Scalable Critical Path Analysis for Hybrid MPI-CUDA Applications

The Fourth International Workshop on Accelerators and Hybrid Exascale Systems, May 19th 2014

Felix Schmitt, Robert Dietrich, Guido Juckeland
Outline

1. Motivation
2. CUDA Dependency Patterns
3. MPI-CUDA Critical Path Analysis
4. Use Cases
5. Outlook and Conclusion
Motivation

CUDA Dependency Patterns

MPI-CUDA Critical Path Analysis

Use Cases

Outlook and Conclusion
Motivation

- CUDA established for using general-purpose graphics-processing units in HPC [1]
- Increasing complexity of hybrid HPC programs requires sophisticated performance-analysis tools
- Problem: no current tool for automated analysis of execution dependencies in MPI-CUDA programs
  - Scalasca: scalable MPI critical-path analysis
  - HPCToolkit: MPI-CUDA profiling, no intra-device dependencies
  - NVIDIA Visual Profiler: CUDA optimization guidance, no MPI
Goals

Guide the developer to optimization targets in hybrid MPI-CUDA programs

- Scalable critical-path analysis based on trace files
- Analyze host/device and device/device dependencies and inefficiencies
- Visualize analysis results in Vampir
- Order activities by their potential optimization influence
**Preliminaries: Wait-State Analysis**

- **Event Stream**: stream of ordered events, e.g. MPI process, CUDA stream
- **Wait State**: time period at which an event stream is blocked [2], result of inter-stream dependencies and load imbalances
- **Blame (HPCToolkit) or cost of idleness (Scalasca)**: attributed to the cause of a wait state

**Examples for MPI Wait-States**

![Diagram showing examples of MPI Wait-States]

- A: Late receiver
- B: Late sender
- C: Load imbalance at barrier

Time

Process 1

- MPI_Send
- A: Late receiver

Process 2

- MPI_Recv
- B: Late sender

Process 3

- MPI_Send
- MPI_Recv
- MPI_Barrier
- C: Load imbalance at barrier

4/19
Preliminaries: Critical Path

Event Dependency Graph (EDG): directed acyclic graph

- Nodes are the events of parallel event streams
- Edges model the *happens-before* relationship and are weighted with the duration between events [3]

EDG for simple MPI example

(MPI_Init, MPI_Send/Recv, MPI_Finalize)
Critical Path: [4]

- Longest path in an EDG without wait states
- Optimizing activities on this path *can* reduce execution time
- Optimizing other activities *can not* (directly)
CUDA Wait-State Analysis

- Create a dependency/wait-state model for CUDA
- Two activity kinds: host (API) and device (kernels, memcpys)

New categorization of CUDA Inefficiency Patterns:

- Blocking Synchronization
- Non-Blocking Synchronization
- Late Synchronization
- Inter-Stream Dependencies
Rule-Based Pattern Detection

**BlameKernelRule**
Identifies blocking synchronization that is delayed by device activities.
Motivation

CUDA Dependency Patterns

MPI-CUDA Critical Path Analysis

Use Cases

Outlook and Conclusion
Critical Sub-Paths

- Combine MPI and CUDA critical path analysis
- MPI critical path detected using Scalasca’s *parallel reverse replay* [5]
- Global CUDA critical path is dominated by MPI critical path
- Determine critical sub-paths to *efficiently* and *concurrently* compute CUDA critical paths using OpenMP
Visualization in Vampir

*Vampir* and *VampirServer* enable scalable visualization of hybrid applications, including timelines, profiles, message and data transfers and performance counters.
(A) Counter Overlay: blocking memory copy (implicit synchronization)
(B) Counter Timeline: the synchronized kernel is attributed blame
(C) Counter Timeline: blocking synchronization is marked as waiting time
Activity Optimization Order

- Goal: Rank activity types by their potential influence

- Create an optimization order for activity types, add
  - normalized fraction of total critical-path duration (direct runtime impact)
  - normalized fraction of total blame (load-balancing impact)

  Highest-rated activities are best optimization candidates
Motivation

CUDA Dependency Patterns

MPI-CUDA Critical Path Analysis

Use Cases

Outlook and Conclusion
Correctness: Jacobi Method

MPI+CUDA application (two processes, one CUDA stream each). Executes two kernels in each iteration.

<table>
<thead>
<tr>
<th>Timeline</th>
<th>0.2295 s</th>
<th>0.2300 s</th>
<th>0.2305 s</th>
<th>0.2310 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA[0:2] 0:1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA[1:2] 1:1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10% work offloaded to GPU

90% work offloaded to GPU

Section of a trace in Vampir with two kernels: `jacobi_kernel` and `copy_kernel`.
Analysis result in Vampir’s performance radar (timeline overlay): CUDA kernels become critical activities (red) for high GPU offloading ratio due to blocking synchronization.
### Correctness: Jacobi Method (3)

<table>
<thead>
<tr>
<th>Activity</th>
<th>Critical Path [%]</th>
<th>Blame [%]</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jacobi_kernel</code></td>
<td>40.69</td>
<td>35.34</td>
<td>0.7603</td>
</tr>
<tr>
<td><code>cuMemcpyDtoH_v2</code></td>
<td>30.10</td>
<td>5.6</td>
<td>0.3570</td>
</tr>
<tr>
<td><code>MPI_Barrier</code></td>
<td>~0</td>
<td>35.62</td>
<td>0.3562</td>
</tr>
<tr>
<td><code>copy_kernel</code></td>
<td>5.04</td>
<td>9.59</td>
<td>0.1463</td>
</tr>
<tr>
<td><code>MPI_Allreduce</code></td>
<td>~0</td>
<td>12.78</td>
<td>0.1278</td>
</tr>
<tr>
<td><code>cuMemcpyHtoD_v2</code></td>
<td>10.15</td>
<td>0.0</td>
<td>0.1015</td>
</tr>
</tbody>
</table>

Activity optimization order for 90% work offloaded to the GPU.
Scalability: HPL CUDA

Scalability of HPL CUDA version and analysis¹. Combining MPI parallel replay and CUDA dependency analysis still scales with the MPI operations of the input trace.

¹ 1 MPI process/node, NVIDIA K20X GPUs
Motivation

CUDA Dependency Patterns

MPI-CUDA Critical Path Analysis

Use Cases

Outlook and Conclusion
Contributions:

- Comprehensive dependency model for CUDA activities
- Scalable tool for critical-path analysis of MPI-CUDA traces
- Identifies waiting time and the causing activities
- Visualization of all metrics in Vampir
- Generates a list of optimization targets, ordered by potential influence
Future Work

- Extend support to applications including OpenMP, CUDA and MPI (prototype available)
- Evaluate usage of hardware performance counters during optimization guidance
  → Which activities are easier to optimize?
- General CPU functions missing in this implementation (added in prototype)

Thank you for your attention!
Questions?
References


   Using cause-effect analysis to understand the performance of distributed programs.

   Time, clocks, and the ordering of events in a distributed system.

   Critical path analysis for the execution of parallel and distributed programs.

   Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications.