Enhancing Programmability, Portability, and Performance with Rich Cross-Layer Abstractions

Nandita Vijaykumar
Thesis Oral
October 11th, 2019

Committee
Co-chairs: Onur Mutlu, Phillip B. Gibbons
James C. Hoe
Mattan Erez (UT Austin)
Wen-Mei Hwu (UIUC)
Problem

Diverse and complex hardware

Efficient resource utilization is crucial

Thread Blocks
Cores
Streaming Multiprocessors

Scratchpad Memory

NUMA memory systems
Optimizing for performance: hardware techniques

Challenge today: Diminishing returns with hardware-only approaches

A key aspect of many optimizations is to predict program behavior
Existing hardware-software interface is limiting

Data Structures

SW Optimizations

Access Patterns

Instruction Set Architecture (ISA)

Virtual Memory

Semantic Gap

Instructions

Memory Addresses
Challenges

- **Hardware techniques**: Limited to heuristic-based and prediction-based solutions
- **Software techniques**: Optimizing compilers, autotuners, runtime systems
  - Little visibility and access to architectural techniques

**Performance**: Inefficient use of hardware resources

**Programmability**: Significant programmer effort to optimize code

**Portability**: Performance is not portable across generations or in the presence of co-running applications
A rich low-overhead cross-layer interface that communicates higher-level application information to hardware enables many hardware-software cooperative mechanisms that significantly improve performance, portability, and programmability.
Approach

Rich Cross-Layer Abstraction

Inform architectural techniques: Enable cross-layer optimizations

Software System

Languages
Compilers

Application

Data Access Properties
Parallelism
Data Structure Semantics
Data Reuse
Data Locality ...

Program

Compute Units
Caches
Memory
Storage
...
A rich low-overhead cross-layer interface that communicates higher-level application information to hardware enables many hardware-software cooperative mechanisms that significantly improve performance, portability, and programmability.
Goals

- **Practicality**: low overhead and minimal changes to existing interfaces
- **Generality**: enable many hardware-software cooperative mechanisms
Thesis statement

A rich low-overhead cross-layer interface that communicates higher-level application information to hardware enables many hardware-software cooperative mechanisms that significantly improve performance, portability, and programmability.
Contributions

1. Expressive Memory (ISCA 2018)
Rich interface for memory optimizations

2. Locality Descriptor (ISCA 2018)
Cross-layer abstraction for expressing data locality

3. Zorua (MICRO 2016)
Decoupling the programming model from resource management

4. Assist Warps (ISCA 2015)
Helper-thread abstraction to leverage underutilization
Talk outline

1. Expressive Memory [ISCA 2018]
   - Analysis of a Rich Hardware-Software Interface [Ongoing]

2. The Locality Descriptor [ISCA 2018]

3. Zorua [MICRO 2016]

4. Assist Warps [ISCA 2015]
Use case 1: Increasing the **portability** of cache optimizations

- Programming for cache locality:
  - SW-based cache optimizations try to fit the working set in the cache
- Examples: hash-join partitioning, cache tiling, stencil pipelining

![Diagram](image)
Problem: Correctly sizing the working set is critical

Must tailor tile size to available cache space
Causes challenges in portability
The portability challenge:

Tune working set

8MB cache

6MB cache

High Performance
Our approach: Enabling intelligent cache management

Enable HW to recognize the semantics of a “tile”: Tile size and access pattern

If tile size > available cache space:
- pin a part of the tile, prefetch the rest (avoid cache thrashing)
Our approach: Enabling intelligent cache management

Tile size is less critical:
Improved programmability and portability

Knowledge of locality semantics enables more intelligent cache management
Less dependence of performance on tile size
SW provides key program information to help HW

Data Structures

Access Patterns

Data Type/Layout

Hardware

Software

Data Placement

Prefetcher

Data Compression
Broader goal: Enable many cross-layer optimizations

Express:
Data structures
Access semantics
Data types
Working set
Reuse
Access frequency
...

