Robust GPU Architectures
Improving Irregular Execution on Architectures Tuned for Regularity

Mattan Erez

The University of Texas at Austin
Lots of interesting multi-level projects

- Resilience/Reliability
- Memory
- GPU/CPU/Superscale
Arch-focused whole-system approach

- Efficiency requirements require crossing layers
- Algorithms are key
  - Compute less, move less, store less
- Proportional systems
  - Minimize waste
- Utilize and improve emerging technologies
- Explore (and act) up and down
  - Programming model to circuits
- Preferably implement at micro-arch/system
Big problems and emerging platforms

- **Memory systems**
  - Capacity, bandwidth, efficiency – impossible to balance
  - Adaptive and dynamic management helps
  - New technologies to the rescue?
  - Opportunities for in-memory computing

- **GPUs, clouds, and more**
  - Throughput oriented designs are a must
  - Centralization trend is interesting

- **Reliability and resilience**
  - More, smaller devices – danger of poor reliability
  - Trimmed margins – less room for error
  - Hard constraints – efficiency is a must
  - Scale exacerbates reliability and resilience concerns
Lots of interesting multi-level projects

- Resilience/Reliability
- Memory
- GPU/CPU/Superscale
Regularity is good for hardware

- Regularity exploited to amortize overheads
- SIMD/vectors
- Long(ish) cache lines and memory transfers
Good algorithms often irregular

- Graphs
- Sparse structures
- Early-exit conditions
- Specialization
- ...

(C) Mattan Erez
What to do?

- Tune and regularize algorithm implementation
  - Works great
  - Hard to do
  - Sometimes impossible

- Build hardware for irregularity
  - Fine-grained control and access
  - Need to make it cheap enough

- Need to not penalize regular(ized) algorithms!

- Need a robust architecture
Dynamic adaptivity and flexibility

• Hardware mostly designed for regularity
  – Negligible impact on regular portions

• Low-overhead structures support irregularity
  – Performance and efficiency gains on irregular parts

• Improving irregular control on wide-SIMD GPUs
  – Dynamically “regularize” diverged branches
  – Utilize divergence to increase TLP

• Improving fine-grain memory access
  – Locality-aware memory hierarchy
  – Dynamic granularity accesses
Outline

• Robust control for wide-SIMD GPUs
  – Brief “GPU” overview
  – Dynamically “regularize” diverged branches
    • Warp compaction
    • Eliminating its impact on regular code
  – Utilize divergence to increase TLP
    • Dual-path execution

• Robust memory access
  – Dynamic granularity accesses
  – Locality-aware memory hierarchy for GPUs
  – Robust caching

• The real credit belongs to the students
  – Min Kyu Jeong, Minsoo Rhu, Michael Sullivan, Doe Hyun Yoon, Dong Li
BRIEF GPU OVERVIEW
Graphic Processing Units (GPUs)

• General-purpose many-core accelerators
  – Use 100s of simple in-order shader cores
  – Shader core == streaming multiprocessor (SM) in NVIDIA GPUs

• Scalar frontend (fetch & decode) + parallel backend
  – Amortize the cost of frontend and control
Exposed hierarchy of data-parallel threads

- **SPMD**: single *kernel* executed by numerous *scalar* threads
- **Kernel / Thread-block hierarchy**
  - Kernel composed of many *thread-blocks* (a.k.a. cooperative-thread-arrays (CTAs) or work-groups)
CUDA exposes hierarchy of data-parallel threads

- **SPMD**: single *kernel* executed by numerous *scalar* threads
- **Kernel / Thread-block / Warp / Thread**
  - Multiple *warps* compose a *thread-block* (OpenCL wavefronts)
  - Multiple *threads* (32) compose a warp (OpenCL work-items)

A warp is scheduled as a *batch* of threads
CUDA exposes hierarchy of data-parallel threads

- **SPMD**: single *kernel* executed by numerous *scalar* threads
- **Kernel / Thread-block / Warp / Thread**
  - Multiple *warps* compose a *thread-block*
  - Multiple *threads (32)* compose a warp

: Thread-ID 0, 32, 64 ... execute in physical lane #0
: Thread-ID 2, 34, 66 ... execute in physical lane #2

Each thread execute in designated *home SIMD lane*: (thread-ID) mod (SIMD width)
GPUs have HW support for conditional branches

- **SIMT**: Single-Instruction Multiple-Thread
  - Underlying hardware uses *vector* SIMD pipelines (lanes)
  - Programmer writes code to be executed by *scalar* threads
  - Hardware/software supports conditional branches
    - Each thread can follow its own control flow
SIMT stack-based reconvergence model

- Current GPUs use per-warps HW stack to manage control flow
  - Always execute active-threads at the top-of-stack (TOS)
  - Active bitmasks of taken/not-taken path dynamically derived
  - Reconverge taken/not-taken path at reconvergence point
  - Reconvergence PC (RPC) point derived at compile-time

- RPC: immediate post-dominator (PDOM) of the branching-point

(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management
SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute active-threads at the top-of-stack (TOS)
  - Active bitmasks of taken/not-taken path *dynamically* derived
  - Reconverge taken/not-taken path at reconvergence point
  - Reconvergence PC (RPC) point derived at compile-time
    - RPC: *immediate post-dominator (PDOM)* of the branching-point

