Performance Profiling with Omniperf

Cole Ramos and Ian Bogle

HIP Training Series
OLCF
16 Oct 2023
Cole Ramos

Software Development Engineer out of AMD’s Research and Advanced Development group.

Omniperf project lead with prior experience building software at Microsoft and startups.

Ian Bogle

MTS Software Development Engineer – AMD’s HPC Software Solutions group

- LAMMPS
- Kokkos

PhD in Graph Algorithms in HPC Contexts at Rensselaer Polytechnic Institute

- Algorithmic Design & Analysis
- Distributed Computing
- Kokkos
Agenda

General Overview

1. Introduction
2. Methodology
3. Implementation
   • Profile Mode
   • Analyze Mode
4. Roofline Analysis
5. Basic Examples

Guided Exercises

1. Launch Parameters
2. LDS Occupancy Limiter
3. VGPR Occupancy Limiter
4. Strided Data Access Pattern
5. Algorithmic Optimizations
Omnitrace
A tool for the entire execution of an application
  • Generated high-level comprehensive trace of your application
  • Identified particular kernels through a critical trace

Omniperf
A kernel-level performance analysis tool
  • Extensive insight into the execution of individual kernels
  • Examine kernel performance organized by logical IP blocks
**Omnitrace**

A tool for the entire execution of an application

- Generated high-level comprehensive trace of your application
- Identified particular kernels through a critical trace

**Omniperf**

A kernel-level performance analysis tool

- Extensive insight into the execution of individual kernels
- Examine kernel performance organized by logical IP blocks

**Omniperf**

Repo + Docs
https://github.com/AMDResearch/omniperf

**Dependencies**

- ROCm (>=5.2)
- Python (>=3.7)
- CMake (>=3.19)

**Support**

- All major Linux distributions
  - Ubuntu, SLES, RHEL
- Mi100, Mi200, Mi300 (in testing)
Client-side tool
Client-side tool

Modes

Modes change the fundamental behavior of the Omniperf command line tool.

1. Profile
   - Toggle Roofline
   - Filter performance counters

2. Analyze
   - Generate data tables directly in terminal
   - Launch a webpage to visualize

```
$ omniperf profile --help
usage:
  omniperf profile --name <workload_name> [profile options] [roofline options] -- <profile_cmd>

Examples:
  omniperf profile -- vcopy.all -- ./vcopy 1048576 256
  omniperf profile -- vcopy.GPU.TO -- b SV TCC -- ./vcopy 1048576 256
  omniperf profile -- vcopy.kernel -- k vecCopy -- ./vcopy 1048576 256
  omniperf profile -- vcopy_disp -- d 8 -- ./vcopy 1048576 256
  omniperf profile -- vcopy_roof -- roof-only -- ./vcopy 1048576 256
```
Client-side tool

Modes

Modes change the fundamental behavior of the Omniperf command line tool.

1. Profile
   - Toggle Roofline
   - Filter performance counters

2. Analyze
   - Generate data tables directly in terminal
   - Launch a webpage to visualize

```
columnos@sv-pdp-2:~$ omniperf analyze --help
usage:
omiperf analyze --path <workload.path> [analyze options]

Examples:
omiperf analyze -p workloads/vcopy/mi200/ --list-metrics gfx90a
omiperf analyze -p workloads/mixbench/mi200/ --filter-dispatch-lds 12 34 --decimal 3
omiperf analyze -p workloads/mixbench/mi200/ --gpu
```
“Modes” in Omniperf
Profile Mode

Features:
• Runtime Filtering
  --kernel, --ipblocks, --dispatch

- The `-k <kernel>` flag allows for kernel filtering, which is compatible with the current rocprof utility.
- The `-d <dispatch>` flag allows for dispatch ID filtering, which is compatible with the current rocprof utility.
- The `-b <ipblocks>` allows system profiling on one or more selected IP blocks to speed up the profiling process. One can gradually incorporate more IP blocks, without overwriting performance data acquired on other IP blocks.
Profile Mode

Features:

- Runtime Filtering
  --kernel, --ipblocks, --dispatch
- Standalone Roofline Analysis
  --roof-only

The above plots are saved as PDF output when the --roof-only option is used.
Profile Mode

Features:

- **Runtime Filtering**
  --kernel, --ipblocks, --dispatch
- **Standalone Roofline Analysis**
  --roof-only
- **No roofline analysis**
  --no-roof

--no-roof will skip the roofline microbenchmark and omit roofline from output
Analyze Mode

Features:

- List top kernels or view list of metrics
  --list-kernels, --list-metrics

Output from the --list-kernel and --list-metric options, showing top kernels and available metrics
Analyze Mode

Features:

- List top kernels or view list of metrics
  ```
  --list-kernels, --list-metrics
  ```
- Filter available kernels, dispatches, gpu-ids
  ```
  --kernel, --dispatch, --gpu-id
  ```

Filtered output from the `--kernel` option isolating kernel at index 0
Analyze Mode

**Features:**

- List top kernels or view list of metrics
  
  ```bash
  --list-kernels, --list-metrics
  ```

- Filter available kernels, dispatches, gpu-ids
  
  ```bash
  --kernel, --dispatch, --gpu-id
  ```

- Filter by metric id(s)
  
  ```bash
  --metric
  ```

Filtering output to isolate data table at index 5
Analyze Mode

Features:

- List top kernels or view list of metrics
  
  ```
  --list-kernels, --list-metrics
  ```

- Filter available kernels, dispatches, gpu-ids
  
  ```
  --kernel, --dispatch, --gpu-id
  ```

- Filter by metric id(s)
  
  ```
  --metric
  ```

- Change normalization unit, time unit, or decimal
  
  ```
  --normal-unit, --time-unit, --decimal
  ```
Analyze Mode (cont.)

Features:
- Baseline Analysis
  --path <workload1_path> --path <workload2_path>
Analyze Mode (cont.)

Features:

- Baseline Analysis
  --path <workload1_path> --path <workload2_path>

- Launch a standalone HTML page from terminal
  --gui <port>

Terminal output from the --gui option with full port forwarding info

The above webpage is launched when the --gui option is used
Analyze Mode - Methodology

- Everything we do in analyze mode is built on top of "yml config files"
- These dynamic config files are flexible and can be easily customized
- The configs are used in both CLI and standalone GUI
Omniperf Metrics
High level Metrics

• System Info

Detailed system info for each app is collected by default
High level Metrics

- System Info
- System Speed-of-Light

Calls attention to high level performance stats to preview overall application performance
High level Metrics

- System Info
- System Speed-of-Light
- Kernel Stats

Preview performance of top N kernels and individual kernel invocations (dispatches)
High level Metrics

- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis

Illustrate data movement and performance on key components of target architecture
High level Metrics

- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis
- Roofline Analysis

Derived Empirical Roofline analysis broken into two major instruction mixes. Showing application performance relative to measured maximum achievable performance
Roofline Analysis Methodology
Background – What is roofline?

- Attainable FLOPs/s =
  - $\min\left\{ \frac{\text{Peak FLOPs/s}}{\text{AI} \times \text{Peak GB/s}} \right\}$

- Machine Balance:
  - Where $AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}}$

- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
  - Bandwidth Bound
  - Poor Performance
Background – What is roofline?

- Attainable FLOPs/s = 
  - \( \min \left\{ \frac{\text{Peak FLOPs/s}}{\text{AI} \times \text{Peak GB/s}} \right\} \)

- Machine Balance:
  - Where \( \text{AI} = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \)

- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
  - Bandwidth Bound
  - Poor Performance
Introduction - Empirical Hierarchical Roofline on AMD Instinct™ MI250X GPU

Peak MFMA GFLOP/sec
Peak VALU GFLOP/sec
Peak vL1D BW
Peak LDS BW
Peak HBM BW
Peak L2 BW
Workload Perf: (GFLOP/sec, AI)
Empirical Hierarchical Roofline on MI200 - Arithmetic

\[
\text{Total FLOP} = 64 \times (\text{SQ}_{\text{INSTS VALU ADD F16}} + \text{SQ}_{\text{INSTS VALU MUL F16}} + \text{SQ}_{\text{INSTS VALU TRANS F16}} + 2 \times \text{SQ}_{\text{INSTS VALU FMA F16}}) \\
+ 64 \times (\text{SQ}_{\text{INSTS VALU ADD F32}} + \text{SQ}_{\text{INSTS VALU MUL F32}} + \text{SQ}_{\text{INSTS VALU TRANS F32}} + 2 \times \text{SQ}_{\text{INSTS VALU FMA F32}}) \\
+ 64 \times (\text{SQ}_{\text{INSTS VALU ADD F64}} + \text{SQ}_{\text{INSTS VALU MUL F64}} + \text{SQ}_{\text{INSTS VALU TRANS F64}} + 2 \times \text{SQ}_{\text{INSTS VALU FMA F64}}) \\
+ 512 \times \text{SQ}_{\text{INSTS VALU MFMA MOPS F16}} \\
+ 512 \times \text{SQ}_{\text{INSTS VALU MFMA MOPS BF16}} \\
+ 512 \times \text{SQ}_{\text{INSTS VALU MFMA MOPS F32}} \\
+ 512 \times \text{SQ}_{\text{INSTS VALU MFMA MOPS F64}}
\]

