[KISTI 기술문서] ISBN 978-89-294-1716-1

AMD GPU를 활용한 HIP 프로그래밍

2024. 10. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

곽재혁

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
jhkwak@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

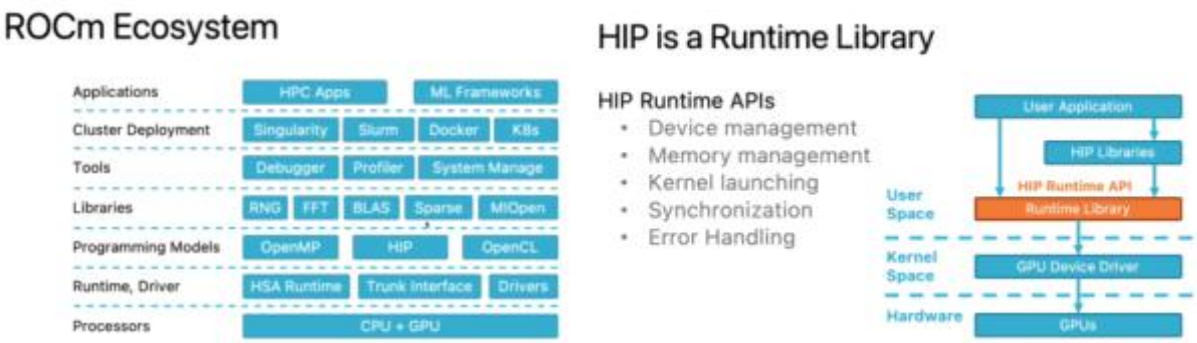
목 차

1. 개요

- GPU (그래픽 처리 장치)의 발전은 PC(개인용컴퓨터)의 발전과 함께 시작되어 지속적으로 발전되어옴.
* 초기 GPU (1980년대 ~ 1990년대 중반) : 주로 2D 처리에 특화됨
* 3D 가속 및 하드웨어 T&L(Transfom and Lighting) (1990년대 후반) : 3D 지오메트리와 조명 연산을 하드웨어로 처리
* 프로그래머블 셰이더 등장(2000년대 초반) : 3D 그래픽스 렌더링 파이프라인에서 특정 단계를 프로그램 가능하게 함으로써 텍스쳐 매핑, 조명, 그림자, 반사 등의 효과를 조정할 수 있음. HLSL, GLSL 등의 프로그래밍 언어가 사용됨
* GPGPU의 시작(2000년대 중반~후반) : GPU를 그래픽 이외의 범용 연산에 활용하기 시작. 버텍스 셰이더와 픽셀 셰이더의 통합으로 그래픽스 파이프라인이 더 단순화 되며, 유연한 프로그래밍이 가능
* CUDA와 OpenCL의 등장(2000대 후반~) : GPU 프로그래밍 접근성이 쉬워지고, 과학계산, 머신 러닝 등의 다양한 분야에서 활용이 확대됨. 텐서코어 등 AI 연산에 특화된 하드웨어가 추가됨.

- ROCm(Radeon Open Compute platform)은 AMD가 2016년에 발표한 GPGPU 소프트웨어 스택으로 NVIDIA의 CUDA에 대응하는 솔루션임. NVIDIA의 CUDA가 2006년에 발표되었고, OpenCL은 애플의 주도로, 인텔, AMD 등이 참여한 Khronos Group에 의해 2008년 발표됨. AMD는 2011년 부터 ATI APP Software Development Kit 2.5 부터 OpenCL을 지원하였으나, OpenCL 생태계의 부진으로 2015년 11월 SC15에서 볼츠만 계획(Boltzmann Initiative)를 발표하고, HCC (Heterogeneous Compute Compiler), HIP (Heterogeneous-Compute Interface for Portability), HSA (Heterogeneous System Architecture) 런타임 등의 개발 계획을 발표함. 그 결과 2016년 ROCm 이 공식 출시됨. 2024년 8월 ROCm 6.2가 발표됨.[1)]

- ROCm은 포괄적인 오픈소스 플랫폼으로 다양한 구성 요소와 도구를 제공하여 GPU기반의 HPC와 AI 애플리케이션 개발을 지원함[2)]
* 드라이버 및 런타임: ROCk(커널 드라이버)[3)], ROCr(HSA 런타임 API 구현)[4)], ROCm(디바이스 라이브러리, LLVM 기반으로 구현)[5)], ROCt 라이브러리(사용자 모드 API 구현)[6)]
* 개발 도구: 컴파일러, 디버거, 프로파일러(rocprof, omnitrace, omniperf, rocgdb) 등
* 프로그래밍 모델: HIP, OpenMP, OpenCL
* 라이브러리: MIOpen, MIVisionX, rocBLAS, rocFFT, rocRAND 등



1) https://insidehpc.com/2024/08/amd-announces-rocm-6-2-software-stack-for-gpu-programming/
2) https://en.wikipedia.org/wiki/ROCm
3) https://github.com/ROCm/ROCK-Kernel-Driver
4) https://github.com/ROCm/ROCR-Runtime
5) https://github.com/ROCm/ROCm-Device-Libs
6) https://github.com/ROCm/ROCT-Thunk-Interface

- AMD의 GPU 아키텍쳐 발전 과정을 다음과 같이 정리함

* TeraScale[7]: AMD가 ATI를 인수한 후 출시한 초기 아키텍처. GPGPU 기능의 초기 단계를 지원함
* GCN(Graphics Core Next)[8] : AMD의 초기 통합 GPU 아키텍쳐, 범용 목적 셰이딩 코어와 고정 함수 그래픽스 하드웨어를 포함함. 그래픽스와 컴퓨트 워크로드 모두를 위해 설계됨.
* RDNA(Radeon DNA)[9] : 게이밍에 최적화된 GPU아키텍쳐, GCN 대비 클럭당 1.25배, 전력당 1.5배의 성능 향상
* CDNA(Compute DNA)[10]: AI 및 고성능 컴퓨팅에 특화된 아키텍처, 칩렛 기술을 적용하여 고성능 및 확장성을 제공.

- AMD GPU 세대별 아키텍쳐 비교

아키텍쳐	제품명	GFX ID/코드명	발표일
GCN 1	HD 7000 시리즈	gfx6	2012년 1월
GCN 2	Rx 200 시리즈	gfx7	2013년
GCN 3	Rx 300/400 시리즈	gfx8	2015 ~ 2017년
GCN 5	Rx VEGA 시리즈, Radeon VII	gfx9	2017 ~ 2019년
RDNA 1	Radeon RX 5000 시리즈	gfx10 / Navi 1X	2019년
RDNA 2	Radeon RX 6000 시리즈	gfx10.3 / Navi 2X	2020년 11월
RDNA 3	Radeon RX 7000 시리즈	gfx11 / Navi 3X	2022년 11월
CDNA 1	Instinct MI100 시리즈	gfx908 / Arcturus	2020년 11월
CDNA 2	Instinct MI200 시리즈	gfx90a Aldeberan	2021년 11월
CDNA 3	Instinct MI300 시리즈	gfx940	2023년 1월

※ GFX ID는 그래픽 프로세서의 세대와 아키텍처를 나타내는 식별자임. NVIDIA 에서는 SM(streaming multiprocessor) 버전을 사용하여 식별함.

※ ROCm은 이전에 사용하던 hcc 컴파일러에서 Clang 컴파일러로 전환됨. HCC 컴파일러[11]는 2019년 6월에 공식적으로 지원이 중단되고, Clang/LLVM 기반의 컴파일러로 전환됨. Clang/LLVM은 널리 사용되는 오픈 소스 컴파일러 인프라로 C, C++, OpenCL 등 다양한 언어를 지원하며, 대규모 개발자 커뮤니티를 가지고 있어, 지속적이 개선과 버그 수정이 이루어짐.

※ RDNA1 와 MI100(CDNA 1)의 메모리 아키텍처 비교

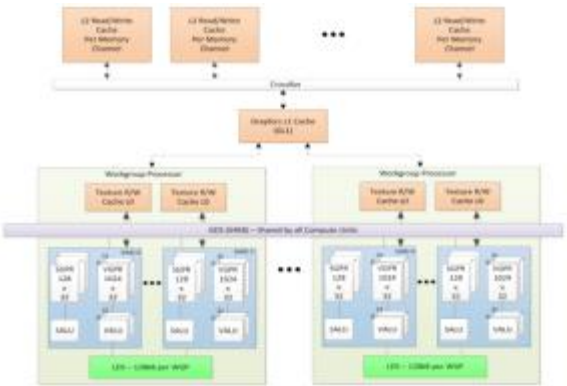


그림 3 RDNA 1 Shared Memory Hierarchy

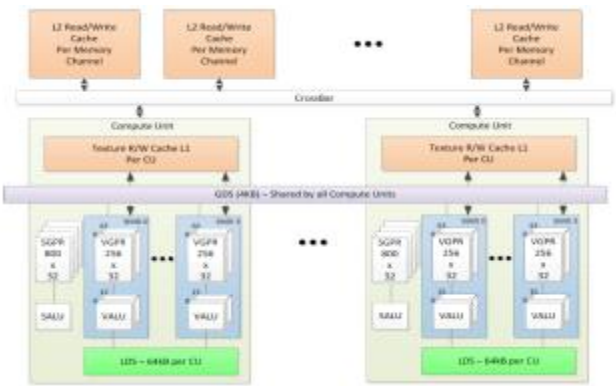


그림 4 MI100 아키텍처

- 본 문서에서 사용자 예제코드는 다음 깃허브 주소에서 찾을 수 있다:

```
git clone https://github.com/swkim85/linux-drill ; cd rocm
```

7) https://en.wikipedia.org/wiki/TeraScale_(microarchitecture)
8) https://en.wikipedia.org/wiki/Graphics_Core_Next
9) https://en.wikipedia.org/wiki/RDNA_(microarchitecture)
10) https://en.wikipedia.org/wiki/CDNA_(microarchitecture)
11) https://github.com/ROCm/hcc

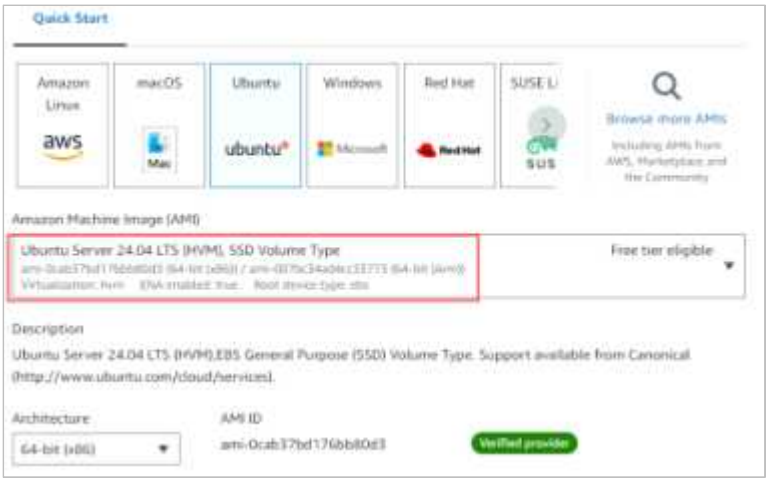
2. 환경 설정

2.1 아마존 인스턴스 설정

- AWS는 G4ad 인스턴스를 통해 AMD Radeon Pro V520 GPU [12]를 제공 [13] [14]
- Radeon PRO V520 GPU의 스펙은 다음과 같다:

항목	설명
GPU Architecture	RDNA 1.0
출시시기	2020년 12월
GPU 코드명	Navi 12
Lithography	TSMC 7nm FinFET
Stream Processors	2304
Compute Units	36
Peak Engine Clock (부스트 클럭)	1.6 GHz
Peak Half Precision (FP16) Performance	14.75 TFLOPs
Peak Single Precision Matrix (FP32) Performance	7.4 TFLOPs
Peak Single Precision (FP32) Performance	7.4 TFLOPs
Peak Double Precision (FP64) Performance	460 GFLOPs
Peak INT4 Performance	58.98 TOPs
Peak INT8 Performance	24.94 TOPs
메모리	8GB HBM2 (2048-bit 버스)
인터페이스	PCIe 4.0 x16

- 아마존 AWS EC2 인스턴스 생성하기: OS는 Ubuntu 최신 버전인 24.04를 선택. 인스턴스 유형은 `g4ad.xlarge` 를 선택



12) https://www.amd.com/en/products/accelerators/radeon-pro/amd-radeon-pro-v520.html
13) https://aws.amazon.com/ko/blogs/korea/new-amazon-ec2-g4ad-instances-featuring-amd-gpus-for-graphics-workloads/
14) https://aws.amazon.com/ko/ec2/instance-types/g4/

- GPU 디바이스 확인하기. **lspci** 명령을 통해서 제공되는 V520 GPU를 확인한다.

```
# lspci -v  ¦ grep AMD
00:1e.0 Display controller: Advanced Micro Devices, Inc. [AMD/ATI] Navi 12 [Radeon Pro V520/V54
0] (rev c3)
        Subsystem: Advanced Micro Devices, Inc. [AMD/ATI] Navi 12 [Radeon Pro V520/V540]

  -nn 옵션을 추가하면 PCI vendor 와 device 코드를 조회할 수 있음
$ lspci -nnvv
[...]
00:1e.0 Display controller [0380]: Advanced Micro Devices, Inc. [AMD/ATI] Navi 12 [Radeon Pro V520/V540]
[1002:7362] (rev c3)
        Subsystem: Advanced Micro Devices, Inc. [AMD/ATI] Navi 12 [Radeon Pro V520/V540] [1002:0a34]
        Physical Slot: 30
        Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR+ FastB2B- DisINTx-
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Region 0: Memory at 440000000 (64-bit, prefetchable) [size=256M]
        Region 2: Memory at 450000000 (64-bit, prefetchable) [size=2M]
        Region 5: Memory at feb00000 (32-bit, non-prefetchable) [size=512K]
        Capabilities: <access denied>
```

※ 해당 디바이스의 Vendor ID는 **1002,** Device ID는 **7362** 임을 알수 있다. PCILookup 사이트[15]를 통해서 해당 ID를 조회할 수 있다. Navi 12[Radeon Pro V520] 임을 확인한다.

2.2 커널 모듈 및 ROCm 소프트웨어 설치

- AMD ROCm(Radeon Open Compute)[16] 소프트웨어 설치하기. Quick start installation 가이드[17]를 참고. 우분투 최신 버전인 24.04를 기준으로 함.

```
 패키지 목록을 최신화함
$ sudo apt update
$ uname -r      리눅스 커널 버전을 확인
6.8.0-1012-aws
  리눅스 커널 헤더 파일과 추가적인 커널 모듈 을 설치
$ sudo apt install "linux-headers-$(uname -r)" "linux-modules-extra-$(uname -r)"

  현재 사용자(ubuntu)를 video,render 그룹에 추가함. 추가한 후에 쉘에 다시 로그인 해야 적용됨
$ sudo usermod -a -G render,video $LOGNAME
$ id ubuntu
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),24(cdrom),27(sudo),30(dip),44(vide
o),105(lxd),992(render)
```

15) https://www.pcilookup.com/
16) https://www.amd.com/en/products/software/rocm.html
17) https://rocm.docs.amd.com/projects/install-on-linux/en/latest/install/quick-start.html

```
 amdgpu-install 스크립트를 사용하기 위하여 설치 파일을 다운로드하고 설치
$ wget https://repo.radeon.com/amdgpu-install/6.2/ubuntu/noble/amdgpu-install_6.2.60200-1_all.deb
$ sudo apt install ./amdgpu-install_6.2.60200-1_all.deb
$ which amdgpu-install
/usr/bin/amdgpu-install
 패키지가 설치되었는지 확인
$ dpkg -l ¦ grep amdgpu
ii  amdgpu-install    6.2.60200-2009582.24.04    all  AMDGPU driver repository and installer
$ dpkg -L amdgpu-install

 툴을 이용하여 설치 할 수 있는 usecase를 조회
$ sudo amdgpu-install --list-usecase
If --usecase option is not present, the default selection is
"dkms,graphics,opencl,hip"
Available use cases:
dkms            (to only install the kernel mode driver)
  - Kernel mode driver (included in all usecases)
graphics        (for users of graphics applications)
  - Open source Mesa 3D graphics and multimedia libraries
multimedia      (for users of open source multimedia)
  - Open source Mesa 3D multimedia libraries
workstation     (for users of legacy WS applications)
  - Open source multimedia libraries
  - Closed source (legacy) OpenGL
rocm            (for users and developers requiring full ROCm stack)
  - OpenCL (ROCr/KFD based) runtime
  - HIP runtimes
  - Machine learning framework
  - All ROCm libraries and applications
[...]

  amdgpu-dkms DKMS(Dynamic Kernel Module Support) 패키지를 설치. 커널에 맞는 드라이버를 빌드하는 과정이 포함됨
$ amdgpu-install --usecase=dkms
[...]
Setting up amdgpu-dkms (1:6.8.5.60200-2009582.24.04) ...
Loading new amdgpu-6.8.5-2009582.24.04 DKMS files...
Building for 6.8.0-1012-aws
Building for architecture x86_64
Building initial module for 6.8.0-1012-aws
[...]
Forcing installation of amdgpu

amdgpu.ko.zst:
Running module version sanity check.
 - Original module
 - Installation
   - Installing to /lib/modules/6.8.0-1012-aws/updates/dkms/

amdttm.ko.zst:
Running module version sanity check.
 - Original module
 - Installation
```

```
   - Installing to /lib/modules/6.8.0-1012-aws/updates/dkms/
[...]

   커널 모듈 설치 후에 시스템을 재부팅해 준다.
$ sudo reboot
  lsmod 에서 모듈이 로드되었는 확인
$ lsmod ¦ grep amdgpu
amdgpu              19644416  0
amddrm_ttm_helper      12288  1 amdgpu
amdttm                118784  2 amdgpu,amddrm_ttm_helper
amddrm_buddy           24576  1 amdgpu
amdxcp                 12288  1 amdgpu
amd_sched              61440  1 amdgpu
amdkcl                 32768  3 amd_sched,amdttm,amdgpu
[...]

  커널 메시지에서 kfd (kernel fusion driver) 관련 메시지를 확인
$ sudo sysctl kernel.dmesg_restrict=0
$ sudo dmesg  ¦ grep amdgpu ¦ grep kfd
[    7.976950] kfd kfd: amdgpu: Allocated 3969056 bytes on gart
[    7.976968] kfd kfd: amdgpu: Total number of KFD nodes to be created: 1
[    7.978317] kfd kfd: amdgpu: added device 1002:7362

  kfd 디바이스 파일을 확인. GPU와 커널간의 인터페이스를 제공한다.
$ ls -la /dev/kfd
crw-rw---- 1 root video 236, 0 Aug 21 07:33 /dev/kfd
  dkms 명령에서 커널 모듈의 상태를 확인
$ dkms status
amdgpu/6.8.5-2009582.24.04, 6.8.0-1012-aws, x86_64: installed
 커널 모듈 소스는 아래 경로에 설치된다.
$ tree -d /usr/src/amdgpu-6.8.5-2009582.24.04

 ROCm 패키지를 설치한다.
$ sudo amdgpu-install --usecase=hiplibsdk,rocm
 설치된 패키지 목록들을 확인하기
$ apt list --installed ¦ egrep "amdgpu¦rocm"
amdgpu-core/noble,now 1:6.2.60200-2009582.24.04 all [installed,automatic]
amdgpu-dkms-firmware/noble,now 1:6.8.5.60200-2009582.24.04 all [installed,automatic]
amdgpu-dkms/noble,now 1:6.8.5.60200-2009582.24.04 all [installed]
amdgpu-install/noble,now 6.2.60200-2009582.24.04 all [installed]
libdrm-amdgpu-amdgpu1/noble,now 1:2.4.120.60200-2009582.24.04 amd64 [installed,automatic]
libdrm-amdgpu-common/noble,now 1.0.0.60200-2009582.24.04 all [installed,automatic]
libdrm-amdgpu-dev/noble,now 1:2.4.120.60200-2009582.24.04 amd64 [installed,automatic]
libdrm-amdgpu-radeon1/noble,now 1:2.4.120.60200-2009582.24.04 amd64 [installed,automatic]
libdrm-amdgpu1/noble,now 2.4.120-2build1 amd64 [installed,automatic]
libdrm2-amdgpu/noble,now 1:2.4.120.60200-2009582.24.04 amd64 [installed,automatic]
[...]

 ROCm System Management Interface 를 이용하여 정보를 확인. 전력소모량, 온도 등을 확인 가능
$ rocm-smi
======================================== ROCm System Management Interface ========================================
================================================== Concise Info ==================================================
Device  Node  IDs              Temp    Power  Partitions          SCLK  MCLK     Fan     Perf  PwrCap  VRAM%  GPU%
              (DID,     GUID)  (Edge)  (Avg)  (Mem, Compute, ID)
==================================================================================================================
```

```
0       1     0x7362,   50651  34.0°C  10.0W  N/A, N/A, 0         None  1000Mhz  18.82%  auto  0.0W    0%     3%
================================================================================================================
============================================ End of ROCm SMI Log ============================================
```

amd-smi 명령을 통해서 메모리 정보등을 확인

```
$ amd-smi  monitor
GPU  POWER  GPU_TEMP  MEM_TEMP  GFX_UTIL  GFX_CLOCK  MEM_UTIL  MEM_CLOCK  ENC_UTIL  ENC_CLOCK
DEC_UTIL  DEC_CLOCK  SINGLE_ECC  DOUBLE_ECCPCIE_REPLAY  VRAM_USED  VRAM_TOTAL   PCIE_BW
  0   11 W     34 °C     31 °C       3 %   1650 MHz       0 %   1000 MHz       N/A    105 MHz
    N/A    102 MHz           0           0          0      12 MB     8027 MB  N/A Mb/s
```

rocminfo 명령을 통해서 시스템의 HSA(Heterogeneous System Architecture) 속성과 가용한 컴퓨팅 장치를 열거. Agent1 은 CPU, Agent 2는 GPU에 해당한다.

```
$ rocminfo
ROCk module version 6.8.5 is loaded
=====================
HSA System Attributes
=====================
Runtime Version:         1.14
Runtime Ext Version:     1.6
System Timestamp Freq.:  1000.000000MHz
Sig. Max Wait Duration:  18446744073709551615 (0xFFFFFFFFFFFFFFFF) (timestamp count)
Machine Model:           LARGE
System Endianness:       LITTLE
Mwaitx:                  DISABLED
DMAbuf Support:          YES
==========
HSA Agents
==========
*******
Agent 1
*******
  Name:                    AMD EPYC 7R32
  Uuid:                    CPU-XX
[...]
*******
Agent 2
*******
  Name:                    gfx1011
```

