Transparent Offloading and Mapping (TOM)

Enabling Programmer-Transparent Near-Data Processing in GPU Systems

Kevin Hsieh
Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O’Connor, Nandita Vijaykumar, Onur Mutlu, Stephen W. Keckler
Many GPU applications are bottlenecked by off-chip memory bandwidth
Opportunity: Near-Data Processing

3D-stacked memory (memory stack)

SM (Streaming Multiprocessor)

Near-data processing (NDP) can significantly improve performance
Near-Data Processing: Key Challenges

- Which operations should we offload?
- How should we map data across multiple memory stacks?
Key Challenge 1

- Which operations should be executed on the logic layer SMs?

\[
T = D0; \quad D0 = D0 + D2; \quad D2 = T - D2; \\
T = D1; \quad D1 = D1 + D3; \quad D3 = T - D3; \\
T = D0; \quad d_{\text{Dst}[i0]} = D0 + D1; \quad d_{\text{Dst}[i1]} = T - D1;
\]
Key Challenge 2

• How should data be mapped across multiple 3D memory stacks?

\[ C = A + B \]
The Problem

• Solving these two key challenges requires **significant programmer effort**

• **Challenge 1:** Which operations to offload?
  • Programmers need to **identify** offloaded operations, and consider **run time behavior**

• **Challenge 2:** How to map data across multiple memory stacks?
  • Programmers need to **map all the operands** in each offloaded operation to the **same memory stack**
Our Goal

Enable near-data processing in GPUs transparently to the programmer
Transparent Offloading and Mapping (TOM)

• **Component 1 - Offloading:** A new programmer-transparent mechanism to identify and decide what code portions to offload
  • The compiler identifies code portions to *potentially* offload based on memory profile.
  • The runtime system decides whether or not to offload each code portion based on runtime characteristics.

• **Component 2 - Mapping:** A new, simple, programmer-transparent data mapping mechanism to maximize data co-location in each memory stack
Outline

• Motivation and Our Approach

• Transparent Offloading

• Transparent Data Mapping

• Implementation

• Evaluation

• Conclusion
TOM: Transparent Offloading

Static compiler analysis

• Identifies code blocks as offloading candidate blocks

Dynamic offloading control

• Uses run-time information to make the final offloading decision for each code block
TOM: Transparent Offloading

**Static compiler analysis**

- Identifies code blocks as offloading candidate blocks

**Dynamic offloading control**

- Uses run-time information to make the final offloading decision for each code block
Static Analysis: What to Offload?

- Goal: Save off-chip memory bandwidth

**Conventional System**

- Load
- Store

**Near-Data Processing**

- Offload

**Compiler uses equations (in paper) for cost/benefit analysis**

Offloading benefit: load & store instructions

Offloading cost: live-in & live-out registers

---

13
Offloading Candidate Block Example

...  

float D0 = d_Src[i0];
float D1 = d_Src[i1];
float D2 = d_Src[i2];
float D3 = d_Src[i3];
float T;

T = D0; D0 = D0 + D2; D2 = T - D2;
T = D1; D1 = D1 + D3; D3 = T - D3;
T = D0; d_Dst[i0] = D0 + D1;
d_Dst[i1] = T - D1;
T = D2; d_Dst[i2] = D2 + D3;
d_Dst[i3] = T - D3;

Code block in Fast Walsh Transform (FWT)
Offloading Candidate Block Example

Offloading benefit outweighs cost

float D0 = d_Src[i0];
float D1 = d_Src[i1];
float D2 = d_Src[i2];
float D3 = d_Src[i3];
float T;

T = D0; D0 = D0 + D2; D2 = T - D2;
T = D1; D1 = D1 + D3; D3 = T - D3;
T = D0; d_Dst[i0] = D0 + D1;
d_Dst[i1] = T - D1;
T = D2; d_Dst[i2] = D2 + D3;
d_Dst[i3] = T - D3;

Cost: Live-in registers
Benefit: Load/store inst

Code block in Fast Walsh Transform (FWT)
The cost of a loop is fixed, but the benefit of a loop is determined by the loop trip count.

The compiler marks the loop as a conditional offloading candidate block, and provides the offloading condition to hardware (e.g., loop trip count > N).
TOM: Transparent Offloading

Static compiler analysis
- Identifies code blocks as offloading candidate blocks

Dynamic offloading control
- Uses run-time information to make the final offloading decision for each code block
When Offloading Hurts: Bottleneck Channel

Transmit channel becomes full, leading to slowdown with offloading.
Memory stack SM becomes full, leading to slowdown with offloading.
Dynamic Offloading Control: When to Offload?

