AMD GPU Hardware Basics

René van Oostrum, Noel Chalmers, Damon McDougall, Paul Bauman, Nicholas Curtis,
Nicholas Malaya, Noah Wolfe

Frontier Application Readiness Kick-Off Workshop
Oct 2019
AGENDA

AMD GCN Hardware Overview

AMD GCN Hardware Hierarchy

AMD GPU Compute Terminology

AMD GCN Compute Unit Internals

GPU Occupancy on GFX9
AMD GCN Hardware Overview
AMD Graphics Core Next (GCN) GPUs

AMD’s first GCN GPUs were released in 2012, family of chips code-named Southern Islands
- Multiple GCN generations released since then, with the most recent being GFX9 which uses the Vega Instruction Set Architecture (ISA)
- Current hardware and ISA documented in “Vega Instruction Set Architecture for GPUs”, see https://developer.amd.com/resources/developer-guides-manuals/
- Our recommendation is to target current GCN hardware
AMD GCN Hardware Hierarchy
AMD GCN GPU Hardware Layout
AMD GCN GPU Hardware Layout

Queues reside in user-visible DRAM

Command Queue

Command Queue

Command Processor

GPU
AMD GCN GPU Hardware Layout

Command Queue

Queues reside in user-visible DRAM

Command Queue

Command Processor

Shader Engine (SE0)

Shader Engine (SE3)

Shader Engine (SE1)

Shader Engine (SE2)
AMD GCN GPU Hardware Layout

Command Queue

Queues reside in user-visible DRAM

Command Queue

Command Processor

workload manager

workload manager

workload manager

workload manager
AMD GCN GPU Hardware Layout

Command Queue

Queues reside in user-visible DRAM

Command Queue

Command Processor

CU CU
CU CU
CU CU
CU CU
CU CU
CU CU
CU CU

workload manager

CU CU
CU CU
CU CU
CU CU
CU CU
CU CU
CU CU

workload manager

CU CU
CU CU
CU CU
CU CU
CU CU
CU CU
CU CU

workload manager

CU CU
CU CU
CU CU
CU CU
CU CU
CU CU
CU CU

workload manager

CU CU
CU CU
CU CU
CU CU
CU CU
CU CU
CU CU
AMD GPU Compute Terminology
Overview of GPU Kernels

GPU Kernel

Functions launched to the GPU that are executed by multiple parallel workers

Examples: GEMM, triangular solve, vector copy, scan, convolution
## Overview of GPU Kernels

### GPU Kernel

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Group of threads that are on the GPU at the same time. Also on the same compute unit. Can synchronize together and communicate through memory in the CU.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workgroup 1</td>
<td></td>
</tr>
<tr>
<td>Workgroup 2</td>
<td></td>
</tr>
<tr>
<td>Workgroup 3</td>
<td></td>
</tr>
<tr>
<td>Workgroup 4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Workgroup n</td>
<td>Programmer controls the number of workgroups – it’s usually a function of problem size.</td>
</tr>
</tbody>
</table>
### Overview of GPU Kernels

#### GPU Kernel

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Wavefront</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection of resources that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off. Can think of this as a vectorized thread. Lanes may access non-adjacent memory locations.</td>
<td></td>
</tr>
</tbody>
</table>

| Workgroup 1 |
| Workgroup 2 |
| Workgroup 3 |
| Workgroup 4 |
| ... |

<table>
<thead>
<tr>
<th>Workgroup n</th>
<th>CUDA Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp</td>
<td></td>
</tr>
</tbody>
</table>
Overview of GPU Kernels

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Workgroup 1</th>
<th>Workgroup 2</th>
<th>Workgroup 3</th>
<th>Workgroup 4</th>
<th>…</th>
<th>Workgroup n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wavefront 0</td>
<td>Wavefront 1</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>64 work items (threads)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Number of wavefronts / workgroup is chosen by developer (in HIP) or compiler (in OpenMP). GCN hardware allows up to 16 wavefronts in a workgroup.
Scheduling work to a GPU

Command Queue

Command Processor

Workload Manager

Workload Manager

Workload Manager

Workload Manager
GPU Memory, I/O, and Connectivity

GPU

- Command Processor
- Command Queue
- System Memory
- DMA Engines
- PCIe® Controllers
- HBM/GDDR Memory
- L2 Cache
- Memory Controllers
- Other GPUs
- DMA Engines
- Infinity Fabric Controllers

Frontier Application Readiness Kick-Off Workshop | Oct 2019
DMA Engines Accept Work from the Same Queues

**Step 1**
CPU submits a DMA Transfer packet to the command queue

This is done with user-level memory writes in Radeon Open Compute (ROCm). No kernel drivers involved.

---

**Command Processor**

DMA Engines

L2 Cache

Memory Controllers

HBM/GDDR Memory

System Memory

DMA Engines

PCIe® Controllers

Other GPUs

Infinity Fabric Controllers
DMA Engines Accept Work from the Same Queues

Step 2
CP Reads the packet and understands the transfer request

GPU

System Memory

Other GPUs

HBM/GDDR Memory

DMA Engines

PCIe® Controllers

L2 Cache

Memory Controllers

Command Queue

Command Queue

Infinity Fabric Controllers

DMA Engines
DMA Engines Accept Work from the Same Queues

Step 3
CP sends transfer command to the target DMA Engine

This can take place in parallel with other compute work & transfers

HBM/GDDR Memory
DMA Engines Accept Work from the Same Queues

Command Queue

Command Processor

Step 4
DMA Engines Transfer Data to/from GPU Memory

GPU

DMA Engines

PCIe® Controllers

System Memory

L2 Cache

Memory Controllers

HBM/GDDR Memory

Other GPUs
AMD GCN Compute Unit Internals
The GCN Compute Unit (CU)

The command processor sends work packages (i.e. workgroups of work-items in HIP) to the Compute Units (CUs)

- Blocks are executed in wavefronts (groups of 64 work-items on a SIMD)
- All wavefronts in a block reside on the same CU
- The CU’s scheduler can hold wavefronts from many blocks
- At most 40 wavefronts total per CU (10 per SIMD)
The GCN Compute Unit (CU)

- The Scalar Unit (SU)
  - Shared by all work-items in each wavefront, accessed on a per-wavefront level
  - Work-items in a wavefront performing the exact same operation can offload this instruction to the SU
  - Used for control flow, pointer arithmetic, dispatch a common constant value, etc.
  - Has its own pool of Scalar General-Purpose Register (SGPR) file, 12.5KB per CU
    - Maximum of 102 SGPRs / wavefront
The GCN Compute Unit (CU)

- **SIMD Units**
  - 4x SIMD vector units (each 16 lanes wide)
  - 4x 64KB (256KB total) Vector General-Purpose Register (VGPR) file
    - Maximum of 256 registers per SIMD – each register is 64x 4-byte entries
  - Instruction buffer for 10 wavefronts on each SIMD unit
    - Each wavefront is local to a single SIMD unit, not spread among the four (more on this in a moment)
The GCN Compute Unit (CU)

- 64KB Local Data Share (LDS, or shared memory)
  - 32 banks with conflict resolution
  - Can share data between all work-items in a workgroup
- 16 KB Read/Write L1 vector data cache
  - Write-through; L2 cache is the device coherence point – shared by all CUs
The GCN Compute Unit (CU)

- **Scheduler**
  - Buffer for up to 40 wavefronts – 2560 work-items
  - Separate decode/issue for
    - VALU, VGPR load/store
    - SALU, SGPR load/store
    - LDS load/store
    - Global mem load/store
    - Special instructions (NoOps, barriers, branch instructions)
The GCN Compute Unit (CU)

- **Scheduler**
  - At each clock, waves on **1 SIMD unit** are considered for execution (Round Robin scheduling among SIMDs)
  - At most **1 instruction per wavefront** may be issued
  - At most **1 instruction from each category** may be issued (SALU/VALU, SGPR/VGPR, LDS, global, branch, etc.)
  - **Maximum of 5** instructions issued to wavefronts on a single SIMD, per cycle per CU
  - VALU instructions take a multiple of four cycles to retire
    - e.g. FP32 FMA: cycle 0 – lanes 0-15 | cycle 1 – lanes 16-31 | cycle 2 – lanes 32-47 | cycle 3 – lanes 48-63
    - Programmer can still ‘pretend’ CU operates in 64-wide SIMD: 64 FP32 FMA ops / cycle / CU
# Hardware Configuration Parameters on Modern AMD GPUs

<table>
<thead>
<tr>
<th>GPU SKU</th>
<th>Chip Code Name</th>
<th>Shader Engines</th>
<th>CU / SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon Instinct™ MI60</td>
<td>Vega 20</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>AMD Radeon Instinct™ MI50</td>
<td>Vega 20</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>AMD Radeon™ VII</td>
<td>Vega 20</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>AMD Radeon Instinct™ MI25</td>
<td>Vega 10</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>AMD Radeon™ Vega 64</td>
<td>Vega 10</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>AMD Radeon™ Vega 56</td>
<td>Vega 10</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>AMD Radeon Instinct™ MI6</td>
<td>Polaris 10</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>
# Software Terminology

<table>
<thead>
<tr>
<th>NVIDIA/CUDA Terminology</th>
<th>AMD Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming Multiprocessor</td>
<td>Compute Unit (CU)</td>
<td>One of many parallel vector processors in a GPU that contain parallel ALUs. All waves in a workgroups are assigned to the same CU.</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel</td>
<td>Functions launched to the GPU that are executed by multiple parallel workers on the GPU. Kernels can work in parallel with CPU.</td>
</tr>
<tr>
<td>Warp</td>
<td>Wavefront</td>
<td>Collection of operations that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off. Think of this as a vector thread. A 64-wide wavefront is a 64-wide vector op.</td>
</tr>
<tr>
<td>Thread Block</td>
<td>Workgroup</td>
<td>Group of wavefronts that are on the GPU at the same time. Can synchronize together and communicate through local memory.</td>
</tr>
<tr>
<td>Thread</td>
<td>Work Item / Thread</td>
<td>Individual lane in a wavefront. On AMD GPUs, must run in lockstep with other work items in the wavefront. Lanes can be individually masked off. GPU programming models can treat this as a separate thread of execution, though you do not necessarily get forward sub-wavefront progress.</td>
</tr>
</tbody>
</table>
### Software Terminology

<table>
<thead>
<tr>
<th>NVIDIA/CUDA Terminology</th>
<th>AMD Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory</td>
<td>Global Memory</td>
<td>DRAM memory accessible by the GPU that goes through some layers cache</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Local Memory</td>
<td>Scratchpad that allows communication between wavefronts in a workgroup.</td>
</tr>
<tr>
<td>Local Memory</td>
<td>Private Memory</td>
<td>Per-thread private memory, often mapped to registers.</td>
</tr>
</tbody>
</table>
GPU Occupancy on GFX9
GPUs: massively parallel, resource limited

AMD GPUs are massively parallel processors with relatively limited on-chip resources. On modern AMD GPUs, typically there are:

- 64 or 60 compute units (CUs), each containing
  - 4x16-wide SIMDs
  - 4x64KB Vector General Purpose Register (VGPR) file
  - 64KB Local Data Share (LDS)
  - 16KB Read/Write L1 vector cache
  - 12.5KB Scalar General Purpose Register (SGPR) file
  - Instruction buffer allowing for 10 wavefronts (WF) in flight per SIMD (→40 WF/CU)

- Achieving improved performance often requires balancing the utilization of different resources pools
What is Occupancy?

**Occupancy**: the ratio of active WF executing on the GPU to the maximum number of possible WF supported by the hardware.