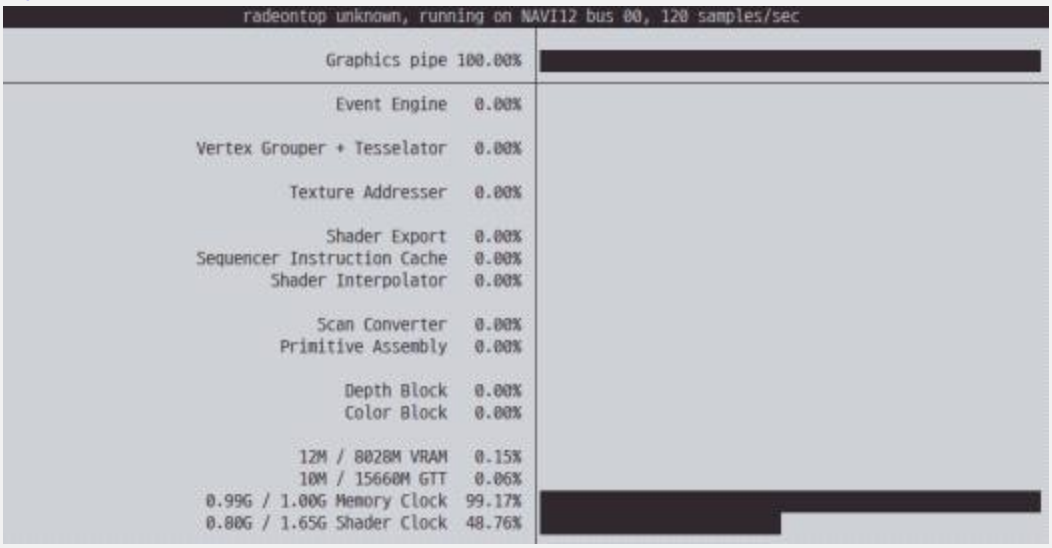
GFX ID 값은 AMD 그래픽 카드의 아키텍쳐와 세대를 식별하는 정보임. GFX10xx 은 RDNA 아키텍처를 의미함

```
  Uuid:                    GPU-XX
  Marketing Name:          AMD Radeon Pro V520
  Vendor Name:             AMD
  Feature:                 KERNEL_DISPATCH
  Profile:                 BASE_PROFILE
  Float Round Mode:        NEAR
  Max Queue Number:        128(0x80)
  Queue Min Size:          64(0x40)
  Queue Max Size:          131072(0x20000)
  Queue Type:              MULTI
  Node:                    1
  Device Type:             GPU
  Cache Info:
    L1:                      16(0x10) KB
```

```
    L2:                      4096(0x1000) KB
  Chip ID:                 29538(0x7362)
  ASIC Revision:           0(0x0)
  Cacheline Size:          64(0x40)
  Max Clock Freq. (MHz):   1650
  BDFID:                   240
  Internal Node ID:        1
  Compute Unit:            36
  SIMDs per CU:            2
  Shader Engines:          2
  Shader Arrs. per Eng.:   2
  WatchPts on Addr. Ranges:4
  Coherent Host Access:    FALSE
  Memory Properties:
  Features:                KERNEL_DISPATCH
  Fast F16 Operation:      TRUE
  Wavefront Size:          32(0x20)
  Workgroup Max Size:      1024(0x400)
  Workgroup Max Size per Dimension:
    x                        1024(0x400)
    y                        1024(0x400)
    z                        1024(0x400)
  Max Waves Per CU:        40(0x28)
  Max Work-item Per CU:    1280(0x500)
  Grid Max Size:           4294967295(0xffffffff)
  Grid Max Size per Dimension:
    x                        4294967295(0xffffffff)
    y                        4294967295(0xffffffff)
    z                        4294967295(0xffffffff)
  Max fbarriers/Workgrp:   32
  Packet Processor uCode:: 149
  SDMA engine uCode::      44
[...]
```

radeontop 명령을 통하여 GPU의 실시간 정보를 모니터링 할수 있다. 비디오 메모리(VRAM) 사용량, GPU 코어 클럭 속도, 그래픽 파이프라인/쉐이더/텍스쳐 관련 사용율

```
$ apt install radeontop
$ radeontop
```



※ 리눅스 커널이 자동으로 업데이트되는 경우 모듈이 올라오지 않을 수 있으므로 주의 할 것. 부팅시 지정된 커널로 부팅가능하게 하기 위해 설정파일(grub.cnf) 등을 수정해 주어야 함

- 환경 변수 설정. ROCm은 아래 경로에 설치된다. 환경변수를 설정해 준다.

```
# ls -la  /opt/rocm
lrwxrwxrwx 1 root root 22 Aug 20 07:29 /opt/rocm -> /etc/alternatives/rocm/

$ du -h /opt/rocm-6.2.0
[...]
27G     /opt/rocm-6.2.0

$ export ROCM_PATH=/opt/rocm-6.2.0
$ export PATH=$PATH:$ROCM_PATH/bin:$ROCM_PATH/llvm/bin
```

```
$ apt show rocm-libs
Package: rocm-libs
Version: 6.2.0.60200-66~24.04
Priority: optional
Section: devel
Maintainer: ROCm Dev Support <rocm-dev.support@amd.com>
Installed-Size: 13.3 kB
Depends: hipblas (= 2.2.0.60200-66~24.04), hipblaslt (= 0.8.0.60200-66~24.04), hipfft (= 1.0.14.60200-66~24.04), hipsolver (= 2.2.0.60200-66~24.04), hipsparse (= 3.1.1.60200-66~24.04), hiptensor (= 1.3.0.60200-66~24.04), miopen-hip (= 3.2.0.60200-66~24.04), half (= 1.12.0.60200-66~24.04), rccl (= 2.20.5.60200-66~24.04), rocalution (= 3.2.0.60200-66~24.04), rocblas (= 4.2.0.60200-66~24.04), rocfft (= 1.0.28.60200-66~24.04), rocrand (= 3.1.0.60200-66~24.04), hiprand (= 2.11.0.60200-66~24.04), rocsolver (= 3.26.0.60200-66~24.04), rocsparse (= 3.2.0.60200-66~24.04), rocm-core (= 6.2.0.60200-66~24.04), hipsparselt (= 0.2.1.60200-66~24.04), composablekernel-dev(= 1.1.0.60200-66~24.04), hipblas-dev (= 2.2.0.60200-66~24.04), hipblaslt-dev (= 0.8.0.60200-66~24.04), hipcub-dev (= 3.2.0.60200-66~24.04), hipfft-dev (= 1.0.14.60200-66~24.04), hipsolver-dev (= 2.2.0.60200-66~24.04), hipsparse-dev (= 3.1.1.60200-66~24.04), hiptensor-dev (= 1.3.0.60200-66~24.04), miopen-hip-dev (= 3.2.0.60200-66~24.04), rccl-dev (= 2.20.5.60200-66~24.04), rocalution-dev (= 3.2.0.60200-66~24.04), rocblas-dev (= 4.2.0.60200-66~24.04), rocfft-dev (= 1.0.28.60200-66~24.04), rocprim-dev (= 3.2.0.60200-66~24.04), rocrand-dev (= 3.1.0.60200-66~24.04), hiprand-dev (= 2.11.0.60200-66~24.04), rocsolver-dev (= 3.26.0.60200-66~24.04), rocsparse-dev (= 3.2.0.60200-66~24.04), rocthrust-dev (= 3.0.1.60200-66~24.04), rocwmma-dev (= 1.5.0.60200-66~24.04), hipsparselt-dev (= 0.2.1.60200-66~24.04)
Homepage: https://github.com/RadeonOpenCompute/ROCm
[...]
```

2.3 설치 확인

- 예제 프로그램은 ROCm 설치경로에 존재한다.

```
  예제 프로그램 경로를 홈디렉터리로 복사함
$ cp -R  $ROCM_PATH/share/openmp-extras/examples  $HOME/
$ cd $HOME/examples/openmp/veccopy
$ clang -O3 -fopenmp   veccopy.c -o veccopy
$ ./veccopy
Success
```

- Hello World with HIP. **hipcc** 를 이용하여 간단한 프로그램을 컴파일하고 확인. 디바이스 관련 API 문서를 참고[18)]

```
// hello.cpp
#include <hip/hip_runtime.h>
```

18) https://rocm.docs.amd.com/projects/HIP/en/docs-6.0.0/doxygen/html/group___device.html

```
#include <stdio.h>
int main(void)
{
  int count, device;
  hipGetDeviceCount(&count);
  hipGetDevice(&device);
  printf("Hello! I'm GPU %d out of %d GPUs in total.\n", device, count);
  return 0;
}
```

```
  컴파일 과정에서 나오는 경고(warning)을 무시하려면 -w 옵션을 사용
$ hipcc -w  hello.cp -o hello
$ ./hello
Hello! I'm GPU 0 out of 1 GPUs in total.
```

- 디바이스의 속성 정보를 알아내기 위한 프로그램

```
// hip_device_attributes.cpp
#include <hip/hip_runtime.h>
#include <stdio.h>
int main() {
  int deviceCount;
  hipError_t err = hipGetDeviceCount(&deviceCount);
  if (err != hipSuccess) { printf("hipGetDeviceCount failed: %s\n", hipGetErrorString(err)); re
turn 1; }
  if (deviceCount == 0) { printf("No HIP-capable devices found.\n"); return 1; }

  for (int dev = 0; dev < deviceCount; ++dev) {
    hipDeviceProp_t deviceProp;
    hipGetDeviceProperties(&deviceProp, dev);
    printf("Device %d: %s\n", dev, deviceProp.name);

    int major, minor;
    hipDeviceGetAttribute(&major, hipDeviceAttributeComputeCapabilityMajor, dev);
    hipDeviceGetAttribute(&minor, hipDeviceAttributeComputeCapabilityMinor, dev);
    printf("  Compute capability: %d.%d\n", major, minor);

    int multiProcessorCount;
    hipDeviceGetAttribute(&multiProcessorCount, hipDeviceAttributeMultiprocessorCount, dev);
    printf("  Number of multiprocessors: %d\n", multiProcessorCount);

    int maxThreadsPerBlock;
    hipDeviceGetAttribute(&maxThreadsPerBlock, hipDeviceAttributeMaxThreadsPerBlock, dev);
    printf("  Max threads per block: %d\n", maxThreadsPerBlock);

    int warpSize;
    hipDeviceGetAttribute(&warpSize, hipDeviceAttributeWarpSize, dev);
    printf("  Warp size: %d\n", warpSize);

    int memoryClockRate;
    hipDeviceGetAttribute(&memoryClockRate, hipDeviceAttributeMemoryClockRate, dev);
    printf("  Memory clock rate (kHz): %d\n", memoryClockRate);

    int memoryBusWidth;
    hipDeviceGetAttribute(&memoryBusWidth, hipDeviceAttributeMemoryBusWidth, dev);
    printf("  Memory bus width (bits): %d\n", memoryBusWidth);
```

```
    int l2CacheSize;
    hipDeviceGetAttribute(&l2CacheSize, hipDeviceAttributeL2CacheSize, dev);
    printf("  L2 cache size (bytes): %d\n", l2CacheSize);
  }
  return 0;
}
$ hipcc -w  hip_device_attributes.cpp  -o hip_device_attributes
$ ./hip_device_attributes
Device 0: AMD Radeon Pro V520
  Compute capability: 10.1
  Number of multiprocessors: 18
  Max threads per block: 1024
  Warp size: 32
  Memory clock rate (kHz): 1000000
  Memory bus width (bits): 2048
  L2 cache size (bytes): 4194304
```