(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management
SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute active-threads at the top-of-stack (TOS)
  - Active bitmasks of taken/not-taken path dynamically derived
  - Reconverge taken/not-taken path at reconvergence point
  - Reconvergence PC (RPC) point derived at compile-time

- RPC: immediate post-dominator (PDOM) of the branching-point

(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management
SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute active-threads at the top-of-stack (TOS)
  - Active bitmasks of taken/not-taken path dynamically derived
  - Reconverge taken/not-taken path at reconvergence point
  - Reconvergence PC (RPC) point derived at compile-time
- RPC: immediate post-dominator (PDOM) of the branching-point

(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management
SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute active-threads at the top-of-stack (TOS)
  - Active bitmasks of taken/not-taken path dynamically derived
  - Reconverge taken/not-taken path at reconvergence point
  - Reconvergence PC (RPC) point derived at compile-time

- RPC: immediate post-dominator (PDOM) of the branching-point

(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management
Is this dynamic adaptive HW?

• Yes!
  – Tuned for regularity
  – Supports irregular

• But irregular control often performs poorly
  – Only fraction of lanes is active
  – Reduced effective TLP
    • Paths serialized and each has fraction of active threads
  – Reduced performance and efficiency
    • SIMD units under utilized
Exploiting irregular control to improve parallelism

DUAL PATH EXECUTION
Serialization of execution paths

- Execution can transition to the next path only when all the instructions in the current path (basic-block) are executed.
  - CFG is managed with STACK!

- Misses opportunities to schedule from other concurrent paths
  - e.g., paths B/C

(a) Example Control Flow Graph

(b) Execution flow using baseline mechanism.

idle cycles: phases where the scheduler is short of warps to schedule (i.e. cache miss, long-latency ops)
Dual-Path Execution Model (DPE)

- Increase thread-level parallelism (or *path-parallelism*)
  - Without sacrificing SIMD lane utilization

- Maintains the *simplicity* of the baseline stack model

- Minimal implementation overhead
  - Purely a microarchitectural solution
  - No additional compiler support needed
Dual-Path Execution Model (DPE)

- Increase thread-level parallelism (or *path-parallelism*)
  - Without sacrificing SIMD lane utilization
  - Previous work *compromises* SIMD efficiency for enhanced TLP

- Maintains the *simplicity* of the baseline stack model

- Minimal implementation overhead
  - Purely a microarchitectural solution
  - No additional compiler support needed

---

Meng et al., “Dynamic Warp Subdivision for Integrated Branch and Memory Divergence”, ISCA-2010
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates *both* paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

RPC value is *shared* by both Left & Right paths (always at PDOM)

(a) Example Control Flow Graph

(b) Using the DPE stack for control flow management

(c) Execution flow using DPE.
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates both paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

(a) Example Control Flow Graph

(b) Using the DPE stack for control flow management

(c) Execution flow using DPE.
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates *both* paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

(a) Example Control Flow Graph

PUSH paths B and C as a *single* operation

(b) Using the DPE stack for control flow management

(c) Execution flow using DPE.
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates *both* paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

(a) Example Control Flow Graph

(b) Using the DPE stack for control flow management

(c) Execution flow using DPE.
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates both paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

(a) Example Control Flow Graph
(b) Using the DPE stack for control flow management
(c) Execution flow using DPE.

*Invalidated D when reaching RPC (F)*

<table>
<thead>
<tr>
<th>PC&lt;sub&gt;L&lt;/sub&gt;</th>
<th>Mask&lt;sub&gt;L&lt;/sub&gt;</th>
<th>PC&lt;sub&gt;R&lt;/sub&gt;</th>
<th>Mask&lt;sub&gt;R&lt;/sub&gt;</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>1111</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>1000</td>
<td>F</td>
<td>0111</td>
<td>G</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>E</td>
<td>0011</td>
<td>F</td>
</tr>
</tbody>
</table>

SIMD Lanes

- L3
- L2
- L1
- L0

Time

Saved

Saved

Saved

Saved
DPE components: dual-path stack

- **DPE stack microarchitecture**
  - Each entry accommodates *both* paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

### Example Control Flow Graph

- **POP stack when E reaches RPC (F)**

### Dual-path stack

<table>
<thead>
<tr>
<th>PC&lt;sub&gt;L&lt;/sub&gt;</th>
<th>Mask&lt;sub&gt;L&lt;/sub&gt;</th>
<th>PC&lt;sub&gt;R&lt;/sub&gt;</th>
<th>Mask&lt;sub&gt;R&lt;/sub&gt;</th>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>1111</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>1000</td>
<td>F</td>
<td>0111</td>
<td>G</td>
</tr>
</tbody>
</table>

### Using the DPE stack for control flow management

(c) Minsoo Rhu & Mattan Erez
DPE components: dual-path stack

- **DPE stack microarchitecture**
  - Each entry accommodates *both* paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

(a) Example Control Flow Graph

(b) Using the DPE stack for control flow management

(c) Execution flow using DPE.
DPE components: scoreboard

- Current GPUs execute *intra*-warp threads back-to-back
- DPE complications because T/NT paths share registers
  - Separate Left/Right scoreboards
  - Shadow-bits for pre-/post-divergence dependencies

(a) SPE Scoreboard
(b) DPE Scoreboard
DPE components: warp scheduler

• Scheduler enhanced to cover both T/NT paths
  – For maximal benefits of DPE, scheduler overhead is doubled

