Programmable Accelerators

Jason Lowe-Power

powerjg@cs.wisc.edu

cs.wisc.edu/~powerjg
Increasing specialization

Need to **program** these accelerators

Challenges

1. Consistent pointers
2. Data movement
3. Security (Fast)

This talk: GPGPUs
Programming accelerators (baseline)

```c
void add(int*a, int*b, int*c) {
    for (int i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }
}

int main() {
    int a[N], b[N], c[N];
    init(a, b, c);
    add(a, b, c);
    return 0;
}
```

Accelerator-side code

CPU-Side code
Programming accelerators (GPU)

```c
void add_gpu(int*a, int*b, int*c) {
    for (int i = get_global_id(0); i < N; i += get_global_size(0)) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    int a[N], b[N], c[N];
    init(a, b, c);
    add(a, b, c);
    return 0;
}
```

Accelerator-side code

CPU-Side code
void add_gpu(int*a, int*b, int*c) {
    for (int i = get_global_id(0); i < N; i += get_global_size(0))
    {
        c[i] = a[i] + b[i];
    }
}

int main() {
    int a[N], b[N], c[N];
    int *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, N*sizeof(int));
    cudaMalloc(&d_b, N*sizeof(int));
    cudaMalloc(&d_c, N*sizeof(int));
    init(a, b, c);
    cudaMemcpy(d_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
    add_gpu(a, b, c);
    cudaMemcpy(c, d_c, N*sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(d_a); cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}
void add_gpu(int*a, int*b, int*c) {
    for (int i = get_global_id(0);
        i < N;
        i += get_global_size(0))
    {
        c[i] = a[i] + b[i];
    }
}

int main() {
    int a[N], b[N], c[N];
    init(a, b, c);
    add_gpu(a, b, c);
    return 0;
}