- hipconfig는 HIP 환경 구성 정보를 표시하는 명령으로 perl 스크립트임. 옵션: --full 전체 정보를 표시. --path HIP 경로, --rocm-path ROCm 경로, --compiler 컴파일러 정보, --platform 플랫폼 정보.

```
$ hipconfig --full
HIP version: 6.2.41133-dd7f95766
==hipconfig
HIP_PATH           :/opt/rocm-6.2.0
ROCM_PATH          :/opt/rocm-6.2.0
HIP_COMPILER       :clang
HIP_PLATFORM       :amd
HIP_RUNTIME        :rocclr
CPP_CONFIG         : -D__HIP_PLATFORM_HCC__= -D__HIP_PLATFORM_AMD__= -I/opt/rocm-6.2.0/include
-I/include

==hip-clang
HIP_CLANG_PATH     :/opt/rocm-6.2.0/lib/llvm/bin
AMD clang version 18.0.0git (https://github.com/RadeonOpenCompute/llvm-project roc-6.2.0 24292
26466ce804ac523b398608f17388eb6d605a3f09)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /opt/rocm-6.2.0/lib/llvm/bin
Configuration file: /opt/rocm-6.2.0/lib/llvm/bin/clang++.cfg
AMD LLVM version 18.0.0git
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: znver2
  Registered Targets:
    amdgcn - AMD GCN GPUs
    r600   - AMD GPUs HD2XXX-HD6XXX
    x86    - 32-bit X86: Pentium-Pro and above
    x86-64 - 64-bit X86: EM64T and AMD64
hip-clang-cxxflags :
 -O3
hip-clang-ldflags :
--driver-mode=g++ -O3 --hip-link
[...]
```

3. HIP 프로그래밍

- HIP (Heterogeneous-Compute Interface for Portability)[19]은 GPU 프로그래밍을 위한 C++ 런타임 API 및 프로그래밍 언어임. ROCm 소프트웨어 플랫폼의 핵심 구성 요소.
- HIP 개발을 위해서는 다음과 같은 구성요소가 필요함: ROCm 소프트웨어 스택, HIP 런타임 및 컴파일러, 관련라이브러리(hipblas, biprand, hipdnn, hipfft 등)
- HIPify 도구는 CUDA 로 작성된 소스코드를 HIP 코드로 자동 변환 할 수 있음. 코드와 문법은 CUDA 와 유사함.

- HIP examples[20]을 참고한다.

```
$ git clone  https://github.com/ROCm/rocm-examples.git
```

3.1 메모리 관리

- 호스트와 디바이스간의 메모리 카피 전송 속도 테스트. hipMemcpy 함수[21]를 사용

```
// bandwidth.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <chrono>
#define HIP_CHECK(status) \
  if (status != hipSuccess) { \
    std::cerr << "HIP error: " << hipGetErrorString(status) << std::endl; \
    exit(1); \
  }
#ifndef  DSIZE
#define  DSIZE  1024*1024*1024       // 1GB
#endif
const size_t   DATA_SIZE =  DSIZE;   // 데이터 크기

int main() {
  float *h_data, *d_data;

  h_data = (float*)malloc(DATA_SIZE); // 호스트 메모리 할당
  HIP_CHECK(hipMalloc(&d_data, DATA_SIZE));  // 디바이스 메모리 할당
  std::cout << "Data Size: " << DATA_SIZE/1024/1024 << " MB" << std::endl;

  for (size_t i = 0; i < DATA_SIZE / sizeof(float); ++i) {
    h_data[i] = static_cast<float>(i);
  }

  // 호스트 -> 디바이스 전송 시간 측정
  auto start = std::chrono::high_resolution_clock::now();
  HIP_CHECK(hipMemcpy(d_data, h_data, DATA_SIZE, hipMemcpyHostToDevice));
  HIP_CHECK(hipDeviceSynchronize());
  auto end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double> h2d_time = end - start;
```

19) https://github.com/ROCm/HIP
20) https://github.com/ROCm/rocm-examples
21) https://rocm.docs.amd.com/projects/HIP/en/docs-6.0.0/doxygen/html/group___memory.html#gac1a055d288302edd641c6d7416858e1e

```
  // 디바이스 -> 호스트 전송 시간 측정
  start = std::chrono::high_resolution_clock::now();
  HIP_CHECK(hipMemcpy(h_data, d_data, DATA_SIZE, hipMemcpyDeviceToHost));
  HIP_CHECK(hipDeviceSynchronize());
  end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double> d2h_time = end - start;

  // 대역폭 계산 (GB/s)
  double h2d_bandwidth = DATA_SIZE / (1024.0 * 1024.0 * 1024.0) / h2d_time.count();
  double d2h_bandwidth = DATA_SIZE / (1024.0 * 1024.0 * 1024.0) / d2h_time.count();
  std::cout << "Host to Device Bandwidth: " << h2d_bandwidth << " GB/s" << std::endl;
  std::cout << "Device to Host Bandwidth: " << d2h_bandwidth << " GB/s" << std::endl;
  free(h_data);
  HIP_CHECK(hipFree(d_data));
  return 0;
}
  전송 데이터 크기를 변화시키면서 전송 속도를 측정. 디바이스 메모리는 8GB 임.
$ hipcc   -o bandwidth1m    bandwidth.hip  -DDSIZE=1048576   # 1MB
$ hipcc   -o bandwidth1g    bandwidth.hip  -DDSIZE=1073741824  # 1GB
$ hipcc   -o bandwidth2g    bandwidth.hip  -DDSIZE=2147483648  # 2GB
$ hipcc   -o bandwidth4g    bandwidth.hip  -DDSIZE=4294967296  # 4GB
$ hipcc   -o bandwidth7g    bandwidth.hip  -DDSIZE=7516192768  # 7GB
$ hipcc   -o bandwidth7.5g  bandwidth.hip  -DDSIZE=8053063680  # 7.5GB
$ hipcc   -o bandwidth8g    bandwidth.hip  -DDSIZE=8589934592   # 8GB (out of memory)
$ ./bandwidth1m
Data Size: 1024 MB
Host to Device Bandwidth: 1.52918 GB/s    H2D 속도가 느린 이유는 warm-up 효과때문임
Device to Host Bandwidth: 6.58159 GB/s
$ ./bandwidth2g
Data Size: 2048 MB
Host to Device Bandwidth: 2.25489 GB/s
Device to Host Bandwidth: 6.58185 GB/s
$ ./bandwidth4g
Data Size: 4096 MB
Host to Device Bandwidth: 3.25487 GB/s
Device to Host Bandwidth: 6.57474 GB/s
$ ./bandwidth7g
Data Size: 7168 MB
Host to Device Bandwidth: 4.10878 GB/s
Device to Host Bandwidth: 6.58196 GB/s
$ ./bandwidth7.5g
Data Size: 7680 MB
Host to Device Bandwidth: 4.0015 GB/s
Device to Host Bandwidth: 6.58171 GB/s
```

- 호스트 메모리를 디바이스로 복사후 다시 호스트로 복사하여 비교하는 프로그램

```
// copy.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <cstring>
#include <random>
#define SIZE  1048576   // 1024*1024
void print_data(int *data, int n) {
  for (int i = 0; i < 10; ++i) std::cout << data[i] << " "; // first 10 items
```

```
  std::cout << " ... ";
  for (int i = n-10; i < n; ++i) std::cout << data[i] << " "; // last 10 items
  std::cout << std::endl;
}
void initialize(int *data, int n) {
  std::mt19937 gen(std::time(0));
  std::uniform_int_distribution<> dis(1, 10000);  // 1~10000 사이의 랜덤 넘버
  for (int i = 0; i < n; ++i) {
    data[i] = dis(gen); // random distribution
  }
}
int  hostData[SIZE];
int  hostDataCopy[SIZE];
int main() {
  int *deviceData;
  int buffer_size = SIZE * sizeof(int);
  initialize(hostData, SIZE); // hostData 를 초기화 (랜덤데이터)
  print_data(hostData, SIZE);

  hipMalloc((void**)&deviceData, buffer_size); // 디바이스 메모리 할당
  hipMemcpy(deviceData, hostData, buffer_size, hipMemcpyHostToDevice); // host -> device
  hipMemcpy(hostDataCopy, deviceData, buffer_size, hipMemcpyDeviceToHost); // device -> host
  print_data(hostDataCopy, SIZE);
  bool isIdentical = (std::memcmp(hostData, hostDataCopy, buffer_size) == 0); // 동일한지 비교
  if (isIdentical) {
    std::cout << "Data matches!" << std::endl;
  } else {
    std::cout << "Data does not match!" << std::endl;
  }
  hipFree(deviceData);
  return 0;
}
$ hipcc -w  copy.hip  -o copy
$ ./copy
6649 6220 6446 1648 3014 6573 2461 8275 1072 5894  ... 2579 2884 554 2430 4161 8830 5902 5249 4675 9457
6649 6220 6446 1648 3014 6573 2461 8275 1072 5894  ... 2579 2884 554 2430 4161 8830 5902 5249 4675 9457
Data matches!
```

3.2 커널

- 커널 함수를 실행하고, 스레드의 인덱스 정보를 파악하기. **hipLaunchKernelGGL** 참고[22]. GGL은 Grid, Group, Launch를 의미함. 이 함수의 5개 매개 변수는 다음과 같다. ① 커널 함수 포인터 ② 그리드 차원(dim3 형식) ③ 블록 차원(dim3 형식) ④ 동적 공유 메모리 크기(바이트 단위) ⑤ 스트림. 이 다섯 개의 필수 매개변수 뒤에 커널 함수에 전달할 인자들이 따라옴. 이 함수는 CUDA의 구문과 유사하게 <<< >>> 구문으로로 표현할 수 있다.

  형식 : kernel<<<gridDim, blockDim, sharedMem, stream>>>(parameters);

- 커널 함수는 반드시 **void** 형으로 선언해야 함. 여러 개의 스레드가 커널 함수를 동시에 실행함. 각 스레드는 **block ID**와 **thread ID**를 이용하여 **global ID**를 계산 할 수 있음.
- 블록들은 동적으로 스케쥴링 되어 CU(compuing unit)에 할당됨. 한 블록의 모든 스레드는 같은 CU에서 실행되며, LDS와 L1 cache를 공유함. 블록내에서 실행되는 스레드를 64-wide chunk를 이루며 이

22) https://rocm.docs.amd.com/projects/HIP/en/latest/reference/cpp_language_extensions.html#kernel-launch-example

를 wavefronts 라고 함. wavefront들은 SIMD(single instruction multiple data) 연산의 단위가 됨. wavefront가 중단(stall)되는 경우 CU는 context switching을 통하여 다른 wavefront를 실행할 수 있음. 따라서 블록크기를 64의 배수로 하여 몇 개의 wavefront가 동시에 실행되도록 하는 것이 성능향상을 위해서 좋다.

- `__global__` 함수는 CPU에서 호출하며 GPU에서 실행됨. `__device__` 함수는 GPU에서만 호출 가능하고 GPU에서 실행됨. `__global__` 함수는 반환값을 가질수 없고, 재귀호출은 불가하며, 정적변수 및 가변 인수를 가질 수 없음.
- 커널을 실행하고 블록과 스레드의 정보를 파악하기 위한 예제 코드

```
// launchkernel.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <random>
//#define GRIDSIZE   10
//#define BLOCKSIZE  10
//#define N 100
struct MyStruct {   // 스레드 내부의 인덱스 값을 저장
  int idx;
  int thread_id;
  int block_id;
  int blockdim_id;
};
__global__ void myKernel(int *data, MyStruct *kinfo) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {   // 실제 실행 스레드의 수는 N보다 클수 있으나, N을 넘어가는 것은 의미가 없음
    kinfo[idx].idx = idx;  // 현재 스레드의 정보를 저장
    kinfo[idx].thread_id   = threadIdx.x;
    kinfo[idx].block_id    = blockIdx.x;
    kinfo[idx].blockdim_id = blockDim.x;
  }
}
[...]
int main() {
  int *hostData;
  int *deviceData;
  MyStruct *d_krnInfo;
  MyStruct *h_krnInfo;

  int data_buffer_size = N * sizeof(int);
  hostData = new int[N];
  hipMalloc((void**)&deviceData, data_buffer_size); // 디바이스 메모리 할당(data)
  hipMemcpy(deviceData, hostData, data_buffer_size, hipMemcpyHostToDevice); // host -> device

  h_krnInfo = new MyStruct[N];
  int krninfo_size = N * sizeof(struct MyStruct);
  hipMalloc((void**)&d_krnInfo, krninfo_size); // 디바이스 메모리 할당(kernel info)

  // Kernel 실행
  //myKernel<<<GRIDSIZE, BLOCKSIZE>>>(deviceData, d_krnInfo);
  hipLaunchKernelGGL(myKernel, GRIDSIZE, BLOCKSIZE, 0, 0, deviceData, d_krnInfo);
  hipDeviceSynchronize();

  hipMemcpy(h_krnInfo, d_krnInfo, krninfo_size, hipMemcpyDeviceToHost); // 호스트로 복사
```

```
  std::cout << "\tidx\t\tthread_id\tblock_id\tblockdim_id" << std::endl;
  for (int i = 0; i < N; i++) {
    std::cout << ""
       << "\t" << h_krnInfo[i].idx
       << "\t\t" << h_krnInfo[i].thread_id
       << "\t\t" << h_krnInfo[i].block_id
       << "\t\t" << h_krnInfo[i].blockdim_id
       << std::endl;
  }
  hipFree(deviceData);
  hipFree(d_krnInfo);
  return 0;
}
```

```
  블럭 0에 스레드를 3개 생성
$ hipcc -w launchkernel.hip  -o launchkernel -DGRIDSIZE=1 -DBLOCKSIZE=3 -DN=3 ; ./launchkernel
       idx             thread_id       block_id        blockdim_id
       0               0               0               3
       1               1               0               3
       2               2               0               3
  GRIDSIZE=2개 이므로, 블록 0,1이 생성됨. 각 블럭에서 3개 총 6개의 스레드를 실행
$ hipcc -w launchkernel.hip  -o launchkernel -DGRIDSIZE=2 -DBLOCKSIZE=3 -DN=6 ; ./launchkernel
       idx             thread_id       block_id        blockdim_id
       0               0               0               3
       1               1               0               3
       2               2               0               3
       3               0               1               3
       4               1               1               3
       5               2               1               3
  BLOCKSIZE 최대 값은 1024임. (hipDeviceAttributeMaxThreadsPerBlock)
$ hipcc -w launchkernel.hip -o launchkernel -DGRIDSIZE=2 -DBLOCKSIZE=1024 -DN=2048 ; ./launchke
rnel
       idx             thread_id       block_id        blockdim_id
       0               0               0               1024
       1               1               0               1024
[...]
       1022            1022            0               1024
       1023            1023            0               1024
       1024            0               1               1024
       1025            1               1               1024
       1026            2               1               1024
[...]
       2047            1023            1               1024
```

※ 블록사이즈 최대값 1024는 **rocminfo** 에서 workgroup max size에 해당한다.

```
$ rocminfo ¦ grep "Workgroup Max Size"
  Workgroup Max Size:      1024(0x400)
  Workgroup Max Size per Dimension:
```

※ **hipLaunchKernelGGL**은 매크로이며, 아래 헤더 파일에 정의된다.

```
// /opt/rocm-6.2.0/include/hip/amd_detail/amd_hip_runtime.h
[...]
#if defined(HIP_TEMPLATE_KERNEL_LAUNCH)
template <typename... Args, typename F = void (*)(Args...)>
```

```
void hipLaunchKernelGGL(F kernel, const dim3& numBlocks, const dim3& dimBlocks,
                        std::uint32_t sharedMemBytes, hipStream_t stream, Args... args) {
    constexpr size_t count = sizeof...(Args);
    auto tup_ = std::tuple<Args...>{args...};
    auto tup = validateArgsCountType(kernel, tup_);
    void* _Args[count];
    pArgs<0>(tup, _Args);

    auto k = reinterpret_cast<void*>(kernel);
    hipLaunchKernel(k, numBlocks, dimBlocks, _Args, sharedMemBytes, stream);
}
#else
#define hipLaunchKernelGGLInternal(kernelName, numBlocks, numThreads, memPerBlock, streamId, ...)  \
    do {                                                                                           \
        kernelName<<<(numBlocks), (numThreads), (memPerBlock), (streamId)>>>(__VA_ARGS__);         \
    } while (0)

#define hipLaunchKernelGGL(kernelName, ...)  hipLaunchKernelGGLInternal((kernelName), __VA_ARGS__)
#endif
```

- 전치행열(matrix transpose) 연산. 블록사이즈와 그리드사이즈 차원을 2차원으로 설정한 예제

```
// transpose.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <cstring>
#include <random>
#define SIZE  8
//#define SIZE  32
[...]
__global__ void matrix_transpose_kernel(float* out, const float* in, const unsigned int width)
{
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  out[y * width + x] = in[x * width + y];
}
void print_matrix(float *data, int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      std::cout.width(4);
      std::cout << data[i*n + j] << "  ";
    }
    std::cout << std::endl;
  }
}
void initialize(float *data, int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      data[i * n + j] = i - j;
    }
  }
}

float h_A[SIZE][SIZE]; // h_A --> d_A --> transpose --> d_B --> h_B
float h_B[SIZE][SIZE];
int main() {
```

```
  float *d_A;
  float *d_B;
  int buffer_size = SIZE * SIZE * sizeof(float);
  int width = SIZE;

  initialize((float *)h_A, width); // 매트릭스 초기화
  std::cout << "A:" << std::endl;
  print_matrix(&h_A[0][0], width); // 매트릭스 출력(A)

  hipMalloc((void**)&d_A, buffer_size); // 디바이스 메모리 할당
  hipMemcpy(d_A, h_A, buffer_size, hipMemcpyHostToDevice); // host -> device
  hipMalloc((void**)&d_B, buffer_size); // 디바이스 메모리 할당

  constexpr unsigned int threads_per_block_x = 8;
  constexpr unsigned int threads_per_block_y = 8;
// GRIDSIZE=(1,1) BLOCKSIZE=(8,8)
  matrix_transpose_kernel<<<dim3(width / threads_per_block_x, width / threads_per_block_y),
                            dim3(threads_per_block_x, threads_per_block_y),
                            0,
                            hipStreamDefault >>>(d_B, d_A, width);
  HIP_CHECK(hipGetLastError());

  hipMemcpy(h_B, d_B, buffer_size, hipMemcpyDeviceToHost); // device -> host
  std::cout << "B = transpose A:" << std::endl;
  print_matrix(&h_B[0][0], SIZE); // 매트릭스 출력(B)

  hipFree(d_A);
  return 0;
}
```

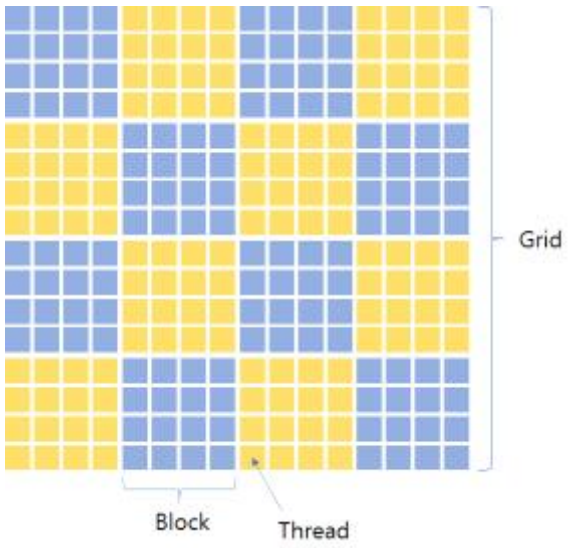
```
$ hipcc -w transpose.hip -o transpose ; ./transpose
A:
   0    -1    -2    -3    -4    -5    -6    -7
   1     0    -1    -2    -3    -4    -5    -6
   2     1     0    -1    -2    -3    -4    -5
   3     2     1     0    -1    -2    -3    -4
   4     3     2     1     0    -1    -2    -3
   5     4     3     2     1     0    -1    -2
   6     5     4     3     2     1     0    -1
   7     6     5     4     3     2     1     0
B = transpose A:
   0     1     2     3     4     5     6     7
  -1     0     1     2     3     4     5     6
  -2    -1     0     1     2     3     4     5
  -3    -2    -1     0     1     2     3     4
  -4    -3    -2    -1     0     1     2     3
  -5    -4    -3    -2    -1     0     1     2
  -6    -5    -4    -3    -2    -1     0     1
  -7    -6    -5    -4    -3    -2    -1     0
```

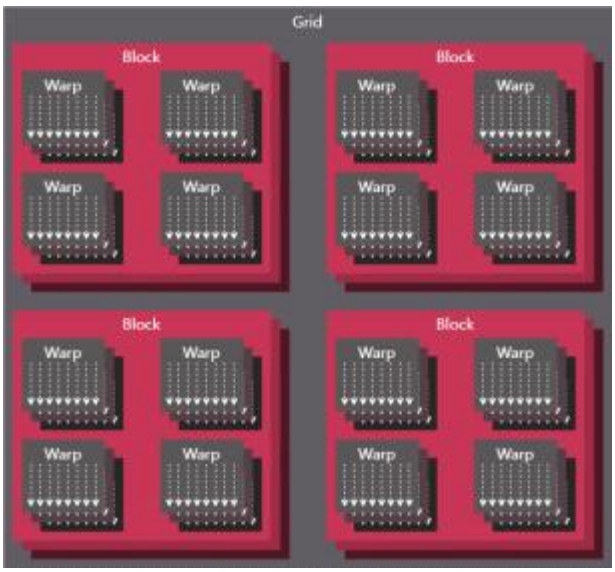
※ dim3는 그리드와 블럭의 크기를 나타내기 위한 3차원 정수 벡터임. 별도로 명시하지 않을 경우 각 차원은 1로 초기화 된다.

```
// hip_runtime_api.h
typedef struct dim3 {
    uint32_t x;  ///< x
    uint32_t y;  ///< y
```

```
    uint32_t z;  ///< z
#ifdef __cplusplus
    constexpr __host__ __device__ dim3(uint32_t _x = 1, uint32_t _y = 1, uint32_t _z = 1) : x(_x), y(_y),
z(_z){};
#endif
} dim3;
```

- 커널은 그리드(grid) 에서 실행됨. GPU는 1D, 2D, 또는 3D 그리드를 지원함. 그리드는 동일한 크기의 블록(block)으로 나누어짐. 각 블럭은 스레드(thread)로 구성됨.