- Occupancy is controlled by the utilization of resources on a CU
- Can indicate over/under utilization of resources, limiting performance

Different “flavors” of occupancy available:

- **Achieved** occupancy is measured on the hardware and is a time-dependent metric (as the number of active WF is not constant).
- **Theoretical** occupancy is a calculated metric, derived from the resources requested by the kernel
- In addition, occupancy may be reported per-CU, or per-GPU

To see why occupancy is important, we will consider a batch matrix-vector multiply kernel.
Occupancy: limiting factors

- Number of wavefronts: max 10 per SIMD, 40 per CU
- Number of wavefronts per workgroup (AKA thread block): max 16 (i.e., max 1024 threads per workgroup).
  - Note that all wavefronts of a workgroup are required to be scheduled on the same CU, but not necessarily on the same SIMD of the CU.
  - Note that with 16 wavefronts per workgroup, we can schedule at most 32 wavefronts on the CU. The remaining 8 are insufficient for another workgroup.
- Number of workgroups per CU:
  - If workgroups have just one wavefront: max number of workgroups/CU is 40
  - If workgroups have more than one wavefront: max number of workgroups/CU is 16
  - Example: if workgroups have three wavefronts, max #wavefronts/CU is 39 (= 13*3)
- Corollary: to maximize occupancy, the #wavefronts/workgroup should be 1, 4, 5, or 8
  - Note that this is a necessary condition, but not a sufficient condition
  - Note again that maximizing occupancy isn’t always necessary for maximum performance
  - Data layout (e.g., stencil size) or algorithmic considerations may dictate other workgroup sizes
Occupancy: limiting factors—register usage

- AMD GPU hardware has two types of general purpose registers:
  - Scalar registers: one instance per wavefront
    - s0, s1, ..
    - Used for e.g. pointers to the base of a data block, and for branching
  - Vector registers: one instance per thread
    - v0, v1, ...

- Registers are 32 bits wide, but can be combined into wider registers:
  - E.g. $s[6:7]$ forms a 64-bit scalar register
    - Lower order bits are in lower numbered register
    - Scalar register pairs forming a 64-bit register must be even-aligned ($s[7:8]$ is not allowed)
    - No such alignment is required for vector register pairs
## Occupancy: limiting factors—register usage

- **Scalar registers:**
  - Total scalar register file size: 12.5 KB (3,200 registers, 800 per SIMD)
  - A single wavefront can allocate up to 112 scalar registers in batches of 16
    - The last 6 of these are used for special purposes (such as VCC), and these cannot be used as general purpose scalar registers by user code
    - The 112 case is special; here, 4 additional registers cannot be used, leaving 102 for GPR purposes
    - For each wavefront, 16 additional registers are allocated for a trap handler

<table>
<thead>
<tr>
<th># SGPRs reserved</th>
<th>16</th>
<th>32</th>
<th>48</th>
<th>64</th>
<th>80</th>
<th>96</th>
<th>112</th>
</tr>
</thead>
<tbody>
<tr>
<td># SGPRS available</td>
<td>10</td>
<td>26</td>
<td>42</td>
<td>58</td>
<td>74</td>
<td>90</td>
<td>102</td>
</tr>
<tr>
<td># allocated inc +16</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td>80</td>
<td>96</td>
<td>112</td>
<td>128</td>
</tr>
<tr>
<td># wavefronts / SIMD</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
Occupancy: limiting factors—register usage

- Vector registers:
  - Vector register file size: 64 KB per SIMD (16K registers = 64 * 256)
  - A single wavefront can allocate up to 256 vector registers per thread
    - Scalar registers are allocated in batches of 4 registers

<table>
<thead>
<tr>
<th># VGPRs</th>
<th>&lt;=24</th>
<th>28</th>
<th>32</th>
<th>36</th>
<th>40</th>
<th>48</th>
<th>64</th>
<th>84</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td># wavefronts / SIMD</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Occupancy: limiting factors—LDS usage

- Local Data Store:
  - Fast memory that can be used to share data across a thread block/workgroup
    - Note that for occupancy calculations, we need to look at the usage per workgroup, not per wavefront
  - AMD equivalent of CUDA’s __shared__ memory
  - 64 KB per Compute Unit
Example: batched matrix-vector multiply

As a test-bed for our occupancy calculations, we will use a batched matrix-vector multiplication kernel:

- $\tilde{A}$ is a $(N_m \times N_m)$ matrix
- $\tilde{x}$ and $\tilde{b}$ are $N_v$ vectors each of size $(N_m \times 1)$
Example: batched matrix-vector multiply

Main implementation ideas:

- Every work-item multiplies $\tilde{A}$ with multiple vectors from $\tilde{x}$.
- The data of a vector from $\tilde{x}$ is reused $N_m$ times.
- Instead of loading a vector from $\tilde{x}$ from HBM for every use, we preload a batch of WG-size * $N_b$ of them in (faster) LDS, and use them repeatedly from there.
Example occupancy calculation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WG-size</td>
<td>128</td>
</tr>
<tr>
<td>$N_m$</td>
<td>4</td>
</tr>
<tr>
<td>$N_b$</td>
<td>32</td>
</tr>
<tr>
<td>$N_y$</td>
<td>4.90E+08*</td>
</tr>
</tbody>
</table>

Kernel configuration V0

* Limits maximum memory allocation to ~2GB

Resulting performance ~33 GFLOP/s, very poor! Why?
- **One reason**: using too much LDS per work-group!

\[
\text{LDS} = \text{WG}_{\text{size}} \times N_b \times N_m \times \text{sizeof(float)} \\
= 128 \times 32 \times 4 \times 4 \text{ bytes} \\
= 64 \text{ KB/WG}
\]
Example occupancy calculation

**Recall:** 64KB of LDS available per CU
→ Limited to a single WG of 128 work-items (or two WF) per CU in this configuration!

**Recall:** 40 Wavefronts possible per CU:
→ Occupancy = \( \frac{2}{40} = 0.05 \)

**Solution:** lower LDS usage per WG
- In this example, we can either decrease the workgroup size, or **decrease the batch size** \( N_b \)
Example occupancy calculation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WG-size</td>
<td>128</td>
</tr>
<tr>
<td>$N_m$</td>
<td>4</td>
</tr>
<tr>
<td>$N_b$</td>
<td>1</td>
</tr>
<tr>
<td>$N_v$</td>
<td>4.90E+08</td>
</tr>
</tbody>
</table>

Kernel configuration V1

In this configuration:
- LDS/WG = 2KB
- 32 WG/CU
- 2 WF/WG
- Occupancy limit from LDS \(= \frac{32+2}{40} = 1.6\)
  \(\rightarrow\) no longer limited by LDS usage

In our informal tests, reducing the batch size from 32 to 1 resulted in a performance increase to \(~215\) GFLOP/s (\(~6.5\times\) speedup)
Example occupancy calculation

Next, we consider the limitation on the number of workgroups/CU and wavefronts/CU:

Limit: 16 WG/CU
- Exception: doesn’t apply to workgroups of a single wavefront (i.e., WG-size=64)

Limit: 40 Wavefronts/CU

To reach full occupancy:

\[ \text{WG} \times \frac{\text{WF}}{\text{WG}} \geq 40 \]

Currently we have:

\[ \text{WG} \times \frac{\text{WF}}{\text{WG}} = 16 \times 2 = 32 \]

Occupancy = \( \frac{32}{40} = 0.8 \)

Solution: Increase WG-size to 256 \( \rightarrow \) WF/WG=4
Example occupancy calculation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WG-size</td>
<td>256</td>
</tr>
<tr>
<td>N_m</td>
<td>4</td>
</tr>
<tr>
<td>N_b</td>
<td>1</td>
</tr>
<tr>
<td>N_v</td>
<td>4.90E+08</td>
</tr>
</tbody>
</table>

Limits:
- LDS limit:
  - LDS/WG = 4KB → 16 WG/CU
- WG limit:
  - Target WG/CU=16
  - WF/WG=4

Occupancy limit from LDS and WG-size:

\[
\text{Occupancy} = \min\left(\frac{\text{WG}}{\text{CU}_{\text{LDS}}}, \frac{\text{WG}}{\text{CU}_{\text{WG}}}\right) \ast \frac{\text{WF}}{\text{WG}} \ast \frac{1}{40}
\]

\[
= \min(16, 16) \ast 4 \ast \frac{1}{40} = 1.6
\]

→ No longer limited by WG-size

However... **performance didn’t increase** (~215 GFLOP/s)
Example occupancy calculation

Occupancy is **not a silver bullet**!
- high occupancy does not always imply peak performance,
- conversely, low occupancy does not imply poor performance

In our case, increasing occupancy from 0.8 to 1.0 had little effect on performance!

For example:
- Increasing occupancy often doesn’t result in improved performance if there is already enough occupancy to hide latencies in the kernel with context switching
Example occupancy calculation

Profiling shows only ~50% VALU usage and low VGPR usage.

**Idea:**
- Apply unrolling to the inner matrix vector multiply
- Experiment with different unroll factors $N_u$
Example occupancy calculation

Varying $N_u$ controls the number of VGPRs allocated per work-item:

VGPR limit on occupancy:

- Recall: 256 KB VGPRs/CU, 4 SIMD/CU
- Minimum of 1WF/SIMD
  $\rightarrow$ Minimum of 64 work-items/SIMD
  $\rightarrow$ For floats, we have at most:

$$\frac{256\text{KB}}{4\text{B}} \times \frac{1\text{CU}}{4\text{SIMD}} \times \frac{1\text{SIMD}}{64\text{WI}} = 256 \frac{\text{VGPRs}}{\text{WI}}$$
Example occupancy calculation

Take $N_u=8$ case*:

$$\frac{WF}{CU_{VGPR}} = \left[ \frac{256 \left( \frac{VGPR}{WI} \right)_{\text{max}} \times \frac{1WF}{SIMD} \times \frac{4SIMD}{CU} \right] = 36$$

This is also limited to wavefronts that can fit into a workgroup:

$$\frac{WG}{CU_{VGPR}} = \frac{WF}{CU_{VGPR}}$$

For our work-group size of 256 ($WF/WG = 4$):

$$\frac{WG}{CU_{VGPR}} = 9$$

*ignoring for the moment that our matrix size is only 4
### Example occupancy calculation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WG-size</td>
<td>256</td>
</tr>
<tr>
<td>$N_m$</td>
<td>4</td>
</tr>
<tr>
<td>$N_b$</td>
<td>1</td>
</tr>
<tr>
<td>$N_v$</td>
<td>$4.90 \times 10^8$</td>
</tr>
<tr>
<td>$N_u$</td>
<td>8</td>
</tr>
</tbody>
</table>

**Kernel configuration V3**

From earlier:

**LDS limit:**
- LDS/WG = 4KB → 16 WG/CU

**WG limit:**
- Target WG/CU=16
  
**WF/WG=4**

**VGPR Limit:**
- 9 WG/CU

**Occupancy** = \[
\min \left( \frac{\text{WG}}{\text{CU}_{\text{LDS}}}, \frac{\text{WG}}{\text{CU}_{\text{WG}}}, \frac{\text{WG}}{\text{CU}_{\text{VGPR}}} \right) \frac{\text{WF}}{\text{WG}} \frac{1}{40}
\]

\[
= \min(16, 16, 9) \times 4 \times \frac{1}{40} = 0.9
\]
Wrap up

Occupancy is **not a silver bullet!**
- high occupancy does not always imply peak performance,
- conversely, low occupancy does not imply poor performance

Good rules of thumb:
- Bandwidth bound kernels may achieve good performance even with low occupancy, as the key is to saturate the memory controller
- For some kernels, data dependencies may cause compute operations to stall. Here, a high occupancy is beneficial to allow context switches in order to hide latencies
- Some compute bound kernels may benefit from higher occupancy to achieve high instruction throughput, however other kernels may benefit from lower occupancy to allow increased VGPR usage and avoid spillage
- **Vary parameters to see interplay between resource usage (LDS, VGPR, WG-size) and occupancy to obtain maximum performance**
Wrap up

Vary parameters to see interplay between resource usage (LDS, VGPR, WG-size) and occupancy to obtain maximum performance!

- For example... this kernel configuration achieves over 575 GFLOP/s with an occupancy of 0.4!

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WG-size</td>
<td>512</td>
</tr>
<tr>
<td>$N_m$</td>
<td>16</td>
</tr>
<tr>
<td>$N_b$</td>
<td>1</td>
</tr>
<tr>
<td>$N_v$</td>
<td>4.90E+08</td>
</tr>
<tr>
<td>$N_u$</td>
<td>8</td>
</tr>
</tbody>
</table>

Kernel configuration V4
DISCLAIMER

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

©2019 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Radeon is the registered trademark of Advance Micro Devices, Inc. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.