala

- Next step
  - Explicit control over memory-mapping
Thank you for your attention.
Questions?