\[
\text{Total IOP} = 64 \times (\text{SQ}_{\text{INSTS VALU INT32}} + \text{SQ}_{\text{INSTS VALU INT64}})
\]

\[
\text{LDS}_{\text{BW}} = 32 \times 4 \times (\text{SQ}_{\text{LDS IDX ACTIVE}} - \text{SQ}_{\text{LDS BANK CONFLICT}})
\]

\[
\text{L1D}_{\text{BW}} = 64 \times \text{TCP TOTAL CACHE ACCESES sum}
\]

\[
\text{L2}_{\text{BW}} = 64 \times \text{TCP TCC READ REQ sum} \\
+ 64 \times \text{TCP TCC WRITE_REQ sum} \\
+ 64 \times (\text{TCP TCC_ATOMIC WITH_RET_REQ sum} + \text{TCP TCC_ATOMIC WITHOUT_RET_REQ sum})
\]

\[
\text{HBM}_{\text{BW}} = 32 \times \text{TCC EA RDREQ 32B sum} + 64 \times (\text{TCC EA RDREQ sum} - \text{TCC EA RDREQ 32B sum}) \\
+ 32 \times (\text{TCC EA WRREQ sum} - \text{TCC EA WRREQ 64B sum}) + 64 \times \text{TCC EA WRREQ 64B sum}
\]

\[
\begin{align*}
A_{\text{LDS}} &= \frac{\text{TOTAL FLOP}}{\text{LDS}_{\text{BW}}} \\
A_{\text{L1D}} &= \frac{\text{TOTAL FLOP}}{\text{L1D}_{\text{BW}}} \\
A_{\text{L2}} &= \frac{\text{TOTAL FLOP}}{\text{L2}_{\text{BW}}} \\
A_{\text{HBM}} &= \frac{\text{TOTAL FLOP}}{\text{HBM}_{\text{BW}}}
\end{align*}
\]

*All calculations are subject to change without notice
# Low level Metrics

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Processor (CPC/CPF)</td>
<td>Packet processor data</td>
</tr>
<tr>
<td>Shader Processor Input (SPI)</td>
<td>Connecting packet processor and CUs</td>
</tr>
<tr>
<td>Wavefront Stats</td>
<td>Kernel launch stats</td>
</tr>
<tr>
<td>Compute Unit – Instruction Mix</td>
<td>Breakdown of instructions issued</td>
</tr>
<tr>
<td>Compute Unit – Compute Pipeline</td>
<td></td>
</tr>
<tr>
<td>Texture Addressor &amp; Texture Data (TA/TD)</td>
<td>Fetch &amp; receive reqs for lookup in vL1D RAM</td>
</tr>
<tr>
<td>Local Data Share (LDS)</td>
<td></td>
</tr>
<tr>
<td>Instruction Cache</td>
<td></td>
</tr>
<tr>
<td>Scalar L1 Data Cache</td>
<td>Cache level stats</td>
</tr>
<tr>
<td>Vector L1 Data Cache</td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td></td>
</tr>
<tr>
<td>L2 Cache (per channel)</td>
<td></td>
</tr>
</tbody>
</table>
Agenda

General Overview

1. Introduction
2. Methodology
3. Implementation
   • Profile Mode
   • Analyze Mode
4. Roofline Analysis
5. Basic Examples

Guided Exercises

1. Launch Parameters
2. LDS Occupancy Limiter
3. VGPR Occupancy Limiter
4. Strided Data Access Pattern
5. Algorithmic Optimizations
Guided Exercises: Logistics/Preamble

• To accommodate the virtual setting and attendees with varied access to Omniperf:
  • I’ll read through the slides without waiting for everyone to finish working through each exercise
  • If you have access to a system with Omniperf, clone the repo and start working through the exercises:
    • `git clone https://github.com/OLCF/hip-training-series`, we’ll be working in the `Lecture5/OmniperfExamples` subdirectory.
    • The READMEs contain all of what I’m saying and include platform-specific instructions for this training in the top-level directory
  • Cole Ramos and Bob Robey will monitor and answer questions while I’m talking

• We have used a publicly available release candidate of Omniperf to generate output for these slides:
  • `https://github.com/AMDResearch/omniperf/releases/tag/v1.1.0-PR1`
  • Behavior may differ if using a different version of Omniperf (e.g. 1.0.10)
  • Generally, building stable releases is the best practice

• The numbers shown in the READMEs and these slides were generated using MI210 accelerators

• Implementations in these exercises are not fully-optimized kernels
Guided Exercises: Representative Optimization Tasks

- The Exercises are roughly in order of ease of development effort and performance impact:
  - Exercise 1: Verify Reasonable Launch Parameters
  - Exercise 2: Attempt to Cache Data in Shared Memory
  - Exercise 3: Determining a Source of Unexpected Resource Usage
  - Exercise 4: Verifying Efficient Data Access Patterns
  - Exercise 5: Analyzing an Algorithmic Change

- The underlying code is kept simple to emphasize the optimization techniques

- These slides are intended as a “Cheat Sheet” starting point providing:
  - Omniperf commands to filter through output for common optimization concerns
  - Some optimization direction given certain Omniperf output
Guided Exercises: Optimizing a yAx Kernel

• We’ll be looking at a relatively simple kernel that solves the same problem in each exercise, yAx
  • yAx is a vector-matrix-vector product that can be implemented in serial as:

```cpp
double result = 0.0;
for (int i = 0; i < n; i++){
    double temp = 0.0;
    for (int j = 0; j < m; j++){
        temp += A[i*m + j] * x[j];
    }
    result += y[i] * temp;
}
```

• Where:
  • A is a 1-D array of size n*m
  • x is an array of size m
  • y is an array of size n
Exercise 1: First Things First, Generate a Roofline

- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
  - `omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
    - The files will appear in the `./workloads/problem_roof_only/mi200` folder.
    - `--roof-only` generates PDF roofline plots, and does **not** generate any non-roofline profiling data
    - `--kernel-names` generates a PDF showing which kernel names correspond to which icons in the roofline

- Rooflines are a useful tool in determining which kernels are good optimization targets
  - They are only one perspective of performance: runtime of the kernel cannot be inferred from the roofline

- Generated PDF roofline plots can have overlapping data points but should still be instructive
  - There are fixes to this, but they may be difficult to setup for different cluster installations
  - Generating the PDF plots from the command line interface should always work

- Complete sets of Roofline plots and commands can be found in the READMEs for each exercise
Exercise 1: Problem Roofline Plots

FP32 Roofline Plot

```
Performance (GFLOP/sec) vs Arithmetic Intensity (FLOPs/Byte)
```

- HBM-FP32
- L2-FP32
- L1-FP32
- LDS-FP32
- Peak VALU-FP32
- Peak MFMA-FP32

**kernel legend**

- Very poor performance!

FP16/INT8 Roofline Plot

```
Performance (GFLOP/sec) vs Arithmetic Intensity (FLOPs/Byte)
```

**Note:** The L2 data point is hidden behind the HBM data point.
Exercise 1: Prep to use Omniperf to Find Kernel Launch Parameters

- Launch parameters are given at the time of the kernel launch, as in lines 49 and 54:
  - `yax<<<grid,block>>>(y,A,x,n,m,result);`
  - Where grid and block are the kernel yax’s launch parameters
  - In problem, grid = (4,1,1), and block = (64,1,1)
  - In solution, grid = (2048,1,1), and block = (64,1,1)

- Sometimes the launch parameters for a given kernel can be obfuscated

- Omniperf can easily show launch parameter information regardless of the code
  - You just need the dispatch ID

- To generate profiling data, use the commands:
  - `omniperf profile -n problem --no-roof -- ./problem.exe`
  - `omniperf profile -n solution --no-roof -- ./solution.exe`
    - `--no-roof` saves time by not generating roofline data – profile commands can take a while

- **Real benchmarks can take prohibitively long to profile** – use smaller representative problems if possible
Exercise 1: CLI Omniperf Comparisons are Easy

omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 7.1.0 7.1.1 7.1.2

Analyze

-------

0. Top Stat

<table>
<thead>
<tr>
<th>KernelName</th>
<th>Count</th>
<th>Count</th>
<th>Sum(ns)</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>yax(double*, double*, double*, int, int, double*)</td>
<td>1.00</td>
<td>1.0 (0.0%)</td>
<td>754934306.50</td>
<td>69702016.5 (-90.77%)</td>
<td>754934306.50</td>
<td>69702016.5 (-90.77%)</td>
<td>754934306.50</td>
<td>69702016.5 (-90.77%)</td>
<td>100.00</td>
<td>100.0 (0.0%)</td>
</tr>
</tbody>
</table>

7. Wavefront
7.1 Wavefront Launch Stats

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Avg</th>
<th>Avg</th>
<th>Min</th>
<th>Min</th>
<th>Max</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.0</td>
<td>Grid Size</td>
<td>256.00</td>
<td>131072.0 (51100.0%)</td>
<td>256.00</td>
<td>131072.0 (51100.0%)</td>
<td>256.00</td>
<td>131072.0 (51100.0%)</td>
<td>Work Items</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Workgroup Size</td>
<td>64.00</td>
<td>64.0 (0.0%)</td>
<td>64.00</td>
<td>64.0 (0.0%)</td>
<td>64.00</td>
<td>64.0 (0.0%)</td>
<td>Work Items</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Total Wavefronts</td>
<td>4.00</td>
<td>2048.0 (51100.0%)</td>
<td>4.00</td>
<td>2048.0 (51100.0%)</td>
<td>4.00</td>
<td>2048.0 (51100.0%)</td>
<td>Wavefronts</td>
</tr>
</tbody>
</table>