  – ‘Constrained’ scheduler
    • Warp-scheduler is fed with only a single path at the TOS
    • Path to be sent to the scheduler is rotated when a long-latency operation is executed
    • Benefits decrease from 14.9% to 11.7%
Simulation environment

• GPGPU-Sim
  – Cycle-based performance simulator of a GPGPU
  – 15 shader cores (Streaming Multiprocessors, SMs)
  – 1536 threads per core, 32K registers per core
  – Cache: 16kB L1, 768kB Unified L2
  – Shared-memory: 48KB
  – Memory Controller : FR-FCFS
    • 29.6GB/s per channel, 6 channels overall

• Applications
  – Chosen from CUDA-SDK, Rodinia, Parboil, Cuda Zone, etc
  – Will focus on apps exhibiting distinct behavior across different schemes
Average path parallelism

- Num of concurrent paths exposed to the scheduler
  - SPE: always ‘1’
  - DPE: ‘2’ if both L/R path available, and ‘1’ otherwise

- Interleavable branch
  - Both “if”/“else” part active
  - DPE scheduler sees ‘2’ paths

- Non-interleavable branch
  - No ‘else’ part
  - DPE scheduler sees ‘1’ path
Avg. path parallelism / idle cycles / speedup

(a) Average path parallelism (Higher is better)

(b) Normalized idle cycles (Lower is better)

(c) Speedup (Higher is better)

Max: 42.5% ↑
Avg: 20.1% ↑

(c) Minsoo Rhu & Mattan Erez
Avg. path parallelism / idle cycles / speedup

(a) Average path parallelism (Higher is better)

Max: 48.1% ↓
Avg: 10.2% ↓

(b) Normalized idle cycles (Lower is better)

(c) Speedup (Higher is better)

Minsoo Rhu & Mattan Erez
Avg. path parallelism / idle cycles / speedup

(a) Average path parallelism (Higher is better)

(b) Normalized idle cycles (Lower is better)

(c) Speedup (Higher is better)

Max: 41.9% ↑
Avg: 14.8% ↑

(c) Minsoo Rhu & Mattan Erez
IMPROVING SIMD EFFICIENCY
Underutilization of SIMD units

- Number of active threads in a warp decreases every time control diverges.
- Theoretically, only a single SIMD lane might be active at some point, for highly divergent apps. 
  - Nested divergent branches.

(a) Example Control Flow Graph

(b) Execution flow using baseline mechanism.

: Threads (lanes) masked out due to control divergence, remaining idle.
Thread-block compaction (TBC) [Fung’11]

- Dynamically **compact** warps within a **thread-block**
  - **Synchronize** all warps at branches and reconvergence points
  - Threads with **different** home SIMD lanes compacted together

: [0,1,… A,B] refers to active thread-IDs executing in that basic block
: [-] refers to inactive threads, masked out from execution

(a) Example control flow graph
(b) Execution flow without compaction
Thread-block compaction (TBC) [Fung’11]

- Dynamically **compact** warps within a *thread-block*
  - *Synchronize* all warps at branches and reconvergence points
  - Threads with *different* home SIMD lanes compacted together

: [0,1⋯ A,B] refers to active thread-IDs executing in that basic block
: [-] refers to inactive threads, masked out from execution
Thread-block compaction (TBC) [Fung’11]

- Dynamically **compact** warps within a *thread-block*
  - **Synchronize** all warps at branches and reconvergence points
  - Threads with **different** home SIMD lanes compacted together

- Path *B* is compacted, whereas Path *C* is non-compactable

[0, 1, …, A, B] refers to *active* thread-IDs executing in that basic block
[-] refers to *inactive* threads, masked out from execution

(a) Example control flow graph
(b) Execution flow without compaction
(c) Execution flow with compaction
Is compaction dynamic adaptive HW?

• Yes!
  – Better support for diverged control flow

• But ...
  – Regular(ized) control may suffer
  – Doesn’t work as is for many diverged branches
Excessive synchronization overhead

- Compaction is applied at all branching points
  - Effectively generates a HW-induced compaction-barrier
  - Warps arriving at the end of BB must wait for other warps in its CTA
  - Don’t compact at unconditional branches [Narasiman’11]

- TBC+ (TBC with no synchronization at unconditional branches)
Branch divergence & *ideal* compactability*

- **Conditional** branches, categorized as:
  - Divergent and *ideally* compactable (D/C)
  - Divergent but non-compactable *even with ideal* (D/NC)
  - Non-divergent (ND)

* Ideally compactable branches: branches that can be compacted if threads can switch its executing SIMD lanes
Branch divergence & *ideal* compactability*

- **Conditional** branches, categorized as:
  - Divergent and *ideally* compactable (D/C)
  - Divergent but non-compactable even with ideal (D/NC)
  - Non-divergent (ND)

* Ideally compactable branches: branches that can be compacted if threads can switch its executing SIMD lanes

**Most conditional branches are ND or D/NC!**

: **Why wait compaction?**

Branches

- MUM
- BFS
- BITONIC
- REDUCT
- MDBROT
- LPS
- DXTC
- BACKP
- AOSSORT
- FDTD3D
- SORTNW
- EIGENL
- 3DFD
- DWHARR
- QSRDM
- BINOM
- CONVSEP
- SOBFLT

![Bar Chart](chart.png)
**CAPRI: Compaction-Adequacy Prediction**

- **Intuition**
  - Not all branch points are likely to be compactable
  - *Speculate* whether compaction is worth it
    1) Activate compaction *only when* necessary
      - When past history says compaction was beneficial
    2) *Bypass* warps from compaction-barrier when inadequate

- **Compaction-adequacy**
  - Compaction is *adequate* when the number of executing warps was reduced at a particular branch

- **Microarchitecture**
  - Per-core *prediction table* (32-entry, *TAG*: PC of branch)
  - Works like a simple single-level branch predictor

CAPRI accuracy

(a) Divergent Benchmarks

(b) Non-divergent Benchmarks
Average SIMD lane utilization*

* Average number of SIMD lanes occupied when a warp is issued

(a) Divergent Benchmarks

(b) Non-divergent Benchmarks

Higher is better

CAPRI: 2 ~ 5% ↓
Average SIMD lane utilization*

* Average number of SIMD lanes occupied when a warp is issued

(a) Divergent Benchmarks

(b) Non-divergent Benchmarks

Higher is better
Idle cycles (normalized)

Lower is better

(a) Divergent Benchmarks

(b) Non-divergent Benchmarks
Overall performance

(a) Divergent Benchmarks

Higher is better

(b) Non-divergent Benchmarks
Limited applicability of compaction

- Average SIMD lanes occupied for execution
  - No_TBC: Baseline architecture without compaction
  - TBC: baseline compaction mechanism

- Threads *maintain* execution in its home SIMD lane

Higher is better
Limited applicability of compaction

- Average SIMD lanes occupied for execution

TBC, in general, only effective for **highly divergent** applications

Higher is better
Why does compaction fail?

- Most (or all) of SIMD lanes *already* occupied
  - Compaction inherently impossible

### Non-divergent

<table>
<thead>
<tr>
<th>Active Threads</th>
<th>0123</th>
<th>456-</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀</td>
<td>0123</td>
<td>0123</td>
</tr>
<tr>
<td>W₁</td>
<td>89AB</td>
<td>4567</td>
</tr>
<tr>
<td>W₂</td>
<td>89AB</td>
<td>89AB</td>
</tr>
<tr>
<td>W₃</td>
<td>CDE-</td>
<td>CD--</td>
</tr>
</tbody>
</table>

### Divergent, but non-compactable

<table>
<thead>
<tr>
<th>Active Threads</th>
<th>0123</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀</td>
<td>0123</td>
</tr>
<tr>
<td>W₁</td>
<td>456-</td>
</tr>
<tr>
<td>W₂</td>
<td>89AB</td>
</tr>
<tr>
<td>W₃</td>
<td>CDE-</td>
</tr>
</tbody>
</table>

(c) Minsoo Rhu & Mattan Erez
Why does compaction fail?

• Most (or all) of SIMD lanes *already* occupied
  – Compaction inherently impossible

<table>
<thead>
<tr>
<th>Active Threads</th>
<th>Active Threads</th>
<th>Active Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_0$ 0123</td>
<td>$W_0$ 0123</td>
<td>$W_0$ 0123</td>
</tr>
<tr>
<td>$W_1$ ----</td>
<td>$W_1$ 456-</td>
<td>$W_1$ 4567</td>
</tr>
<tr>
<td>$W_2$ 89AB</td>
<td>$W_2$ 89AB</td>
<td>$W_2$ 89AB</td>
</tr>
<tr>
<td>$W_3$ ----</td>
<td>$W_3$ CDE-</td>
<td>$W_3$ CD--</td>
</tr>
</tbody>
</table>

• Active threads *aligned (clustered)* on certain lanes
  – Compaction theoretically feasible, if crossbars can allow threads to execute in different SIMD lanes

<table>
<thead>
<tr>
<th>Active Threads</th>
<th>Active Threads</th>
<th>Active Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_0$ 0----</td>
<td>$W_0$ 01--</td>
<td>$W_0$ 0-2-</td>
</tr>
<tr>
<td>$W_1$ 4----</td>
<td>$W_1$ 45--</td>
<td>$W_1$ 4-6-</td>
</tr>
<tr>
<td>$W_2$ 8----</td>
<td>$W_2$ 89--</td>
<td>$W_2$ 8-A-</td>
</tr>
<tr>
<td>$W_3$ C------</td>
<td>$W_3$ CD--</td>
<td>$W_3$ C-E--</td>
</tr>
</tbody>
</table>
Why does compaction fail?