Accelerator-side code

CPU-Side code
Key challenges

```
Memory

CPU

MMU

a[i]

Virtual address pointer

ld: 0x1000000

Cache

Physical address

0x5000

Memory

0x1000000```

Virtual address pointer

ld: 0x1000000

Physical address

0x5000
Key challenges

Consistent pointers

Virtual address pointer

Data movement

Memory

Virtual address: 0x1000000

0x5000

ld: 0x1000000

GPU

Cache

MMU

GPU

CPU

MMU

Cache

Virtual address pointer

Consistent pointers

Data movement

Memory
Consistent pointers
Supporting x86-64 Address Translation for 100s of GPU Lanes

Data movement
Heterogeneous System Coherence

[HPCA 2014]  [MICRO 2014]
Why not CPU solutions?

It’s all about bandwidth!

Translating 100s of addresses

500 GB/s at the directory (many accesses per-cycle)

*NVIDIA via anandtech.com
Consistent pointers
Supporting x86-64 Address Translation for 100s of GPU Lanes

Data movement
Heterogeneous System Coherence

[HPCA 2014]

[MICRO 2014]
Why virtual addresses?
Why virtual addresses?

Virtual memory

Simply copy data

Transform to new pointers

Transform to new pointers

GPU address space
Bandwidth problem

Virtual memory requests

Physical memory requests

CPU

TLB
Bandwidth problem

GPU Processing Elements
(one GPU core)
Solution: Filtering

GPU Processing Elements (one GPU core)

- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane
- Lane

Shared Memory (scratchpad)

- 1x
- 0.45x

Coalescer

- 0.06x

TLB
Poor performance

Average 3x slowdown

Shared page walk unit
Page table walker

Performance relative to ideal MMU

callbacks, bis, gaussian, hotspot, lud, nn, nw, pathfinder, sort, Average
Bottleneck 1: Bursty TLB misses

Average: 60 outstanding requests
Max 140 requests

Huge queuing delays

Solution:
Highly-threaded pagetable walker
Bottleneck 2: High miss rate

Large 128 entry TLB doesn’t help

Many address streams

**Need low latency**

Solution: **Shared page-walk cache**
Performance: Low overhead

Worst case: 12% slowdown

Average: Less than 2% slowdown
Consistent pointers
Supporting x86-64 Address Translation for 100s of GPU Lanes

Shared virtual memory is important

Non-exotic MMU design
  • Post-coalescer L1 TLBs
  • Highly-threaded page table walker
  • Page walk cache

Full compatibility with minimal overhead

Still room to optimize
Consistent pointers
Supporting x86-64 Address Translation for 100s of GPU Lanes

Data movement
Heterogeneous System Coherence

[HPCA 2014]

[MICRO 2014]
Legacy Interface

1. CPU writes memory
2. CPU initiates DMA
3. GPU direct access

High bandwidth
No directory access
CC Interface

1. CPU writes memory

2. GPU access

**Bottleneck: Directory**

1. Access rate
2. Buffering

Memory
Directory Bottleneck 1: Access rate

Many requests per cycle
Difficult to design multi-ported directory
Directory Bottleneck 2: Buffering

Must track many outstanding requests

Huge queuing delays

Solution: Reduce pressure on directory
HSC Design

Goal:
Direct access (B/W) + Cache coherence

Add:
Region Directory
Region Buffers

Decouples permission from access
HSC: Performance Improvement

The diagram illustrates the performance improvement in HSC, with comparative normalized speed-up for different benchmarks. The benchmarks include bp, bfs, hs, lud, nw, km, sd, bn, dct, hg, and mm. The normalized speed-up ranges from 0 to 5, with higher values indicating better performance improvement.
Want cache coherence without sacrificing bandwidth

Major bottlenecks in current coherence implementations
1. High bandwidth difficult to support at directory
2. Extreme resource requirements

Heterogeneous System Coherence
Leverages spatial locality
Reduces bandwidth and resource requirements by 95%
Increasing specialization

Need to **program** these accelerators

Challenges
1. Consistent pointers
2. Data movement
3. Security (Fast)

This talk: GPGPUs

* NVIDIA via anandtech.com
Security & tightly-integrated accelerators

What if accelerators come from 3rd parties?

Untrusted!

All accesses via IOMMU
Safe
Low performance

Bypass IOMMU
High performance
Unsafe
Border control: sandboxing accelerators

Solution:
Border control

Key Idea: Decouple translation from safety

Safety + Performance
Conclusions

Goal: Enable programmers to use the whole chip

Challenges

1. Consistent addresses
   GPU MMU Design

2. Data movement
   Heterogeneous System Coherence

3. Security
   Border Control
Consistent pointers
Supporting x86-64 Address Translation for 100s of GPU Lanes
[HPCA 2014]

Data movement
Heterogeneous System Coherence
[MICRO 2013]

Security
Border Control: Sandboxing Accelerators
[MICRO 2015]

I’m on the job market this year!
Graduating in Spring

Contact: powerjg@cs.wisc.edu
            cs.wisc.edu/~powerjg

Jason Power, Mark D. Hill, David A. Wood


Lena E. Olson, Jason Power, Mark D. Hill, David A. Wood

Jason Lowe-Power
## Other work

### Analytic database + Tightly-integrated GPUs

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BPOE 2016]</td>
<td>When to use 3D Die-Stacked Memory for Bandwidth-Constrained Big-Data Workloads</td>
<td>Jason Lowe-Power, Mark D. Hill, David A. Wood</td>
</tr>
<tr>
<td>[DaMoN 2015]</td>
<td>Towards GPUs being mainstream in analytic processing</td>
<td>Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, David A. Wood</td>
</tr>
<tr>
<td>[SIGMOD Rec. 2015]</td>
<td>Implications of Emerging 3D GPU Architecture on the Scan Primitive</td>
<td>Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, David A. Wood</td>
</tr>
</tbody>
</table>

### Simulation Infrastructure

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CAL 2014]</td>
<td>gem5-gpu: A Heterogeneous CPU-GPU Simulator</td>
<td>Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, David A. Wood</td>
</tr>
</tbody>
</table>
Comparison to CAPI/OpenCAPI

<table>
<thead>
<tr>
<th></th>
<th>Same virtual address space</th>
<th>Cache coherent</th>
<th>System safety from accelerator</th>
<th>Assumes on-chip accel.</th>
<th>Allows accel. physical caches</th>
<th>Allows pre-translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPI</td>
<td>Yes ✔</td>
<td>Yes ✔</td>
<td>Yes ✔</td>
<td>No ✔</td>
<td>No ✗</td>
<td>No ✗</td>
</tr>
<tr>
<td>My work</td>
<td>Yes ✔</td>
<td>Yes ✔</td>
<td>Yes ✔</td>
<td>Yes ✗</td>
<td>Yes ✔</td>
<td>Yes ✔</td>
</tr>
</tbody>
</table>

Allows for high-performance accelerator optimizations