Optimizations:
Cache Management
Data Placement in DRAM
Data Compression
Approximation
DRAM Cache Management
NVM Management
NUCA/NUMA Optimizations
...

Benefits:
More efficient HW:
✓ Performance
Reduced burden on SW:
✓ Programmability
✓ Portability
Cross-layer approaches are promising but hard to adopt

Many proposed cross-layer approaches
- Hints, pragmas, directives, HW-SW codeigns

- One-off approaches
  - Full-stack changes for one problem
  - Not scalable

- Programmer effort
  - Requires reasoning about HW
  - Becomes obsolete when HW changes
Our approach: A rich cross-layer abstraction

1. Generality: Enable a wide range of cross-layer approaches
2. Minimize programmer effort
3. Overhead

Approach: Associate program information with data
Associating program semantics with data

Challenges:
1. Changing program properties
2. Overhead
3. Programmer effort
The ATOM: An abstraction to express data semantics

Atom: **Dynamically associates flexible program semantics with memory regions of any granularity**

Valid/invalid at current execution point
The software interface: Three Atom operators

1. CREATE
2. MAP/UNMAP
3. ACTIVATE/DEACTIVATE
Using Atoms to express program semantics

\[ A = \text{malloc ( size )}; \]

Feature 1: Static attributes, dynamic mapping

\[ \text{Atom2} = \text{CreateAtom(“INT”, “Irregular”, ...}); \]

\[ \text{UnMapAtom( Atom1, A, size);} ; \]

\[ \text{MapAtom( Atom2, A, size );} \]

\[ \text{ActivateAtom(Atom2);} \]
Compile Time (CREATE)

A = malloc ( size );
Atom1 = CreateAtom("INT", "Regular", ...);
MapAtom( Atom1, A, size );
ActivateAtom(Atom1);
....
....
Atom2 = CreateAtom("INT", "Irregular", ...);
UnMapAtom( Atom1, A, size );
MapAtom( Atom2, A, size );
ActivateAtom(Atom2);
Load Time (CREATE)

Feature 2: Translator for programmability and portability
Run Time (MAP and ACTIVATE)

A = malloc (size);

Atom1 = CreateAtom("INT", "Regular", ...);

MapAtom(Atom1, A, size);

ActivateAtom(Atom1);

....

....

Atom2 = CreateAtom("INT", "Irregular", ...);

UnMapAtom(Atom1, A, size);

MapAtom(Atom2, A, size);

ActivateAtom(Atom2);
**Architectural support**

For a performance abstraction: low overhead is critical

No impact on functionality (hint-based):
*We can heavily optimize for low overhead*

<table>
<thead>
<tr>
<th>Atom</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- OS
- Caches
- Controller
- Prefetcher
- DRAM Cache
Expressing the semantics of a “tile”

Map each tile to an atom

ATOM_A = CREATE( 1 , {REGULAR,{Stride-X, Stride-Y}})
MAP(ATOM_A, Tile-Coordinates, 2D-Tile-Size)

Inserted by the compiler:
Informs cache and prefetching policies

Attributes: Tile size (implicit), Access pattern, Reuse
Leveraging expressed semantics

Last-level cache miss:

Software

Hardware

Matrix Multiply

Memory Address

Atom ID

If tile size > cache space:
Insert part of tile with high priority
Do not insert the rest

Calculate next address based on access stride

Atom ID

Attrib.

1

Reuse, Tile size

L3 Cache

Prefetcher

Atom ID

Attrib.