- Most (or all) of SIMD lanes *already* occupied
  - Compaction inherently impossible

- Active threads *aligned (clustered)* on certain lanes
  - Compaction theoretically feasible, if crossbars can allow threads to execute in different SIMD lanes

**Aligned Divergence!**
**Aligned divergence – (1)**

- **Case 1:** Branch condition depends on *data* array
  - Each thread references *different* element of the array
  - Unlikely to cause aligned divergence

---

**Code #1**  Branch depending on *data* arrays

```c
// Code snippet from the kernel of BFS benchmark
// *g_graph_visited* and *g_graph_edges* are data array parameters.

int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;

... 

int id = g_graph_edges[...];

if( !g_graph_visited[id] )
{
    ... 
}
```
**Aligned divergence – (1)**

- **Case 1:** Branch condition depends on *data* array
  - Each thread references *different* element of the array
  - Unlikely to cause aligned divergence

---

**Code #1) Branch depending on data arrays**

```c
// Code snippet from the kernel of BFS benchmark
// g_graph_visited and g_graph_edges are data array parameters.
int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;

... 

int id = g_graph_edges[...];
if( !g_graph_visited[id] )
{
    ...
}
```

---

**D-Branches**
**Aligned divergence – (2)**

- **Case 2:** Branch cond. depends on *programmatic* value
  - *Scalar* input parameters to the kernel, *constants*
  - Threads sharing home SIMD lane likely to reference *same* value

---

### Code #2)
Programmatic branch causing only the 1st half of the warp active

```c
0  // Code snippet from the kernel of BACKP benchmark
1  // CTA is a (8 × 16) 2-D array of threads.
2
3  int tx = threadIdx.x;
4  int ty = threadIdx.y;
5
6  ... for (int i=1; i<=__log2f(HEIGHT); i++){
7     int power_two = __powf(2,i);
8
9      if( ty % power_two == 0 ) {...}
10    ...}
11 }
```
**Aligned divergence – (2)**

- **Case 2:** Branch cond. depends on *programmatic* value
  - *Scalar* input parameters to the kernel, *constants*
  - Threads sharing home SIMD lane likely to reference *same* value

---

**Code #2)** Programmatic branch causing only the 1st half of the warp active

```c
0  // Code snippet from the kernel of BACKP benchmark
1  // CTA is a (8 x 16) 2-D array of threads.
2
3  int tx = threadIdx.x;
4  int ty = threadIdx.y;
5  ...
6  for (int i=1; i<=__log2f(HEIGHT); i++){
7      int power_two = __powf(2,i);
8      if( ty % power_two == 0 ) { ... }
9  }
```

**P-Branches**
Compaction rate of p-/d-branches (with TBC)

- **Definition:** Fraction of *compactable* paths among *all* paths generated by divergent branches.

Branch categorization done with GPUOcelot using Taint-analysis (detailed in paper)
Compaction rate of p-/d-branches (with TBC)

- **Definition**: Fraction of *compactable* paths among *all* paths generated by divergent branches.
Compaction rate of p/d branches (with TBC)

D-branch compacts well with TBC
P-branch not so much compared to Ideal

all paths generated by divergent branches

Ideal: TBC with crossbar
Higher is better
TBC with crossbar (ideal compaction)

- Average SIMD lanes occupied for execution

Higher is better

(c) Minsoo Rhu & Mattan Erez
Significant opportunities to enhance SIMD efficiency by tackling aligned divergence.

But crossbars are expensive!

Higher is better

- MUM
- BFS
- BITONIC
- REDUCT
- MDBROT
- LPS
- DXTC
- BACKP
- AOSSORT

- FDTD3D
- SORTNW
- EIGENVL
- 3DEF
- DWHARR
- QSRDM
- BINOM
- CONVSEP
- SOBFLT

Significant opportunities to enhance SIMD efficiency by tackling aligned divergence.
SIMD Lane Permutation (SLP)*

- **Motivation:** Permute home SIMD lanes to alleviate aligned divergence

* WID: Warp-ID

(a) Compaction as-is,

(b) XOR warps with odd WID by 1

(c) Flip warps with odd WID

(d) Rotate all warps by WID

(e) Rotate warps with odd WID by 1

(f) XOR all warps by bit-reverse of WID

* Rhu et al., “Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation”, ISCA-2013
**Balanced permutation**

- **Observation:** Divergence frequently exhibits a *highly skewed* concentration of active lanes
- **Intuition:** Evenly balance active threads across all lanes

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7 0 0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

(c) Minsoo Rhu & Mattan Erez
**Balanced permutation**

- **Observation:** Divergence frequently exhibits a **highly skewed** concentration of active lanes
- **Intuition:** Evenly balance active threads across all lanes

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

| For W0  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W1  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W2  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W3  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W4  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W5  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W6  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| For W7  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |

Baseline: assigns lane-IDs based on **round-robin** manner
**Balanced permutation**

- **Observation:** Divergence frequently exhibits a **highly skewed** concentration of active lanes
- **Intuition:** Evenly balance active threads across all lanes

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>For W0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>For W7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>For W0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W7</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
**Balanced permutation**

- **Observation**: Divergence frequently exhibits a highly skewed concentration of active lanes.

- **Intuition**: Evenly balance active threads across all lanes.

---

**Balanced permutation algorithm**

- Even-ID warps: XOR lane-ID by \((\text{Warp-ID} \gg 1)\)
- Odd-ID warps: XOR lane-ID by \(~(\text{Warp-ID} \gg 1)\)

---

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>For W0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>For W7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
**Balanced permutation**

- **Observation**: Divergence frequently exhibits a highly skewed concentration of active lanes.
- **Intuition**: Evenly balance active threads across all lanes.