• **Key idea:** offload only when doing so is estimated to be beneficial

• **Mechanism:**
  - The hardware does *not* offload code blocks that increase traffic on a bottlenecked channel
  - When the computational capacity of a logic layer’s SM is full, the hardware does *not* offload more blocks to that logic layer
Outline

• Motivation and Our Approach
• Transparent Offloading
• Transparent Data Mapping
• Implementation
• Evaluation
• Conclusion
TOM: Transparent Data Mapping

- **Goal:** Maximize data co-location for offloaded operations in each memory stack

- **Key Observation:** Many offloading candidate blocks exhibit a *predictable* memory access pattern: fixed offset
Fixed Offset Access Patterns: Example

\[
\text{for (n = 0; n < Nmat; n++)}{
    L_b[n] = -v \times \text{delta} / (1.0 + \text{delta} \times L[n])
}\]

85% of offloading candidate blocks exhibit fixed offset access patterns
Transparent Data Mapping: Approach

• **Key idea:** Within the fixed offset bits, find the memory stack address mapping bits so that they *maximize data co-location* in each memory stack

• **Approach:** Execute a tiny fraction (e.g., 0.1%) of the offloading candidate blocks to find the best mapping among the most common consecutive bits

• **Problem:** How to avoid the overhead of data remapping after we find the best mapping?
Conventional GPU Execution Model

CPU

Launch Kernel

GPU

CPU Memory

GPU Data

GPU Memory
Transparent Data Mapping: Mechanism

Learn the best mapping among the most common consecutive bits

Memory copy happens only after the best mapping is found
There is no remapping overhead
Outline

• Motivation and Our Approach
• Transparent Offloading
• Transparent Data Mapping
• Implementation
• Evaluation
• Conclusion
TOM: Putting It All Together

Make offloading decision
Sends offloading request
Monitors TX/RX memory bandwidth
Outline

• Motivation and Our Approach
• Transparent Offloading
• Transparent Data Mapping
• Implementation
• Evaluation
• Conclusion
Evaluation Methodology

• **Simulator:** GPGPU-Sim

• **Workloads:**
  • Rodinia, GPGPU-Sim workloads, CUDA SDK

• **System Configuration:**
  • 68 SMs for baseline, 64 + 4 SMs for NDP system
  • 4 memory stacks
  • Core: 1.4 GHz, 48 warps/SM
  • Cache: 32KB L1, 1MB L2
  • Memory Bandwidth:
    • GPU-Memory: 80 GB/s per link, 320 GB/s total
    • Memory-Memory: 40 GB/s per link
    • Memory Stack: 160 GB/s per stack, 640 GB/s total
Results: Performance Speedup

30% average (76% max) performance improvement
Results: Off-chip Memory Traffic

13% average (37% max) memory traffic reduction
2.5X memory-memory traffic reduction
More in the Paper

• Other design considerations
  • Cache coherence
  • Virtual memory translation

• Effect on energy consumption

• Sensitivity studies
  • Computational capacity of logic layer SMs
  • Internal and cross-stack bandwidth

• Area estimation (0.018% of GPU area)
Conclusion

• Near-data processing is a promising direction to alleviate the memory bandwidth bottleneck in GPUs

• Problem: It requires significant programmer effort
  • Which operations to offload?
  • How to map data across multiple memory stacks?

• Our Approach: Transparent Offloading and Mapping
  • A new programmer-transparent mechanism to identify and decide what code portions to offload
  • A programmer-transparent data mapping mechanism to maximize data co-location in each memory stack

• Key Results: 30% average (76% max) performance improvement in GPU workloads
Transparent Offloading and Mapping (TOM)

Enabling Programmer-Transparent Near-Data Processing in GPU Systems

Kevin Hsieh
Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee,
Mike O’Connor, Nandita Vijaykumar,
Onur Mutlu, Stephen W. Keckler

SAFARI  Carnegie Mellon  NVIDIA  KAIST  ETH Zürich
Observation on Access Pattern

85% of offloading candidate blocks exhibit fixed offset pattern
Bandwidth Change Equations

\[ BW_{TX} = (REG_{TX} \cdot S_W) - (N_{LD} \cdot Coal_{LD} \cdot Miss_{LD} + N_{ST} \cdot (S_W + Coal_{ST})) \]  
(3)

\[ BW_{RX} = (REG_{RX} \cdot S_W) - (N_{LD} \cdot Coal_{LD} \cdot S_C \cdot Miss_{LD} + 1/4 \cdot N_{ST} \cdot Coal_{ST}) \]  
(4)
Best memory mapping search space

• We only need 2 bits to determine the memory stack in a system with 4 memory stacks. The result of the sweep starts from bit position 7 (128B GPU cache line size) to bit position 16 (64 KB).

• Based on our results, sweeping into higher bits does not make a noticeable difference.

• This search is done by a small hardware (memory mapping analyzer), which calculates how many memory stacks would be accessed by each offloading candidate instance for all different potential memory stack mappings (e.g., using bits 7:8, 8:9, ..., 16:17 in a system with four memory stacks)
Best Mapping From Different Fraction of Offloading Candidate Blocks

- Baseline mapping
- Best mapping in first 0.5% NDP blocks
- Best mapping in all NDP blocks
- Best mapping in first 0.1% NDP blocks
- Best mapping in first 1% NDP blocks

Probability of accessing one memory stack in an offloading candidate instance
Energy Consumption Results

![Energy Consumption Results Diagram]
Sensitivity to Computational Capacity of memory stack SMs

![Speedup Chart]

![Normalized Memory Traffic Chart]
Sensitivity to Internal Memory Bandwidth