1

Access Pattern
Other cross-layer approaches enabled

- **Cache Management** [Pan+ SC’15, Sartor+ PACT ‘14, Dusser+ SC ’09, WANG+ PACT’02, Brock+ ISMM’13]
- **Data Placement in DRAM** [Liu+ PACT’12, Liu+ TACO’14, Ding+ MICRO’14, Bheda+ MEMSYS’16]
- **Data Compression** [Vijaykumar+ ISCA ’15, Pekhimenko+ PACT’12]
- **Prefetching** [Volos+ MICRO’14, Fuchs+ MICRO’14, Ebrahimi+ HPCA’09, Somogyi+ ISCA’09]
- **Approximation** [Sampson+ MICRO’13]
- **DRAM Cache Management** [Yu+, MICRO ’17, Olson+ NAS’18]
- **NVM Management** [Agarwal+ ASPLOS’15, Dulloor+ EuroSys ’16, Yoon+ ICCD’12]
- **NUCA/NUMA Optimizations** [Mukarra+ ASPLOS ’16, Dashti+ ASPLOS’13, Wang+ PACT’13]
Evaluating tradeoffs in a metadata management system

- Full-system open-source implementation of Expressive Memory on an FPGA
  - RISC-V cores
  - Software libraries + hardware support + OS support

- Evaluating feasibility
  - Area overhead (22nm): <0.5%
  - Memory overhead: 0.2%
  - Static power overhead: <1%
Key architectural components

Atom Address Map

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Atom ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Address

Atom Lookside Buffer (ALB)

Caches

Memory Controller

Prefetcher

DRAM Cache

...
Performance analysis: ALB size

![Graph showing normalized performance for different ALB sizes across various operations such as 3D Array, Matrix Multiply, Linked List, Streaming, and Random Access. The graph illustrates the impact of ALB size on performance improvements compared to baseline and no ALB.]
Performance analysis: Effect of contention

<table>
<thead>
<tr>
<th>3D Array</th>
<th>Matrix Multiply</th>
<th>Linked List</th>
<th>Streaming</th>
<th>Random Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1 client</td>
<td>2 clients</td>
<td>Baseline</td>
<td>1 client</td>
</tr>
</tbody>
</table>

Normalized Performance (%)
Performance analysis: the TLB effect

- 3D Array
- Matrix Multiply
- Linked List
- Streaming
- Random Access

Normalized Performance (%)

- Baseline
- Utilizing the TLB
- Skipping the TLB
Talk outline

1. Expressive Memory [ISCA 2018]
   - Analysis of a Rich Hardware-Software Interface [Ongoing]

2. The Locality Descriptor [ISCA 2018]

3. Zorua [MICRO 2016]

4. Assist Warps [ISCA 2015]
A rich cross-layer abstraction for GPUs

Key Challenge: How to express hierarchical parallelism

Structured hierarchy of parallelism in GPU programming models
Data locality is critical to GPU performance

Two forms of data locality:

Reusable-based locality (cache locality)

NUMA locality
Data locality is critical to GPU performance

Two forms of data locality:

- Reuse-based locality (cache locality)
- NUMA locality
The GPU execution and programming models are designed to explicitly express parallelism...

But there is no explicit way to express data locality

Exploiting data locality in GPUs is a challenging and elusive feat
A case study in leveraging data locality: Histo

Data Structure A

CTAs along the Y dim share the same data

Type of data locality: Inter-CTA
Leveraging cache locality

CTA scheduling is required to leverage inter-CTA cache locality

CTA scheduling is insufficient: we also need other techniques
Leveraging **NUMA locality**

Exploiting NUMA locality requires both **CTA scheduling** and data placement
Today, leveraging data locality is challenging

As a programmer:
- **No easy access** to architectural techniques — CTA scheduling, cache management, data placement, etc.
- Even when using work-arounds, optimization is **tedious and not portable**

As the architect:
- **Key program semantics** are not available to the hardware

*Where to place data?*

*Which CTAs to schedule together?*
To make things worse:

- There are many different locality types: Inter-CTA, inter-warp, intra-thread, ...

- Each type requires a different set of architectural techniques:
  - Inter-CTA locality requires CTA scheduling + prefetching
  - Intra-thread locality requires cache management
  - ...