---

### Balanced permutation algorithm

- **Even-ID warps**: XOR lane-ID by \((\text{Warp-ID} \gg 1)\)
- **Odd-ID warps**: XOR lane-ID by \(~(\text{Warp-ID} \gg 1)\)

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>For W0</td>
<td>XOR-000</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>For W1</td>
<td>XOR-111</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>For W2</td>
<td>XOR-001</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>For W3</td>
<td>XOR-110</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>For W4</td>
<td>XOR-010</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>For W5</td>
<td>XOR-101</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>For W6</td>
<td>XOR-011</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>For W7</td>
<td>XOR-100</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Compaction rate

- **Definition**: Fraction of *compactable* paths among all paths generated by divergent branches

- **SLP**: significantly improves $P$-branch compaction rate

![Compaction Rate Graph](image-url)

*Higher is better*
Compaction rate

- P-branches
  - TBC: average 3.2%
  - Odd_Even: average 28.9%
  - Balanced: average 71.5%
  - Ideal: average 72.7%

- D-branches
  - TBC: average 42.5%
  - Odd_Even: average 45.2%
  - Balanced: average 59.3%
  - Ideal: average 68.5%
Average SIMD lane utilization

- **Definition:** average number of SIMD lanes *actually* executing and committing results

```
<table>
<thead>
<tr>
<th></th>
<th>TBC</th>
<th>ROTATE_all</th>
<th>ROTATE_odd_1</th>
<th>FLIP_odd</th>
<th>WID</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITONIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REDUCT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDBROT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACKP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AOSSORT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDTD3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DFD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Higher is better
Average SIMD lane utilization

- **TBC**: lowest utilization due to un-compacted aligned divergence
- **SLP**:
  - **Odd Even**: average 2.1% (max 18%) increase over TBC
  - **Balanced**: average 7.1% (max 34%) increase over TBC

(c) Minsoo Rhu & Mattan Erez
Speedup

- **Baseline**: no compaction
- **TBC+CAPRI** only effective for ‘irregular’ applications
- **SLP** widens the range of applications that benefit from compaction techniques
  - 7.1% (max 34%) improvements on top of TBC+CAPRI

![Graph showing Speedup improvements for various applications](chart.jpg)
IRREGULAR AND FINE-GRAIN MEMORY ACCESS
CG-only and FG-only systems

CG-Only: Wide DRAM channel

FG-Only: Many narrow channels
Latency + throughput + efficiency \rightarrow parallelism

Many pins in parallel over multiple cycles

Access even more cells in parallel
Hierarchy + parallelism

→ potential waste
CG access MAY Waste BW

GUPS microbenchmark

for( i=0; i<N; i++ ) {
    a[ b[i] ] += x;
}

Buffer a

Initialized with random numbers

Waste BW on unused data
Conventional CG-only always retrieves 64B

Only 8B is useful

DRAM 0
DRAM 1
DRAM 2
DRAM 3
DRAM 4
DRAM 5
DRAM 6
DRAM 7
DRAM 8
FG-only is a very expensive solution

Higher throughput for FG access but, at higher cost (ABUS & ECC)
CG and FG Access

- **Coarse-grained access**
  - Control and ECC overheads are amortized over a large block
  - Waste BW when spatial locality is low

- **Fine-grained access**
  - Higher control and ECC overheads
  - Potentially higher throughput when spatial locality is low
Spatial locality in actual applications

Low spatial locality

~50% is referenced

High spatial locality

Low spatial locality

1~2

3~6

7~8

PIN-based profiling with a 1MB cache with a 64B cache line

(c) Minsoo Rhu & Mattan Erez
Dynamic granularity is best of both worlds

Locality Predictor

Sector cache (8 8B sectors per line)

Co-scheduling CG/FG w/ split CG

Sub-Ranked DRAM w/ 2X ABUS
Support both CG and FG
Sub-ranked memory enables DGMS

- Control individual DRAM chips independently
  - Originally proposed (mostly) for energy efficiency w/ CG
    - Threaded module, MC-DIMM, Mini-rank, S/G DIMM
- Access granularity = 8B (8 bit x 8 burst)
Sector Cache

- Caches also need to manage fine-grained data
- We use a simple sector cache
  - Allow partially valid cache lines
  - Low tag overhead

**Long cache line**

| tag | D | V | data |

**Short cache line**

| tag | D | V | data |

**Sector cache**

| tag | D | V | sector 0 | D | V | sector 1 | D | V | sector 2 | D | V | sector 3 |
Common data layout for CG and FG

Burst 8

<table>
<thead>
<tr>
<th>ABUS</th>
<th>Reg/Demux</th>
<th>x8</th>
<th>x8</th>
<th>x8</th>
<th>x8</th>
<th>x8</th>
<th>x8</th>
<th>x8</th>
<th>x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td></td>
<td>B8</td>
<td>B9</td>
<td>B10</td>
<td>B11</td>
<td>B12</td>
<td>B13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
<td>B9</td>
<td>B10</td>
<td>B11</td>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td>B10</td>