- 용어

* 워프(warp) 또는 웨이브프런트(wavefront): 가장 안쪽의 스레드 그룹을 워프(warp) 또는 ISA 용어로 파면(wavefron)이라고 함. 워프의 크기는 아키텍쳐에 따라 달라지며 항상 고정됨.
* 블록(block) : 중간 크기의 그룹을 블록 또는 스레드 블록(thread block)이라고 함. 블록의 모든 스레드가 메모리 인스턴스를 공유하여 데이터를 공유하거나 서로 동기화 하는데 사용할 수 있음. 블록의 크기는 사용자가 구성할 수 있지만, 하드웨어의 기능에 따라 제한됨.
* 그리드(grid): 가장 바깥쪽 그룹을 그리드 라고 함. 그리드는 하나의 커널 함수가 GPU에서 실행되는 전체 작업 단위를 나타냄. 커널이 호출될때, 그리드의 구조에 따라 작업이 GPU의 여러 실행 유닛에 분배됨. GPU 하드웨어는 그리드 단위로 작업을 스케쥴링 하고 관리함.

※ AMD GPU의 GCN 아키텍쳐에서는 64개의 작업 아이템으로 구성된 wavefront를 사용하여 SIMD 엔진들이 한번에 하나의 wavefront를 처리함. RDNA 아키텍처에서는 32스레드(Wave32)까지 1클럭 사이클에 처리가능하며 64스레드(Wave64)를 처리하려면 2클럭 사이클이 필요했음. RDNA 3 아키텍처에서는 Wave64 도 1 클럭 사이클에 처리가능하게 됨

3.3 성능 측정

- SAXPY(Single precision A X plus Y, 스칼라값을 곱한 벡터와 벡터의 덧셈) 예제 프로그램

```
// saxpy.hip
#include <hip/hip_runtime.h>
#include <iostream>
#define HIP_CHECK(status) \
  if (status != hipSuccess) { \
```

```
  std::cerr << "HIP error: " << hipGetErrorString(status) << std::endl; \
  exit(1); \
  }
__global__ void saxpy_kernel(const float a, const float* d_x, float* d_y, const unsigned int si
ze)
{
  const unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < size) {
    d_y[idx] = a * d_x[idx] + d_y[idx];
  }
}
void initialize(float *vec, int n, float value) {
  for (int i = 0; i < n; ++i) {
    vec[i] = value;
  }
}
void print_data(float *data, int n) {
  for (int i = 0; i < 10; ++i) std::cout << data[i] << " "; // first 10 items
  std::cout << " ... ";
  for (int i = n-10; i < n; ++i) std::cout << data[i] << " "; // last 10 items
  std::cout << std::endl;
}
int main() {
  constexpr unsigned int size = 1000000; // 벡터의 크기
  constexpr size_t size_bytes = size * sizeof(float); // 벡터의 메모리 용량
  constexpr unsigned int block_size = 256; // 블록당 스레드 개수
  constexpr unsigned int grid_size = (size + block_size - 1) / block_size; // 블럭의 개수 ceil
(size/block_size).
  constexpr float a = 3.14f; // 상수 a (Y += a*X)
  float *h_x, *h_y;
  h_x = new float[size];
  h_y = new float[size];
  initialize(h_x, size, 2.0);
  initialize(h_y, size, 1.0);
  print_data(h_x, size);
  print_data(h_y, size);
  // Allocate and copy vectors to device memory.
  float* d_x;
  float* d_y;
  HIP_CHECK(hipMalloc(&d_x, size_bytes));
  HIP_CHECK(hipMalloc(&d_y, size_bytes));
  HIP_CHECK(hipMemcpy(d_x, h_x, size_bytes, hipMemcpyHostToDevice));
  HIP_CHECK(hipMemcpy(d_y, h_y, size_bytes, hipMemcpyHostToDevice));
  std::cout << "Calculating y[i] = a * x[i] + y[i] over " << size << " elements." << std::endl;
  // Launch the kernel on the default stream.
  saxpy_kernel<<<dim3(grid_size), dim3(block_size), 0, hipStreamDefault>>>(a, d_x, d_y, size);
  HIP_CHECK(hipGetLastError());
  HIP_CHECK(hipMemcpy(h_y, d_y, size_bytes, hipMemcpyDeviceToHost)); // device->host
  print_data(h_y, size);
  HIP_CHECK(hipFree(d_x));
  HIP_CHECK(hipFree(d_y));
  return 0;
}
$ hipcc -w saxpy.hip  -o saxpy
```

```
$ ./saxpy                 3.14(a) * 2(x) + 1(y) ===> 7.28
2 2 2 2 2 2 2 2 2 2  ... 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1  ... 1 1 1 1 1 1 1 1 1 1
Calculating y[i] = a * x[i] + y[i] over 1000000 elements.
7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28  ... 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28
```

- 커널 실행부분을 포함한 데이터 전송에 걸리는 시간을 측정하기. C++의 표준 라이브러리인 std::chrono 를 이용

```
// saxpy2.hip
#include <chrono>
//#define NSIZE 100000
[...]
int main()
{
  constexpr unsigned int size = NSIZE; // 벡터의 크기
[...]

  auto t0start = std::chrono::high_resolution_clock::now(); // 시간측정 시작
  HIP_CHECK(hipMemcpy(d_x, h_x, size_bytes, hipMemcpyHostToDevice));  // host->device
  HIP_CHECK(hipMemcpy(d_y, h_y, size_bytes, hipMemcpyHostToDevice));  // host->device

  std::cout << "Calculating y[i] = a * x[i] + y[i] over " << size << " elements." << std::endl;

  // Launch the kernel on the default stream.
  auto t1start = std::chrono::high_resolution_clock::now(); // 시간측정 시작
  saxpy_kernel<<<dim3(grid_size), dim3(block_size), 0, hipStreamDefault>>>(a, d_x, d_y, size);
  HIP_CHECK(hipGetLastError());
  auto t1end = std::chrono::high_resolution_clock::now(); // 시간측정 종료
  std::chrono::duration<double> t1 = t1end - t1start;
  std::cout << "t1:" << t1.count() << " s" << std::endl; // t1은 커널만 실행한 시간

  HIP_CHECK(hipMemcpy(h_y, d_y, size_bytes, hipMemcpyDeviceToHost)); // device->host
  auto t0end = std::chrono::high_resolution_clock::now(); // 시간측정 종료
  std::chrono::duration<double> t0 = t0end - t0start;
  std::cout << "t0:" << t0.count() << " s" << std::endl; //  t0는 데이터 이동을 포함한 시간
[...]
```

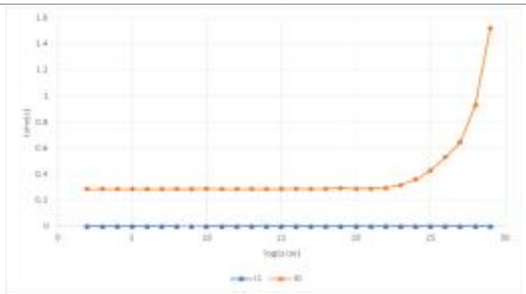
```
$ hipcc -w saxpy2.hip  -o saxpy2  -DNSIZE=100000 ; ./saxpy2
2 2 2 2 2 2 2 2 2 2  ... 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1  ... 1 1 1 1 1 1 1 1 1 1
Calculating y[i] = a * x[i] + y[i] over 100000 elements.
t1:0.000458439 s
t0:0.288491 s
7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28  ... 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28 7.28
```

```
# loop_saxpy2.py   # 실행을 위한 파이썬 스크립트
value = 4
while value <= 1024*1024*1024:
    cmd = "hipcc -w saxpy2.hip  -o saxpy2  -DNSIZE=%d ; ./saxpy2" % (value);
    print(cmd)
    value *= 2
```

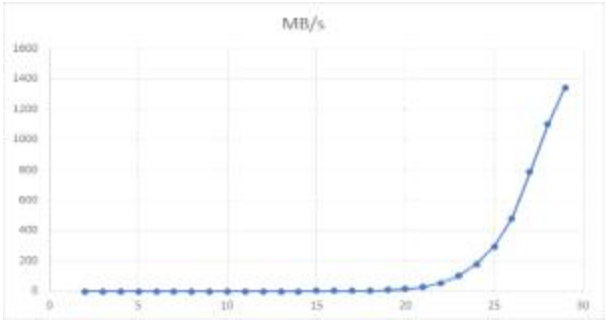
- 벡터의 크기에 따라 측정된 실행 시간을 도식화 하면 다음과 같다. 벡터의 크기에 따라 커널 실행시간(t1)은 큰 변화가 없으나, log(NSIZE)>24 (64MB) 이상에서부터는 데이터 전송 시간 변화가 나타남. float 타입의 4바이트 이므로, NSIZE=2^29(536870912) 일때 t0=1.5219. 따라서 커널 계산 시간은 무

시하고, 전송 bandwidth만 계산하면 4*536870912/1.52/1024/1024=1.347GB/s 로 계산된다. 프로그램에서데이터 전송은 host2device 로 2번, device2host로 1번 전송됨.

NSIZE	log(NSIZE)	t1	t0
4	2	0.00051062	0.28808
8	3	0.0004845	0.287025
16	4	0.0004694	0.287256
32	5	0.00046257	0.288349
64	6	0.00045926	0.288467
128	7	0.000500051	0.288771
256	8	0.0005048	0.288051
512	9	0.000413889	0.286994
1024	10	0.000524021	0.289624
2048	11	0.00047312	0.287821
4096	12	0.00047655	0.287368
8192	13	0.000453609	0.287305
16384	14	0.000452859	0.287483
32768	15	0.00048777	0.288653
65536	16	0.00048682	0.289252
131072	17	0.000526641	0.288083
262144	18	0.00048016	0.290055
524288	19	0.00047672	0.295697
1048576	20	0.00045929	0.29086
2097152	21	0.00048279	0.290761
4194304	22	0.00046369	0.295091
8388608	23	0.000511111	0.317639
16777216	24	0.000586322	0.361359
33554432	25	0.000526411	0.434073
67108864	26	0.000602603	0.533113
134217728	27	0.000624013	0.650244
268435456	28	0.000585053	0.929531
536870912	29	0.000595922	1.5219



데이터 전송 포함 소요 시간(t0)