10.8x speedup

In general, it is difficult to pre-determine optimal launch bounds, so some experimentation is likely necessary

Increased launched wavefronts, which increases Grid Size

These slides always put problem as the baseline, and solution as the comparative
Exercise 1: Comparing Problem and Solution Roofline Plots

Generally, moving up and to the right is good.
Exercise 1: It’s Easy to Check Launch Parameters with Omniperf

- Use this omniperf command to check launch parameters:
  - omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 7.1.0 7.1.1 7.1.2
    - Shows the launch parameters of the kernel with dispatch ID 1
    - --metric filters the output to only show these launch parameters

- Good launch parameters are essential to a performant GPU kernel
  - Determining which parameters give the best performance usually requires experimenting

- It can be difficult to track down where launch parameters are set in code

- Omniperf can easily show the launch parameters of a kernel
  - Need the dispatch ID or index given by --list-kernels
  - --list-kernels index can be passed to -k as in:
    - omniperf analyze -p workloads/problem/mi200 -k 0 --metric 7.1.0 7.1.1 7.1.2

- Note:
  - These metric numbers are for Omniperf 1.0.10
Exercise 2: Diagnosing a Shared Memory Occupancy Limiter

• Using LDS (Local Data Store – Shared Memory) to cache re-used data can be an effective optimization strategy

• Using too much LDS can restrict occupancy however, and reduce performance

• Line 12 in problem.cpp shows the allocation of LDS:
  • __shared__ double tmp[fully_allocate_lds];

• There are two solutions:
  • solution-no-lds removes the LDS allocation, and thus the occupancy limiter
  • solution reduces the size of the LDS allocation, removes occupancy limiter, and is faster than solution-no-lds
    • This is the solution used to generate the Omniperf output in the next slide

• Omniperf makes it easy to determine if LDS allocations restrict occupancy, as before profile with:
  • omniperf profile -n problem --no-roof -- ./problem.exe
  • omniperf profile -n solution --no-roof -- ./solution.exe
### Exercise 2: LDS Occupancy Limiter – Relevant Omniperf Output

```bash
omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 2.1.26 6.2.7
```

#### Analyze

0. Top Stat

<table>
<thead>
<tr>
<th>KernelName</th>
<th>Count</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>yax(double*, double*, double*, int, int, int, double*)</td>
<td>1.00</td>
<td>175427205.00</td>
<td>50366185.0</td>
<td>175427205.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

#### 2. System Speed-of-Light

2.1 Speed-of-Light

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
<th>Peak</th>
<th>PoP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.26</td>
<td>Wave Occupancy</td>
<td>102.70</td>
<td>Wavefronts</td>
<td>3328.00</td>
<td>3.09</td>
</tr>
</tbody>
</table>

+ ~11% Occupancy (overall)

Sharp decrease in SPI stat

#### 6. Shader Processor Input (SPI)

6.2 SPI Resource Allocation

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Avg</th>
<th>Unit</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.7</td>
<td>Insufficient CU LDS</td>
<td>6015745446.00</td>
<td>Cu</td>
<td>6015745446.00</td>
</tr>
</tbody>
</table>

3.4x speedup
Exercise 2: Use SPI Stats to Determine if LDS Limits Occupancy

- Occupancy limiters can negatively impact performance

- Workgroup manager (SPI) stats in Omniperf indicate whether a kernel resource limits occupancy

- You can get the SPI stat for LDS for a single kernel with:
  - omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 2.1.26 6.2.7

Note:

- In current Omniperf release 1.0.10, the SPI “insufficient resource” stats are a count of cycles, meaning:
  - Large numbers (on the order of over 1 million) are expected if a field is not zero
  - The magnitude of these fields does not necessarily indicate how severely occupancy is impacted
  - If two fields are nonzero, the larger number indicates that resource is limiting occupancy more

- In a coming release, these “insufficient resource” fields are changing to percentages:
  - Large numbers will no longer be expected, but the other points will still hold
Exercise 3: Diagnosing a Register Occupancy Limiter

- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - In this case an assert on line 15 causes occupancy to be limited by register usage
  - The solution simply removes the assert

- The types of registers on AMD GPUs are:
  