<td>B11</td>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td>B11</td>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
<td>B18</td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td></td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
<td>B18</td>
<td>B19</td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td></td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
<td>B18</td>
<td>B19</td>
<td>B20</td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td></td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
<td>B18</td>
<td>B19</td>
<td>B20</td>
<td>B21</td>
<td></td>
</tr>
<tr>
<td>B16</td>
<td></td>
<td>B16</td>
<td>B17</td>
<td>B18</td>
<td>B19</td>
<td>B20</td>
<td>B21</td>
<td>B22</td>
<td></td>
</tr>
<tr>
<td>B32</td>
<td></td>
<td>B32</td>
<td>B33</td>
<td>B34</td>
<td>B35</td>
<td>B36</td>
<td>B37</td>
<td>B38</td>
<td></td>
</tr>
<tr>
<td>B40</td>
<td></td>
<td>B40</td>
<td>B41</td>
<td>B42</td>
<td>B43</td>
<td>B44</td>
<td>B45</td>
<td>B46</td>
<td></td>
</tr>
<tr>
<td>B48</td>
<td></td>
<td>B48</td>
<td>B49</td>
<td>B50</td>
<td>B51</td>
<td>B52</td>
<td>B53</td>
<td>B54</td>
<td></td>
</tr>
<tr>
<td>B56</td>
<td></td>
<td>B56</td>
<td>B57</td>
<td>B58</td>
<td>B59</td>
<td>B60</td>
<td>B61</td>
<td>B62</td>
<td></td>
</tr>
</tbody>
</table>
Common data layout for CG and FG
Throughput & Power Efficiency (4-core)

System throughput

Throughput per power
4-core results

Apps with low spatial locality

- CG+ECC
- FG+ECC
- AG+ECC

System throughput

- mst x4
- em3d x4
- canneal x4
- SSCA2 x4
- linked list x4
- gups x4
8-core results

- CG+ECC
- FG+ECC
- AG+ECC

System Throughput

MIX-A: mcf x4, omnetpp x4
MIX-B: SSCA2 x2, mcf x2, omnetpp x2, mst x2
MIX-C: SSCA, mcf, omnetpp, mst, astar, hmmer, lbm, bzip2
MIX-D: libquantum x2, hmmer x2, mst x2, mcf x2
Irregularity even worse impact on a GPU

- Well-structured memory accesses are coalesced into a single memory transaction
  - e.g., all the threads in a warp access the same cache block
- A single warp (32 threads) can generate up to 32 memory transactions
  - e.g., threads in a warp access distinct cache blocks
- Irregularity severely degrades efficiency
  - Cache thrashing
  - FG memory access
Cache Block Locality (Spatial)

- Number of sectors referenced in L1/L2 cache blocks (CG-only memory-hierarchy, 128-Byte cache block)
Cache Block Locality (Temporal)

- Number of repeated accesses to L1/L2 cache blocks, after fill (CG-only memory-hierarchy)
Memory Hierarchy (CG-only)

- Fermi (GF110) / Kepler (GK110) / Southern-Island
  - Each channel width: 64b
  - 4 ~ 6 channels overall, depending on product variants

- 64B (64b x 8-bursts) minimum access granularity

(a) Baseline cache and memory system (V: valid)
Memory Hierarchy (FG-enabled)

- Each channel divided into two sub-ranks
  - 32B (32b x 8-bursts) minimum access granularity
  - Allows finer control of data fetches from DRAM (64B vs 32B)

(a) Memory hierarchy with a sectored cache and a sub-ranked memory system
   (128B cache block, V: valid)
**LAMAR: Locality-Aware Memory Hierarchy**

- Sectored caches + sub-ranked memory system
  - **Motivation**: massive multithreading + memory divergence limits cache block lifetime
    - Prefetching effectiveness
Bi-modal Predictor (all or on-demand) as GDU

- GPUs contain 10s / 100s number of cores
  - Spatial-pattern predictor is heavy-weight / not scalable

- Dual-bitarray bloom-filter
  - Light-weight, temporally overlapped for history preservation
  - Always maintain subset of insertion-history
Off-chip byte traffic (normalized to # of instructions)

Applications: highly divergent ones which can benefit with FG-accesses

Lower is better
Off-chip byte traffic (normalized to # of instructions)

CG : *Static-GDU with CG-only fetches* (Baseline)
FG : *Static-GDU with FG-only fetches*
Off-chip byte traffic (normalized to # of instructions)

Traffic/Instr. (absolute number)

Lower is better
Off-chip byte traffic (normalized to # of instructions)

Traffic/Instr. (Normalized to CG)

Lower is better
Off-chip byte traffic (normalized to # of instructions)

FG: avg. 53% (max 71%) ↓

FG-fetches reduce number of READ and WRITE transactions through memory channels.

Lower is better
Performance improvement

Higher is better

<table>
<thead>
<tr>
<th>Speedup</th>
<th>IIX</th>
<th>SSSP</th>
<th>BFS1</th>
<th>SP</th>
<th>SSC</th>
<th>BFS2</th>
<th>MUM</th>
<th>NW</th>
<th>PVC</th>
<th>WP</th>
<th>H-Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>1.5</td>
<td>1.4</td>
<td>1.3</td>
<td>1.2</td>
<td>1.1</td>
<td>1.0</td>
<td>1.0</td>
<td>0.9</td>
<td>0.8</td>
<td>0.7</td>
<td>0.6</td>
</tr>
<tr>
<td>FG</td>
<td>1.4</td>
<td>1.3</td>
<td>1.2</td>
<td>1.1</td>
<td>1.0</td>
<td>0.9</td>
<td>0.8</td>
<td>0.7</td>
<td>0.6</td>
<td>0.5</td>
<td>0.4</td>
</tr>
</tbody>
</table>

(c) Minsoo Rhu & Mattan Erez
Performance improvement

FG: avg. 13% (max 47%)
DRAM power consumption

- Power (Watts)
- Perf/Watt (Normalized)

- Background
- ACT/PRE
- READ
- WRITE
- REFRESH
- Perf/Watt

- CG
- FG
- IIX
- SSSP
- BFS1
- SP
- SSC
- BFS2
- MUM
- NW
- PVC
- WP