대역폭(전송크기/시간) MB/s

- SGEMV는 단정밀도의 일반 행렬-벡터 곱셈(General Matrix-Vector Multiplication) 연산을 수행함. BLAS Level2에 해당한다.

```
// sgemv.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <chrono>
#define BLOCKSIZE 1024
#define ROWS 16
#define COLS 16
[...]
__global__ void sgemv_kernel(int m, int n, const float alpha, const float* A, int lda,
                             const float* x, int incx, const float beta, float* y, int incy) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < m) {
    float dot = 0.0f;
    for (int j = 0; j < n; ++j) {
      dot += A[i * lda + j] * x[j * incx];
    }
    y[i * incy] += alpha * dot + beta * y[i * incy];
  }
}

// y += alpha * A * x + beta * y
void cpu_sgemv(int m, int n, const float alpha, const float *A, int lda,
        const float *x, int incx,
```

```
        const float beta, float *y, int incy) {
  for (int i = 0; i < m; i++) {
    float dot = 0.0f;
    for (int j = 0; j < n; j++) {
      dot += A[i * lda + j] * x[j * incx];
    }
    y[i * incy] += alpha * dot + beta * y[i * incy];
  }
}
[...]
int main() {
  const int rows = ROWS; const int cols = COLS;
  float *h_A = new float[rows * cols];
  float *h_x = new float[cols];
  float *h_y = new float[rows];
  float *h_y2 = new float[rows];
  float alpha = 2.0f;
  float beta = 10.0f;
  int lda = cols;   // leading dimension of A (row-major 에서는 보통 열의 수와 같음)
  int incx = 1, incy = 1;

  initialize_matrix(h_A, rows, cols);
  initialize_vector(h_x, cols);
  initialize_vector(h_y, cols);
  initialize_vector(h_y2, cols);
  print_matrix("A", h_A, rows, cols);
  print_vec("x", h_x, cols);
  print_vec("y", h_y, cols);
  print_vec("y2", h_y2, cols);
  std::cout << " incx: " << incx << "  incy: " << incy
          << " alpha: " << alpha << "  beta: " << beta << std::endl;

  // cpu에서 계산 (결과는 h_y2 에 저장)
  cpu_sgemv(rows, cols, alpha, h_A, lda, h_x, incx, beta, h_y2, incy);
  std::cout << "cpu result : " << std::endl;
  print_vec("", h_y2, rows);

  // allocate device memory
  float *d_A, *d_x, *d_y;
  HIP_CHECK(hipMalloc(&d_A, rows * cols * sizeof(float)));
  HIP_CHECK(hipMalloc(&d_x, cols * sizeof(float)));
  HIP_CHECK(hipMalloc(&d_y, rows * sizeof(float)));
  HIP_CHECK(hipMemcpy(d_A, h_A, rows * cols * sizeof(float), hipMemcpyHostToDevice));
  HIP_CHECK(hipMemcpy(d_x, h_x, cols * sizeof(float), hipMemcpyHostToDevice));
  HIP_CHECK(hipMemcpy(d_y, h_y, rows * sizeof(float), hipMemcpyHostToDevice));

  dim3 blockSize(BLOCKSIZE);
  dim3 gridSize((rows + blockSize.x - 1) / blockSize.x);
  std::cout << "  blockSize.x: " << blockSize.x
            << "  gridSize.x : " << gridSize.x
            << "  rows: " << rows << std::endl;

  sgemv_kernel<<<gridSize, blockSize, 0, 0>>>
      (rows, cols, alpha, d_A, lda, d_x, incx, beta, d_y, incy);
```

```
            결과는 h_y 에 저장
  HIP_CHECK(hipMemcpy(h_y, d_y, rows * sizeof(float), hipMemcpyDeviceToHost));
  std::cout << "gpu result : " << std::endl;
  print_vec("", h_y, rows);
        h_y와 h_y2 를 비교
  if (compare_vector(h_y, h_y2, rows)) std::cout << "Varification sucess!" << std::endl;
  else std::cout << "Varification FAIL!!!!!" << std::endl;

  HIP_CHECK(hipFree(d_A));
  HIP_CHECK(hipFree(d_x));
  HIP_CHECK(hipFree(d_y));
  return 0;
}
$ hipcc -w sgemv.hip  -o sgemv ; ./sgemv
A
   0    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
   1    2    1    1    1    1    1    1    1    1    1    1    1    1    1    1
   1    1    4    1    1    1    1    1    1    1    1    1    1    1    1    1
   1    1    1    6    1    1    1    1    1    1    1    1    1    1    1    1
   1    1    1    1    8    1    1    1    1    1    1    1    1    1    1    1
   1    1    1    1    1   10    1    1    1    1    1    1    1    1    1    1
   1    1    1    1    1    1   12    1    1    1    1    1    1    1    1    1
   1    1    1    1    1    1    1   14    1    1    1    1    1    1    1    1
   1    1    1    1    1    1    1    1   16    1    1    1    1    1    1    1
   1    1    1    1    1    1    1    1    1   18    1    1    1    1    1    1
   1    1    1    1    1    1    1    1    1    1   20    1    1    1    1    1
   1    1    1    1    1    1    1    1    1    1    1   22    1    1    1    1
   1    1    1    1    1    1    1    1    1    1    1    1   24    1    1    1
   1    1    1    1    1    1    1    1    1    1    1    1    1   26    1    1
   1    1    1    1    1    1    1    1    1    1    1    1    1    1   28    1
   1    1    1    1    1    1    1    1    1    1    1    1    1    1    1   30
x [ 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  ]
y [ 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  ]   y <---> y2가 동일함
y2 [ 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  ]
 incx: 1  incy: 1 alpha: 2  beta: 10
cpu result :
 [ 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101  ]
  blockSize.x: 1024  gridSize.x : 1  rows: 16
gpu result :
 [ 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101  ]
Varification sucess!
```

3.4 공유 메모리

- __shared__ 로 선언된 공유 메모리 변수는 LDS(local data store) 공유 메모리에 할당됨. 같은 블록의 모든 스레드가 공유함. 글로벌 메모리 보다 빠른 접근이 가능. 블록단위의 동기화를 위해서 __syncthreads() 함수[23]를 사용함. 공유 변수는 블록 내에서만 공유되며, 서로 다른 블록 간에는 공유되지 않음. 커널을 분할하여 여러 커널 호출로 작업을 분할할 경우 중간 결과를 글로벌 메모리에 저장해야 함.

```
// shared.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <chrono>
```

23) https://rocm.docs.amd.com/projects/HIP/en/latest/reference/cpp_language_extensions.html#synchronization-functions

```
[...]   매트릭스와 백터의 곱셈 연산으로 커널1은 글로벌 메모리만 사용, 커널2는 공유메모리를 사용
__global__ void matrixVectorMulKernel1 (float* matrix, float* vector, float* result, int rows,
int cols) {   // 커널1은 글로벌 메모리만 사용 (공유메모리 사용x)
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if (row < rows) {
    float sum = 0.0f;
    for (int col = 0; col < cols; ++col) {
      sum += matrix[row * cols + col] * vector[col];
    }
    result[row] = sum;
  }
}

template<unsigned int Rows>
__global__ void matrixVectorMulKernel2 (float* matrix, float* vector, float* result, const int
rows, const int cols) {   // 커널2는 공유 메모리를 사용
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  //__shared__ float  s_vector[Rows];  // static shared memory
  extern __shared__ float  s_vector[];  // dynamic shared memory
  if (row < rows) s_vector[row] = vector[row]; // 공유메모리에 벡터를 복사한다.
  __syncthreads();
  if (row < rows) {
    float sum = 0.0f;
    for (int col = 0; col < cols; ++col) {
      sum += matrix[row * cols + col] * s_vector[col];
    }
    result[row] = sum;
  }
}
void cpuMatrixVectorMul(const float* matrix, const float* vector, float* result, int rows, int
cols) {   // cpu 에서 계산
  for (int i = 0; i < rows; ++i) {
    float sum = 0.0f;
    for (int j = 0; j < cols; ++j) {
      sum += matrix[i * cols + j] * vector[j];
    }
    result[i] = sum;
  }
}
void initialize_matrix(float *matrix, int rows, int cols) {
  float v;
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      if (i == j) v = i + j; else v = 1;
      matrix[i*rows+j] = v;
    }
  }
}
void initialize_vector(float *vec, int n) {
  for (int i = 0; i < n; i++) {
    vec[i] = 1.0;
  }
}
[...]
```

```
bool compare_vector(float *v1, float *v2, int n) { // 결과 검증을 위해 두 벡터를 비교
  float diff;
  constexpr float eps    = 1.0E-6f; // 허용 오차 epsilon
  for (int i = 0; i < n; ++i) {
    diff = std::fabs(v1[i] - v2[i]);
    if (diff > eps) {
      std::cout << "diff : " << diff << std::endl;
      return false;
    }
  }
  return true; // equal
}
int main()  {
[...]
  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  std::chrono::duration<double> t1;
  // cpu에서 계산 (h_result2)
  start = std::chrono::high_resolution_clock::now(); // 시간측정 시작
  cpuMatrixVectorMul(h_matrix, h_vector, h_result2, rows, cols);
  end = std::chrono::high_resolution_clock::now(); // 시간측정 종료
  t1 = end - start;
  std::cout << "cpu execution time:" << t1.count() << " s" << std::endl;

  // allocate device memory
  float *d_matrix, *d_vector, *d_result;
  HIP_CHECK(hipMalloc(&d_matrix, rows * cols * sizeof(float)));
  HIP_CHECK(hipMalloc(&d_vector, cols * sizeof(float)));
  HIP_CHECK(hipMalloc(&d_result, rows * sizeof(float)));

  // 호스트에서 디바이스로 데이터 복사
  HIP_CHECK(hipMemcpy(d_matrix, h_matrix, rows * cols * sizeof(float), hipMemcpyHostToDevice));
  HIP_CHECK(hipMemcpy(d_vector, h_vector, cols * sizeof(float), hipMemcpyHostToDevice));

  dim3 blockSize(BLOCKSIZE); // 블럭크기는 매크로에서 입력됨
  dim3 gridSize((rows + blockSize.x - 1) / blockSize.x);
  std::cout << "blockSize.x : " << blockSize.x << std::endl;
  std::cout << "gridSize.x  : " << gridSize.x << std::endl;
  std::cout << "rows : " << rows << std::endl;

  // kernel1 (글로벌 메모리)
  start = std::chrono::high_resolution_clock::now(); // 시간측정 시작
  matrixVectorMulKernel1<<<gridSize, blockSize, 0, 0>>>(d_matrix, d_vector, d_result, rows, col
s); // 커널1 실행
  end = std::chrono::high_resolution_clock::now(); // 시간측정 종료
  t1 = end - start;
  std::cout << "kernel1 execution time:" << t1.count() << " s" << std::endl;

  // 디바이스에서 호스트로 결과 복사
  HIP_CHECK(hipMemcpy(h_result, d_result, rows * sizeof(float), hipMemcpyDeviceToHost));
  print_vec(h_result, rows);
  if (compare_vector(h_result, h_result2, rows)) std::cout << "Varification sucess!" << std::en
dl;
  else std::cout << "Varification FAIL!!!!!" << std::endl;
```

```
  // kernel2 (공유메모리)
  size_t sharedMemSize = rows * sizeof(float);
  start = std::chrono::high_resolution_clock::now(); // 시간측정 시작
  matrixVectorMulKernel2<rows>    // 커널2 실행
    <<<gridSize, blockSize, sharedMemSize>>>(d_matrix, d_vector, d_result, rows, cols);
  end = std::chrono::high_resolution_clock::now(); // 시간측정 종료
  t1 = end - start;
  std::cout << "kernel2 execution time:" << t1.count() << " s" << std::endl;

  // 디바이스에서 호스트로 결과 복사
  HIP_CHECK(hipMemcpy(h_result, d_result, rows * sizeof(float), hipMemcpyDeviceToHost));
  print_vec(h_result, rows);
  if (compare_vector(h_result, h_result2, rows)) std::cout << "Varification sucess!" << std::en
dl;
  else std::cout << "Varification FAIL!!!!!" << std::endl;
[...]
$ hipcc -w shared.hip  -o shared   -DROWS=1024 -DCOLS=1024 -DBLOCKSIZE=256 ; ./shared
cpu execution time:0.00093915 s
blockSize.x : 256
gridSize.x  : 4     블록의 개수는 4개
rows : 1024
kernel1 execution time:0.000518941 s
Varification sucess!    커널1은 글로벌 메모리를 사용하므로 검증에 성공
kernel2 execution time:4.05e-06 s
Varification FAIL!!!!!  4개의 블록 간에는 메모리 공유가 안되므로, 검증에 실패함

$ hipcc -w shared.hip  -o shared   -DROWS=1024 -DCOLS=1024 -DBLOCKSIZE=512 ;  ./shared
cpu execution time:0.00093917 s
blockSize.x : 512
gridSize.x  : 2      블록의 개수는 2개
rows : 1024
kernel1 execution time:0.000542591 s
Varification sucess!
kernel2 execution time:4.69e-06 s
Varification FAIL!!!!!   <<< 같은 이유로 검증에 실패함

$ hipcc -w shared.hip  -o shared   -DROWS=1024 -DCOLS=1024 -DBLOCKSIZE=1024 ; ./shared
cpu execution time:0.00093975 s
blockSize.x : 1024
gridSize.x  : 1     블록의 개수는 1개
rows : 1024
kernel1 execution time:0.000531072 s
Varification sucess!
kernel2 execution time:4.65e-06 s
Varification sucess!       모든 스레드가 같은 블럭에서 실행되므로 공유 메모리가 정상 작동함
```

