Improving Data-dependent Applications in GPUs

AmirAli Abdolrashidi
Introduction

- GPUs are ideal for massively parallel computations.
- Single-instruction, multiple-data (SIMD) paradigm
- Thread block (TB) → Task
Introduction

• Workloads are getting more complex

• Data dependency is common among kernels
Introduction

• Despite support for massive parallelism, GPUs have limited support for data-dependent parallelism.
Introduction

• Despite support for massive parallelism, GPUs have limited support for data-dependent parallelism.
Introduction

- With fine-grained data dependency support
  - More utilization
  - Speedup
Motivation

• Need for generic finer-grain data dependency management
  • More performance
  • More efficient
Our Goal

• Towards a fine-grained generalized dependency support

• Step 1
  • A single mega-kernel
  • Task-based API
    • Dependency graph provided by user
  • Dependency-aware TB scheduler
    • Handles dependencies within kernel
Our Goal

• User must alter code/provide graph
  • User must know the dependencies

• Step 2
  • TBs as tasks
  • Minimize user intervention in extracting dependencies
    • Static JIT analysis
  • Kernel pre-launching
    • Hides kernel-launch overhead
  • TB scheduling
    • Inter-kernel dependency resolution
Our Goal

• **Step 3 (Ongoing work)**
  • We cannot detect non-static dependencies
    • Indirect memory access
    • Input-dependent branch
  • Indirect estimation of data dependency without profiling
  • Use pre-trained ML framework in the device
  • Handle Misprediction

```
threadIdx
  ↓
i
  ↓
A[i]
  ↓
B[A[i]]
```

?
Introduction

• Main goals
  • Mitigate overheads, such as kernel launch, etc.
  • Reduce significant programming required by the user
  • More generalized hardware framework for dependency resolution
Step 1: Wireframe

Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs
Example: Wavefront Pattern

...until the application ends
Example

Global Barriers (Original)

for i = 1 to nWave:
- Kernel Launch
- Synchronize

Enormous host-side kernel-launch overhead!

Waiting on non-parent thread blocks
Example

CUDA Dynamic Parallelism (Nested)

RUN:
- Parent Kernel Launch
  - Synchronize

Parent Kernel:
for i = 1 to nWaves:
  - Child Kernel Launch
  - Synchronize
Example

CUDA Dynamic Parallelism (Nested)

**RUN:**
- Parent Kernel Launch
- Synchronize

Parent Kernel:
for i = 1 to nWaves:
  - Child Kernel Launch
  - Synchronize

No more host-side kernel launch
Device-side kernel launch still has significant overhead
NO multi-parent dependency support
Still NO general dependency support!
Wireframe Overview

### Programming Model

```c
#define parent1 dim3 (blockIdx.x-1, blockIdx.y, blockIdx.z);
#define parent2 dim3 (blockIdx.x, blockIdx.y-1, blockIdx.z);

void* DepLink() {
    if (blockIdx.x > 0)
        WF::AddDependency(parent1);
    if (blockIdx.y > 0)
        WF::AddDependency(parent2);
}

int main() {
    kernel<<<GridSize, BlockSize, DepLink>>>(0, args);
    _WF__ void kernel(args) {
        processWave();
    }
}
```

### Dependency Graph

- Global Memory
- Global Node Array
- Global Edge Array

- Pending Update Buffer
- Node Insertion Buffer

### DATS Hardware

- TB Scheduler
- Convert to CSR

- Local Node Array
- Local Edge Array
- Pending Update Buffer
- Node Insertion Buffer

- Global Node Array
- Global Edge Array
Dependency Graph

• Application → One mega-kernel!
  • User-provided dependencies

• Contains node information such as “Parent count” and “Level (from root)”

• Sent to the GPU’s global memory at kernel call
Node Renaming

• To minimize data level range in the buffers
Dependency-Aware TB Scheduler (DATS)

• Thread block scheduler
  • Issues the relevant thread block at the time for execution based on the dependency graph

• Dependency Graph Buffer (DGB)
  • Cache data from global memory
  • Challenge: Efficient caching and data utilization
Dependency-Aware TB Scheduler (DATS)

Data stored in compressed sparse (CSR) format

• To reduce memory usage

• Thread blocks → Node Array

• Dependencies → Edge Array

• $O(n \log n)$ space complexity
DATS Overview

GLOBAL MEMORY

Translation of Global Edge Start to Local Edge Start

$$LES_i = (GES_i) \mod |EdgeArray|$$

DEPENDENCY GRAPH BUFFER (DGB)

(Circular buffer)
# Node State Table

<table>
<thead>
<tr>
<th>State</th>
<th>R</th>
<th>W</th>
<th>W</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Count</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Global Node ID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**States:**
- Wait
- Ready
- Processing
- Done

![Diagram with Node States and Edges](image)
Example: Child Node Execution

<table>
<thead>
<tr>
<th>State</th>
<th>1</th>
<th>D</th>
<th>3</th>
<th>R</th>
<th>3</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Count</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Node ID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

States:
- Wait
- Ready
- Processing
- Done
Example: Update Buffer Store

<table>
<thead>
<tr>
<th>State</th>
<th>D</th>
<th>P</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Count</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Global Node ID</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

States:
- Wait
- Ready
- Processing
- Done

Pending Update Buffer
Example: Invalidation...

<table>
<thead>
<tr>
<th>State</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Count</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Global Node ID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

States:
- Wait
- Ready
- Processing
- Done

Enough spaces to load to DGB
Example: ...Reloading data

Load complete!