(c) Minsoo Rhu & Mattan Erez
DRAM power consumption

Power (Watts): Lower is better
DRAM power consumption

- **Perf/Watt**: Higher is better

The diagram shows the DRAM power consumption across different benchmarks and operations, with a focus on Perf/Watt, which is a performance-to-power ratio. Higher Perf/Watt values indicate more efficient power usage per unit of performance.
< DRAM power >
FG: avg. 19% (max 42%) ↓
DRAM power consumption

< Perf/Watt >
FG: avg. 42% (max 121%) ↑
ROBUST GPU CACHING
Memory divergence and caching don’t mix well

- Massive multithreading + irregular memory accesses
  - Bursts of concurrent cache accesses within a given time frame
- Cache capacity per thread very small
  - Only 24B/thread
  - Xeon has 16KB/thread!

**Misses per warp**

- **SCLUSTER**: 100%
- **KMEANS**: 80%
- **COMD**: 60%
- **BFS**: 40%
- **SSSP**: 20%
- **PATHFINDER**: 0%
- **NW**: 32%
- **HOTSPOT**: 21~31%
- **21~20**: 11~20%
- **4~10**: 4~10%
- **3**: 3%
- **2**: 2%
- **1**: 1%
Memory divergence and caching don’t mix well

- Massive multithreading + irregular memory accesses
  - Bursts of concurrent cache accesses within a given time frame
- Cache capacity per thread very small
  - Most cache lines fetched and evicted before reused
Possible solution: throttle parallelism

• Throttle number of schedulable threads
  – Fewer threads $\rightarrow$ less thrashing
    • Rogers et al., MICRO 2012
    • Kayiran et al., PACT 2013

• But, sacrifice latency hiding
  – Not robust

• No better than software tuning
Better solution: Priority-based Cache ALlocation

- Bypass the cache instead of throttling parallelism
- Prioritize some warps for cache allocation/deallocation
  - Oldest $T$ warps get cache-use token
- Other warps use cache opportunistically

![Diagram showing cache allocation and scheduling]

- **I**: Invalid block
- **V**: Valid block from token holder
- **V**: Valid block from non-token holder
PCAL hierarchy

W: schedulable warps
T: # available tokens
PCAL is robust and performant

- Two operating points:
  - ITLP: increase TLP while maintaining hit rate
    - \( T = W_{opt}, \ W > W_{opt} \)
  - MTLP: maintain TLP and improve hit rate
    - \( T > W_{opt}, \ W = W_{opt} \)
Performance improvement (relative to throttling)
L1 miss rate improvements

![Graph showing L1 miss rate improvements for different benchmarks and techniques.](image)
Off-chip throughput improvement (rel. to throttling)
Also working on ...

- DGMS with chipkill
- LAMAR with ECC (and chipkill)
- Addressing memory divergence better on GPUs

- And:
Resilience at scale

• Containment domains
  – Elevate resilience to first-class abstraction
  – Proportional, hierarchical, distributed, and cross-layer
  – Programming model, runtime, compiler, and hardware

• Avoiding SDCs with proportional (low) overhead
  – Approximate duplication
  – Circuit-level detection
  – Algorithm hints and compiler optimizations

• Proportional memory protection
  – Adapt and tune protection
    • Meet diverse and dynamic application and system requirements
  – Virtualized and multi-tier ECC, other adaptations
Resilience for emerging technologies

• Proportional NVM wearout-protection
  – Fine-grained NVM wearout-protection
  – ECC schemes for MLC memories

• On-package memory protection
  – Reliability a huge concern (replacement costs)
  – Limited capacity and granularity concerns
  – Rich design space

• Parallel pipelines with high process variation
  – Effective SIMD timing speculation in near-threshold

• Energy-efficient STT-RAM (w/ Orshansky and Samsung)
  – Dynamic write architecture for high-reliability writes
  – Array islands and adaptive placement
Proportional and adaptive memory

• Resilience schemes

• Adaptive and dynamic access granularity
  – Efficiency of coarse-grained access for spatial locality
  – Performance of fine-grained for highly irregular access
  – Power savings with sub-row DRAM activation

• Reducing interference and improving locality
  – Bank-partitioning eliminates inter-core row conflicts
  – Constructively-interleaved page allocation creates locality in GPU context

• Adaptive QoS for heterogeneous processors
  – Balancing QoS for real-time and best-effort cores
  – Consistent and cooperative QoS prioritization
Up-and-coming platforms

- Improved GPU μarch for divergent control flow
  - Robust and effective thread compaction
  - Increasing parallelism with dual-path execution

- Cloud Radio Access Network (C-RAN)
  (w/ Heath, De Veciana, Evans, and Huawei)
  - Wireless base-stations are expensive and inefficient
  - Desire to co-process signals across base-stations
  - Move processing to a specialized cloud
    - Improved system and facilities
    - Improved maintenance
    - Enable joint processing
Conclusions

• Regularity good for hardware
• Irregular good for (lots of) software
• Dynamic Adaptivity: No-compromise robust architecture

• Support regularity as well as possible
• Introduce features for irregularity
• Can do a lot just at microarch level
  – But even more if involving other parts of the system