  - **VGPRs (Vector General Purpose Registers):** registers that can hold distinct values for each thread in the wavefront
  
  - **SGPRs (Scalar General Purpose Registers):** uniform across a wavefront. If possible, using these is preferable
  
  - **AGPRs (Accumulation vector General Purpose Registers):** special-purpose registers for MFMA (Matrix Fused Multiply-Add) operations, or low-cost register spills

- Using too many of one of these register types can impact occupancy and negatively impact performance

- We use the same profile commands to get the profiling data:
  
  - `omniperf profile -n problem --no-roof -- ./problem.exe`
  
  - `omniperf profile -n solution --no-roof -- ./solution.exe`
Exercise 3: Register Occupancy Limiter – Relevant Omniperf Output

Omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 2.1.26 6.2.5 7.1.5 7.1.6 7.1.7

### 0. Top Stat

<table>
<thead>
<tr>
<th>KernelName</th>
<th>Count</th>
<th>Count</th>
<th>Sum(ns)</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>yax( double*, double*, double*, int, int, double*)</td>
<td>1.00</td>
<td>1.0 (0.0%)</td>
<td>76983902.00</td>
<td>69815871.0 (-9.31%)</td>
<td>76983902.00</td>
<td>69815871.0 (-9.31%)</td>
<td></td>
<td></td>
<td>100.00</td>
<td>100.0 (0.0%)</td>
</tr>
</tbody>
</table>

**Minor speedup**

### 2. System Speed-of-Light

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Value</th>
<th>Value</th>
<th>Unit</th>
<th>Peak</th>
<th>Peak</th>
<th>Pop</th>
<th>Pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.26</td>
<td>Wave Occupancy</td>
<td>438.00</td>
<td>444.1 (1.39%)</td>
<td>Wavefronts</td>
<td>3328.00</td>
<td>3328.0 (0.0%)</td>
<td>13.16</td>
<td>13.34 (1.4%)</td>
</tr>
</tbody>
</table>

**Small increase in occupancy**

### 6. Shader Processor Input (SPI)

#### 6.2 SPI Resource Allocation

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Avg</th>
<th>Avg</th>
<th>Min</th>
<th>Min</th>
<th>Max</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.5</td>
<td>Insufficient SIMD VGPRs</td>
<td>13733460.00</td>
<td>0.0 (-100.0%)</td>
<td>13733460.00</td>
<td>0.0 (-100.0%)</td>
<td>13733460.00</td>
<td>0.0 (-100.0%) Simd</td>
<td></td>
</tr>
</tbody>
</table>

**Large decrease in SPI stat**

### 7. Wavefront

#### 7.1 Wavefront Launch Stats

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Avg</th>
<th>Avg</th>
<th>Min</th>
<th>Min</th>
<th>Max</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.5</td>
<td>VGPRs</td>
<td>92.00</td>
<td>32.0 (-65.22%)</td>
<td>92.00</td>
<td>32.0 (-65.22%)</td>
<td>92.00</td>
<td>32.0 (-65.22%) Registers</td>
<td></td>
</tr>
<tr>
<td>7.1.6</td>
<td>AGPRs</td>
<td>132.00</td>
<td>0.0 (-100.0%)</td>
<td>132.00</td>
<td>0.0 (-100.0%)</td>
<td>132.00</td>
<td>0.0 (-100.0%) Registers</td>
<td></td>
</tr>
<tr>
<td>7.1.7</td>
<td>SGPRs</td>
<td>48.00</td>
<td>96.0 (100.0%)</td>
<td>48.00</td>
<td>96.0 (100.0%)</td>
<td>48.00</td>
<td>96.0 (100.0%) Registers</td>
<td></td>
</tr>
</tbody>
</table>

**Able to use:**
Fewer VGPRs,  
No AGPRS,  
More SGPRs
Exercise 3: Register Occupancy Limiter - Takeaways

• Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  • Asserts, and even excessive use of math functions in kernels can degrade performance

• In this case the occupancy limit was very minor, despite a large number in the SPI stat

• AGPR usage in the absence of MFMA (Matrix Fused Multiply Add) instructions can indicate degraded performance.
  • Spilling registers to AGPRs, due to running out of VGPRs

• To determine if any SPI “insufficient resource” stats are nonzero, you can do:
  • `omnipерf analyze -p workloads/problem/mi200 --dispatch 1 --metric 6.2`
    • Note: This will report more than just all “insufficient resource” fields
Exercise 4: Data Access Patterns are Important to Performance

• The way in which threads access memory has a big impact on performance

• “Striding” in global memory has adverse effects on kernel performance, especially on GPUs.
  • “Strided data access patterns” lead to poor utilization of cache memory systems

• These access patterns can be difficult to spot in the code
  • They are valid methods of indexing data

• Using Omniperf can quickly show if a kernel's data access is adversarial to the caches
Exercise 4: What is a “Strided Data Access Pattern”?