48
The Locality Descriptor

A hardware-software abstraction to express and exploit data locality
Designing the Locality Descriptor

```
LocalityDescriptor ldesc(X; Y, Z);
```
An Overview: The components of the Locality Descriptor

1) **Data Structure**
   Basis of the abstraction

LocalityDescriptor `ldesc(A, len, INTER-THREAD, tile, loc);`

2) **Locality Type**
   Determines optimizations used
Driving underlying architectural techniques

**Locality Type?**

- **INTER_THREAD**
  - Determined based on more information

- **NO_REUSE**
  - Cache Bypassing
  - CTA Scheduling (if NUMA)
  - Memory Placement (if NUMA)

- **INTRA_THREAD**
  - CTA Scheduling
  - Cache Soft Pinning
  - Memory Placement (if NUMA)
An Overview: The components of the Locality Descriptor

LocalityDescriptor ldesc(A, len, INTER-THREAD, tile, loc);

1) Data Structure
   Basis of the abstraction

2) Locality Type
   Determines optimizations used

3) Tile Semantics
   Which threads share data?
   Thread-data mapping
Expressing locality with Tiles

Idea: Partition the data structure and compute grid into tiles

Basic unit of locality:
1) Data Tile
2) Compute Tile
3) Mapping between them

```
tile((X_tile, Y_len, 1),
     (1, GridSize.y, 1),
     (1, 0, 0));
```
An Overview: The components of the Locality Descriptor

1) Data Structure
   Basis of the abstraction

LocalityDescriptor ldesc(A, len, INTER-THREAD, tile, loc);

2) Locality Type
   Determines optimizations used

3) Tile Semantics
   Which threads share data?
   Thread-data mapping

4) Locality Semantics
   Access Pattern, Sharing Type
Leveraging the Locality Descriptor

LocalityDescriptor ldesc(A, INTER-THREAD, tile, loc);

Architectural techniques:
1) CTA Scheduling
2) Prefetching
3) Data Placement
CTA Scheduling

Data Structure A

CTA Compute Grid

Cluster 0
Cluster 1
Cluster 2
Cluster 3

Core 0
Core 1
Core 2
Core 3

L1D
L1D
L1D
L1D
Data Placement + CTA Scheduling

Data Structure A

Cluster 0
Cluster 1
Cluster 2
Cluster 3

Cluster Queues

SM 0
SM 1
SM 2
SM 3

L1D
L1D
L1D
L1D

Memory
Memory
Memory
Memory

NUMA Zone 0
NUMA Zone 1
NUMA Zone 2
NUMA Zone 3
Locality descriptors are an effective means to leverage cache locality

<table>
<thead>
<tr>
<th>SK</th>
<th>DT</th>
<th>HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTER-THREAD (COACCESSSED)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D2D</th>
<th>C2D</th>
<th>SPMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTER-THREAD (NEARBY)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LIB</th>
<th>LMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRA-THREAD</td>
<td></td>
</tr>
</tbody>
</table>

Geomean

Different locality types require different optimizations
A single optimization is often insufficient

<table>
<thead>
<tr>
<th>1.5</th>
<th>26.6%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

59
Performance Impact: Leveraging NUMA Locality

- Baseline
- FirstTouch-Distrib
- LDesc-Placement
- LDesc

53.7% (up to 2.8x)

Geomean
Talk outline

1. Expressive Memory [ISCA 2018]
   - Analysis of a Rich Hardware-Software Interface [Ongoing]

2. The Locality Descriptor [ISCA 2018]

3. Zorua [MICRO 2016]

4. Assist Warps [ISCA 2015]
On-Chip Resources in GPUs

- Compute Units
- Register File
- Scratchpad Memory
- Thread Slots
Every thread in a thread block needs to be allocated enough (worst-case) resources to execute and complete.
Abstraction of On-Chip Resources

Programmer/Software

Tight coupling between resource specification and allocation

Thread Slots

Register File

Scratchpad Memory
Key Issues

1. Static underutilization
2. Dynamic underutilization
1. Static Underutilization

<#Threads,#Registers,Scratchpad(KB)>
To make things worse...