<table>
<thead>
<tr>
<th>State</th>
<th>W</th>
<th>W</th>
<th>D</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Count</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Level</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Global Node ID</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

States:
- Wait
- Ready
- Processing
- Done

Load complete!
Example: Update Buffer Load

<table>
<thead>
<tr>
<th>State</th>
<th>R</th>
<th>R</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Count</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Level</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Global Node ID</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

States: Wait, Ready, Processing, Done

States:
- Wait
- Ready
- Processing
- Done

![Diagram with nodes and arrows showing states and transitions.](image-url)
Level Range

• Unbalanced execution may entail using the baseline TB scheduling policy (LRR).

Sample benchmark (HEAT2D) w/ LRR scheduler
Level Range

• Unbounded level range means:
  • Larger DGB is required
  • Limiting TB execution

Key challenge: Efficient scheduling
Level-bound Scheduling (LVL)

- Prioritizing lower-level thread blocks in the graph
- Minimizing the buffering operation
- Limiting the level range to avoid serialization
Performance

• Speedup across different number of nodes (same input size)

![Bar chart showing overall speedup (LVL) for different benchmarks and node counts (1K, 4K, 9K)].

- Works best for many smaller TBs
Step 2: BlockMaestro (BM)

Enabling Programmer-Transparent Task-based Execution in GPU Systems
Overview

• Wireframe requires user intervention
  • Heavy burden to establish dependencies
    • Difficult for more complex/irregular applications
  • Program rewrite required

• Wireframe needs converting application’s algorithm into tasks.

• Wireframe works on only one kernel
Motivation

• Finding dependencies among TBs **across kernels** with minimal burden
  • Different grid/block sizes
  • Minimal user intervention

• Increasing utilization by exploiting fine-grained dependencies
BlockMaestro Overview

CUDA Code (*.cu) → PTX Code Analysis → Inter-Kernel TB Dependency Graphs → GPU Memory

PTX SMs

K1
K2
K3
...

GPU Scheduler

SMs
Command Queue

- Kernels can be launched in a non-blocking fashion.

- Host must wait for blocking instructions.

- Adjacent kernels can be launched together

```
cudaMalloc(A)
cudaMemcpy(A, H2D)
K1<<<>>>(A)
cudaMalloc(B)
cudaMemcpy(B, H2D)
K2<<<>>>(A, B)
cudaMemcpy(D2H, B)
```
Inter-kernel Dependencies

• Dependencies can be detected through just-in-time (JIT) analysis
  • e.g., Read-after-write between the kernels

• Available info at kernel launch
  • Thread ID
  • Block ID
  • Base pointers
  • Kernel parameters
Static vs Non-static

• Only memory locations determined up to kernel-launch time can be used for dependency resolution.

```assembly
mov r1, ctaid.x
mul r1, r1, blk.x
add r1, r1, tid.x
add r1, r1, [A]
ld.global r2, [r1]

mov r1, [B]
ld.global r2, [r1]
add r2, r2, [A]
ld.global r3, [r2]
...```

...
Static vs Non-static

Static

Load

Address

Add

&A

Add

tid

Mul

blk

cTAid

Non-static

Load

Address

Add

&A

Load

Data

&A

&b

Unknown before runtime!
Hardware Support

- Dependency List Buffer
  - Child kernel TB IDs for parent kernel TBs

- Parent counter
  - # of unfinished parents for child kernel TBs
Example:
Enforcing Dependencies

- K1 and K2 are loaded in the GPU
- K1 TBs are scheduled and start executing
Example: Enforcing Dependencies

• K1 TBs finish
  • The rest of K1 TBs are loaded

• Parent count of finished TBs decrements
  • Their child TBs from K2 can now begin execution
Example: Enforcing Dependencies

- All TBs from K1 have finished
  - K1 is marked as finished.
- K2 becomes the producer kernel for the next kernel to be loaded.
- All parent counts are invalidated
  - Other TBs from K2 can now run.
Example: Enforcing Dependencies

- K3 is loaded as the new consuming kernel
- K3 dependency list is read from memory and loaded on TB scheduler
Performance

Works best for many smaller TBs
Step 3: SEER *(Ongoing work)*

Estimating Runtime Data Dependencies in GPU Applications
Overview

• BM only handles static data dependencies

• Uncertain fine-grained dependencies in non-static cases
  • Indirect memory access
  • Input-dependent data flow

• Possible to speculate non-static dependencies?
  • Ongoing work...
Example: Breadth-First Search (BFS)

```c
__global__ void Kernel(...)
{
    int tid = blockIdx.x * MAX_THREADS_PER_BLOCK + threadIdx.x;
    if( tid < no_of_nodes && g_graph_mask[tid])
    {
        g_graph_mask[tid] = false;
        for(int i = g_graph_nodes[tid].starting
            i < g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting);
            i++)
        {
            int id = g_graph_edges[i];
            if(!g_graph_visited[id])
            {
                g_cost[id] = g_cost[tid] + 1;
                g_updating_graph_mask[id] = true;
            }
        }
    }
}
```
Kernel Dependency Tree

• Static dependencies can be known at kernel-launch
  • e.g. tid

• Non-static dependencies are only known at run-time

• Correlation could exist!
  • tid → id?
Future Challenges

• How to represent the data?
• ML network?
• Input-dependent branches?
• Error in dependency resolution?