Data that each thread accesses at each step requires striding through memory, which leads to sub-optimal memory system usage.
Exercise 4: Strided Data Access Patterns

Increasing the **locality** of data accesses of nearby threads allows for more efficient memory usage.

**Note:** This is the same computation as before, only data layout has changed.
Exercise 4: Using Omniperf to Diagnose a Strided Data Access Pattern

• This exercise’s setup makes it very easy to change the data access pattern
  • Generally, these optimizations can have nontrivial development overhead
  • Re-conceptualizing the data structure can be difficult

• All the solution does is re-work the indexing scheme to better use caches
  • No required change to underlying data, because all the values in y, A, and x are set to 1

• To get started run:
  • omniperf profile -n problem --no-roof -- ./problem.exe
  • omniperf profile -n solution --no-roof -- ./solution.exe
Exercise 4: Strided Data Access Pattern – Relevant Omniperf Output

omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 16.1 17.1

---

**0. Top Stat**

<table>
<thead>
<tr>
<th>KernelName</th>
<th>Count</th>
<th>Count (0.0%)</th>
<th>Sum(ns)</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
<th>Pct (0.0%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yax(double*, double*, double*, int, int, double*)</td>
<td>1.00</td>
<td>1.0 (0.0%)</td>
<td>69875592.00</td>
<td>12469690.5 (-82.15%)</td>
<td>69875592.00</td>
<td>12469690.5 (-82.15%)</td>
<td>69875592.00</td>
<td>12469690.5 (-82.15%)</td>
<td>100.00</td>
<td>100.0 (0.0%)</td>
</tr>
</tbody>
</table>

---

**16. Vector L1 Data Cache**

16.1 Speed of Light

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Value</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1.0</td>
<td>Buffer Coalescing</td>
<td>25.00</td>
<td>25.0 (0.0%)</td>
<td>Pct of peak</td>
</tr>
<tr>
<td>16.1.1</td>
<td>Cache Util</td>
<td>87.80</td>
<td>98.08 (11.7%)</td>
<td>Pct of peak</td>
</tr>
<tr>
<td>16.1.2</td>
<td>Cache BW</td>
<td>8.69</td>
<td>12.18 (40.19%)</td>
<td>Pct of peak</td>
</tr>
<tr>
<td>16.1.3</td>
<td>Cache Hit</td>
<td>0.00</td>
<td>49.98 (inf%)</td>
<td>Pct of peak</td>
</tr>
</tbody>
</table>

---

**17. L2 Cache**

17.1 Speed of Light

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Value</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1.0</td>
<td>L2 Util</td>
<td>98.74</td>
<td>98.39 (-0.36%)</td>
<td>Pct</td>
</tr>
<tr>
<td>17.1.1</td>
<td>Cache Hit</td>
<td>93.45</td>
<td>0.52 (-99.44%)</td>
<td>Pct</td>
</tr>
<tr>
<td>17.1.2</td>
<td>L2-EA Rd BW</td>
<td>125.69</td>
<td>688.98 (448.16%)</td>
<td>Gb/s</td>
</tr>
<tr>
<td>17.1.3</td>
<td>L2-EA Wr BW</td>
<td>0.00</td>
<td>0.0 (inf%)</td>
<td>Gb/s</td>
</tr>
</tbody>
</table>

---

5.6x speedup

+ ~50% in L1 hit

The solution better uses the L1, but our L2 hit rate has degraded, which points to a deficiency in our algorithm.

L2 Cache Hit decreases sharply, Read BW from HBM increases by ~5x
Exercise 4: Omniperf Speed-of-Light Cache Access Statistics

- This Omniperf command will show high-level details about L1 and L2 cache accesses:
  - omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 16.1 17.1

- Ensuring better data locality will generally provide better performance

- In this case, we start hitting in the L1 cache, rather than having to go out to L2 for everything

- **Note:** In a real code, optimizations of this type likely have much more development overhead
  - Need to change how the data structure is indexed everywhere
Exercise 5: Algorithmic Optimizations

• These types of optimizations are the most difficult to execute
  • Generally, it is difficult to determine if the runtime of one algorithm will be faster than another

• We start with the solution from last exercise as our problem
  • Speed-of-light cache statistics showed that we had ~0% hit rate in the L2, could it be better?