- Same problem exists for other on-chip resources
  - registers, scratchpad memory, thread blocks

- The programmer needs to get it right for all of them at the **same time**
Implication 1: Programming Ease

Requires programmer effort to avoid sub-optimal specifications

**MST (Minimum Spanning Tree)**
Implication 2: Performance Portability

Programs need to be retuned to fit different GPUs

Normalized Execution Time

Threads/Block

Maxwell
Kepler
Fermi

DCT (Discrete Cosine Transform)
Key Issues

1. Static Underutilization

2. Dynamic Underutilization
2. Dynamic Underutilization

Resource requirements of a thread vary throughout execution

```c
__global__ void CUDAkernel2DCT(float *dst, float *src, int I)
{
    int OffsThreadInRow = threadIdx.y * B + threadIdx.x;
    ...
    for(unsigned int i = 0; i < B; i++)
        bl_ptr[i * X] = src[i * I];
__syncthreads();
...
CUDAsubroutineInplaceDCTvector(...);

    for(unsigned int i = 0; i < B; i++)
        dst[i * I] = bl_ptr[i * X];
}
```

16 regs

32 regs

16 regs
Our approach: Decouple specification from allocation

Virtual Resources

- Thread Slots

Hardware Resources

- Register File
  - Programmer/Software

Provide HW more flexibility:
Use a swap space in memory to provide extra resources
Zorua: Virtualization Strategy

Virtual Resources  |  Physical Resources

Fine-grained dynamic allocation provides resource efficiency
Zorua: Virtualization Strategy

Virtual Resources

Physical Resources

Swap Space (in the mem. hierarchy)

Careful oversubscription using a swap space provides flexibility in the amount of resources
Addressing Key Issues

- Static Underutilization
  - Provide an illusion of a flexible amount of resources

- Dynamic Underutilization
  - Enable dynamic allocation/deallocation of resources
Resulting performance curves

When there’s a mismatch:

1. Less effort and expertise to write high-performance GPU code
2. Better performance for highly-optimized code

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>Threads/Block</th>
<th>Scratchpad/Block</th>
<th>MST</th>
<th>NQU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>256</td>
<td>9000</td>
<td>8000</td>
<td>29000</td>
</tr>
<tr>
<td>0.5</td>
<td>512</td>
<td>29000</td>
<td>8000</td>
<td>49000</td>
</tr>
</tbody>
</table>
Talk outline

1. Expressive Memory [ISCA 2018]
   - Analysis of a Rich Hardware-Software Interface [Ongoing]
2. The Locality Descriptor [ISCA 2018]
3. Zorua [MICRO 2016]
4. Assist Warps [ISCA 2015]
Motivation: Unutilized On-chip Memory

Similar trends for on-chip scratchpad memory
Motivation: Idle Pipelines

67% of cycles idle

35% of cycles idle
Idea: Assist Warps

- **Goal:** Leverage idle resources to perform useful work
- A new HW-SW abstraction: **Assist Warps**
  - A “helper-thread” abstraction in GPUs
  - Performs simultaneous light-weight execution of useful tasks
  - Software defines the task
  - Dynamically triggered/throttled based on availability of resources (in hardware)
- **Use Cases:** performance optimization, background tasks, system-level tasks
  - Flexible data compression, prefetching, interrupts, memorization, instrumentation, etc.
Use case: Data compression

CABA achieves performance close to that of designs with no overhead for compression
Conclusion

- Cross-layer approaches enable a wide space of performance optimizations
- A key requirement: a richer cross-layer abstraction
- Cross-layer interfaces can be architected to be:
  - Low overhead with minimal changes to existing interfaces
  - General, enabling many hardware-software mechanisms
- We demonstrated HW-SW cooperative mechanisms in CPUs and GPUs that:
  - Enable more intelligent and efficient architectural techniques
  - Reduce effort required to write high performance code
  - Make performance portable across different processor generations
Enhancing Programmability, Portability, and Performance with Rich Cross-Layer Abstractions

Nandita Vijaykumar
Thesis Oral
October 11th, 2019

Committee
Co-chairs: Onur Mutlu, Phillip B. Gibbons
James C. Hoe
Mattan Erez (UT Austin)
Wen-Mei Hwu (UIUC)