• Our initial algorithm is naïve in terms of parallelization:
  • Each thread computes the sum of a row

• Exposing more parallelism is possible and should get us more performance in this case
Exercise 5: Algorithmic Optimizations

In our current algorithm, each thread computes the sum of a single row
Exercise 5: Algorithmic Optimizations

Matrix A

In a more efficient implementation, wavefronts have multiple threads sum up the rows in parallel, using shared memory to reduce partial sums.

Note: The original data layout allows the wavefronts to avoid striding memory.
Exercise 5: Using Omniperf to Evaluate an Algorithmic Optimization

- The strided data access pattern issue is everywhere
  - This solution gets about 2x faster when the data layout is switched to optimize locality

- Though the solution shows a 29x speedup from the problem, cache speed-of-light stats aren’t convincing
  - The rooflines for these problems do not tell the full performance story either

- Running the solution shows it is much faster, but does it use the caches more efficiently?

- To get started, run:
  - omniperf profile -n problem --no-roof -- ./problem.exe
  - omniperf profile -n solution --no-roof -- ./solution.exe
### Exercise 5: Sometimes the Full Story is in the Details

```
$ omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 16.3 17.2 17.3
```

<table>
<thead>
<tr>
<th>KernelName</th>
<th>Count</th>
<th>Count</th>
<th>Sum(ns)</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>yax(double*,</td>
<td>1.00</td>
<td>1.00</td>
<td>12443928.00</td>
<td>408316.0 (-96.72%)</td>
<td>12443928.00</td>
<td>408316.0 (-96.72%)</td>
<td>12443928.00</td>
<td>408316.0 (-96.72%)</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>(double*,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int, int,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 16. Vector L1 Data Cache

**16.3 L1D Cache Accesses**

- **Index**: 16.3.0
- **Metric**: Total Req
- **Avg**: 524368.00
- **Min**: 16448.00
- **Max**: 16448.00
- **Unit**: Req per wave

- **Index**: 16.3.5
- **Metric**: Cache Accesses
- **Avg**: 131140.00
- **Min**: 4097.00
- **Max**: 4097.00
- **Unit**: Req per wave

- **Index**: 16.3.6
- **Metric**: Cache Hits
- **Avg**: 65538.00
- **Min**: 2864.00
- **Max**: 2864.00
- **Unit**: Req per wave

- **Index**: 16.3.7
- **Metric**: Cache Hit Rate
- **Avg**: 49.98
- **Min**: 69.9 (39.87%)
- **Max**: 69.9 (39.87%)
- **Unit**: Pct

**17. L2 Cache**

- **Index**: 17.2.0
- **Metric**: Read BW
- **Avg**: 4194916.56
- **Min**: 65688.69
- **Max**: 65688.69
- **Unit**: Bytes per wave

- **Index**: 17.3.0
- **Metric**: Req
- **Avg**: 32945.33
- **Min**: 617.41
- **Max**: 617.41
- **Unit**: Req per wave

- **Index**: 17.3.6
- **Metric**: Hits
- **Avg**: 171.28
- **Min**: 104.03
- **Max**: 104.03
- **Unit**: Hits per wave

- **Index**: 17.3.7
- **Metric**: Misses
- **Avg**: 32774.06
- **Min**: 513.38
- **Max**: 513.38
- **Unit**: Misses per wave

- **Index**: 17.3.8
- **Metric**: Cache Hit
- **Avg**: 0.52
- **Min**: 16.85 (3140.15%)
- **Max**: 16.85 (3140.15%)
- **Unit**: Pct

**Note:** In the context of performance profiling, cache hit rates alone do not give a convincing reason for our performance increase. Large relative gains can be observed in both the L1D cache and L2 cache access times, indicating significant improvements in memory access performance.
Exercise 5: It Can Be Hard to Compare Rooflines Between Algorithms

- omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe
- omniperf profile -n solution_roof_only --roof-only --kernel-names -- ./solution.exe

Looking at just the rooflines, it’s difficult to tell which approach is more performant.

Problem FP32 Roofline

Solution FP32 Roofline

The problem is closer to being HBM bandwidth bound: It needs to request much more data from HBM than the optimized version.
Exercise 5: Omniperf Detailed Cache Statistics - Takeaways

- To get detailed cache statistics (including data movement) for kernel with dispatch ID 1:
  - omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 16.2 16.3 17.2 17.3
    - Note: The slide omitted some Omniperf output from this metric filtering

- Algorithmic optimizations can be powerful, but are usually time-intensive to design and implement

- It can be difficult to understand the performance differences between algorithms
  - Rooflines can be misleading
  - Assuming correctness is verified, timings don’t lie
  - Detailed profiling data can help shed light on the why of performance differences
DISCLAIMERS AND ATTRIBUTIONS

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

© 2023 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, Radeon™, Instinct™, EPYC, Infinity Fabric, ROCm™, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.