※ 로컬 공유 메모리는 64KB 까지만 허용된다. 커널코드에서 주석 처리한 부분을 해제하고 Rows 값을 크게 했을 경우 다음과 같은 에러가 발생함

```
// matrixVectorMulKernel2 에서
  __shared__ float  s_vector[Rows];  // static shared memory
$ hipcc -w matvec.hip  -o matvec -DROWS=32768 -DCOLS=32768
matvec.hip:27:17: error: local memory (131072) exceeds limit (65536) in 'void matrixVectorMulKe
rnel2<32768u>(float*, float*, float*, int, int)'
```

※ 첫번째 커널 실행시 실행 시간이 많이 걸리는 것은 프로그램의 cold start 또는 warm-up 효과 때문으로 커널1과 커널2의 실행 순서를 바꾸어 테스트해 보아도 항상 첫번째 실행시 시간이 오래 걸리는 것을 알 수 있음.

※ 오브젝트 코드에 내장되어 있는 프로그램 바이너리를 `roc-obj-ls` 유틸리티를 이용하여 조회할 수 있다. object code version 4에 해당하는 커널을 포함하고 있음을 알 수 있다. LLVM 타겟[24]은 `<architecture>-<vendor>-<os>-<environment>` 임. `roc-obj -d (disassemble)` 옵션을 통하여 역어셈블링 할 수 있다.

```
$ roc-obj-ls  -v  ./shared
Bundle# Entry ID:                                URI:
1       host-x86_64-unknown-linux--              file://./shared#offset=8192&size=0
1       hipv4-amdgcn-amd-amdhsa--gfx1011         file://./shared#offset=8192&size=6880
$ roc-obj  -d  ./shared
$ ls -la shared:*
-rw-rw-r-- 1 ubuntu ubuntu  6880 Aug 28 08:01  shared:1.hipv4-amdgcn-amd-amdhsa--gfx1011
-rw-rw-r-- 1 ubuntu ubuntu 18526 Aug 28 08:01  shared:1.hipv4-amdgcn-amd-amdhsa--gfx1011.s
$ cat shared:1.hipv4-amdgcn-amd-amdhsa--gfx1011.s
<stdin>:       file format elf64-amdgpu
Disassembly of section .text:
0000000000002000 <_Z22matrixVectorMulKernel1PfS_S_ii>:
        s_clause 0x1                                    // 000000002000: BFA10001
        s_load_dword s0, s[4:5], 0x2c                   // 000000002004: F4000002 FA00002C
        s_load_dwordx2 s[8:9], s[4:5], 0x18             // 00000000200C: F4040202 FA000018
        s_waitcnt lgkmcnt(0)                            // 000000002014: BF8CC07F
        s_and_b32 s0, s0, 0xffff                        // 000000002018: 8700FF00 0000FFFF
        v_mad_u64_u32 v[0:1], s0, s6, s0, v[0:1]        // 000000002020: D5760000 04000006
        v_cmp_gt_i32_e32 vcc_lo, s8, v0                 // 000000002028: 7D080008
        s_and_saveexec_b32 s0, vcc_lo                   // 00000000202C: BE803C6A
        s_cbranch_execz 42                              // 000000002030: BF88002A <_Z22matrixVectorMulKe
rnel1PfS_S_ii+0xdc>
        s_clause 0x1                                    // 000000002034: BFA10001
        s_load_dwordx4 s[0:3], s[4:5], null             // 000000002038: F4080002 FA000000
        s_load_dwordx2 s[6:7], s[4:5], 0x10             // 000000002040: F4040182 FA000010
[...]
```

- 커널의 assembly code에서 접두어의 의미

  `s_* 로 시작하는 명령: scala unit instructions`
  `v_* 로 시작하는 명령: SIMD unit instructions`
  `global_* : global memory load/store`
  `ds_* : LDS memory load/store`

3.5 스트림

- HIP에서 스트림(stream)은 연속된 작업의 큐임. kernel, memcpy, event등의 작업이 큐를 구성한다. 스트림내의 작업들은 순서대로(first-enqueue first-complete) 실행된다. 서로 다른 스트림에서 실행되는 작업들은 오버랩이 가능하여 디바이스 자원에 동시에 접근이 가능하다.
- `hipStream_t` 타입과 `hipStreamCreate()` 으로 스트림을 생성함. `0` 또는 `NULL`은 '널(NULL) 스트림'을 의미하며, 널스트림에 들어간 작업들은 다른 모든 스트림의 작업이 모두 종료된 이후 실행된다. `hipMemcpy`와 같은 블록킹 함수는 널스트림에서 실행됨.

24) https://www.llvm.org/docs/AMDGPUUsage.html#target-triples

```
// streams.hip
#include <hip/hip_runtime.h>
#include <iostream>
#include <vector>
[...]
template<unsigned int Width>
__global__ void matrix_transpose_static_shared(float* out, const float* in)
{
    __shared__ float shared_mem[Width * Width]; // Shared memory of constant size
    const unsigned int x = blockDim.x * blockIdx.x + threadIdx.x;
    const unsigned int y = blockDim.y * blockIdx.y + threadIdx.y;
    shared_mem[y * Width + x] = in[x * Width + y];
    __syncthreads();
    out[y * Width + x] = shared_mem[y * Width + x];
}
__global__ void matrix_transpose_dynamic_shared(float* out, const float* in, const int width)
{
    extern __shared__ float shared_mem[]; // Dynamic shared memory
    const unsigned int x = blockDim.x * blockIdx.x + threadIdx.x;
    const unsigned int y = blockDim.y * blockIdx.y + threadIdx.y;
    shared_mem[y * width + x] = in[x * width + y];
    __syncthreads();
    out[y * width + x] = shared_mem[y * width + x];
}
[...]
template<unsigned int Width, unsigned int Size>
void deploy_multiple_stream(const float* h_in, float **h_transpose_matrix, const int  num_strea
ms)
{
  // Set the block dimensions
  constexpr unsigned int threads_per_block_x = 4;
  constexpr unsigned int threads_per_block_y = 4;

  // 스트림을 만들기
  std::vector<hipStream_t> streams(num_streams);
  for (int i = 0; i < num_streams; i++) {
    HIP_CHECK(hipStreamCreate(&streams[i]));
  }

  // Allocate device input and output memory and copy host input data to device memory
  std::vector<float*> d_in(num_streams);
  std::vector<float*> d_transpose_matrix(num_streams);
  const size_t size_in_bytes = sizeof(float) * Size;
  // Allocate device input memory
  HIP_CHECK(hipMalloc(&d_in[0], size_in_bytes));
  HIP_CHECK(hipMalloc(&d_in[1], size_in_bytes));
  // Allocate device output memory
  HIP_CHECK(hipMalloc(&d_transpose_matrix[0], size_in_bytes));
  HIP_CHECK(hipMalloc(&d_transpose_matrix[1], size_in_bytes));
  for (int i = 0; i < num_streams; i++) {
      HIP_CHECK(hipMemcpyAsync(d_in[i], h_in, size_in_bytes, hipMemcpyHostToDevice, streams[i
]));
  }
  static_assert(Width % threads_per_block_x == 0);
```

```
  static_assert(Width % threads_per_block_y == 0);

  // stream[0] 에서 커널을 실행
  matrix_transpose_static_shared<Width>
        <<<dim3(Width / threads_per_block_x, Width / threads_per_block_y),
           dim3(threads_per_block_x, threads_per_block_y),
           0,
           streams[0]>>>(d_transpose_matrix[0], d_in[0]);

  // stream[1] 에서 커널을 실행
  matrix_transpose_dynamic_shared
      <<<dim3(Width / threads_per_block_x, Width / threads_per_block_y),
         dim3(threads_per_block_x, threads_per_block_y),
         sizeof(float) * Width * Width,
         streams[1]>>>(d_transpose_matrix[1], d_in[1], Width);

  // Asynchronously copy the results from device to host
  for (int i = 0; i < num_streams; i++) {
        HIP_CHECK(hipMemcpyAsync(h_transpose_matrix[i],
                                 d_transpose_matrix[i],
                                 size_in_bytes,
                                 hipMemcpyDeviceToHost,
                                 streams[i]));
  }
  // Wait for all tasks in both the streams to complete on the device
  HIP_CHECK(hipDeviceSynchronize());

  // Destroy the streams
  for (int i = 0; i < num_streams; i++) {
      HIP_CHECK(hipStreamDestroy(streams[i]))
  }
  // Free device memory
  for (int i = 0; i < num_streams; i++) {
      HIP_CHECK(hipFree(d_in[i]));
      HIP_CHECK(hipFree(d_transpose_matrix[i]));
  }
}
int main() {
  // Dimension of the input square matrix is width x width
  constexpr unsigned int width = 8;
  constexpr unsigned int size  = width * width;
  constexpr unsigned int num_streams = 2; // number of streams
  const size_t size_in_bytes = sizeof(float) * size;

  float* h_in = nullptr; // Host input memory
  HIP_CHECK(hipHostMalloc(&h_in, size_in_bytes)); // host pinned memory allocation

  float *h_transpose_matrix[num_streams]; // Host output memory
  HIP_CHECK(hipHostMalloc(&h_transpose_matrix[0], size_in_bytes));
  HIP_CHECK(hipHostMalloc(&h_transpose_matrix[1], size_in_bytes));

  // Initialize the host input matrix
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
```

```
      h_in[ i * width + j ] = i;
    }
  }
  print_matrix("h_in", h_in, width, width);

  deploy_multiple_stream<width, size>(h_in, h_transpose_matrix, num_streams);
  print_matrix("transpose[0]",  h_transpose_matrix[0], width, width);
  print_matrix("transpose[1]",  h_transpose_matrix[1], width, width);

  HIP_CHECK(hipHostFree(h_in));    // Free host memory
  for (int i = 0; i < num_streams; i++) {
    HIP_CHECK(hipHostFree(h_transpose_matrix[i]));
  }
  std::cout << "streams completed!" << std::endl;
  return 0;
}
```

```
$ hipcc -w streams.hip  -o streams;  ./streams
h_in
   0    0    0    0    0    0    0    0
   1    1    1    1    1    1    1    1
   2    2    2    2    2    2    2    2
   3    3    3    3    3    3    3    3
   4    4    4    4    4    4    4    4
   5    5    5    5    5    5    5    5
   6    6    6    6    6    6    6    6
   7    7    7    7    7    7    7    7
transpose[0]
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
   0    1    2    3    4    5    6    7
transpose[1]   transpose[0] 과 동일
[...]
streams completed!
```

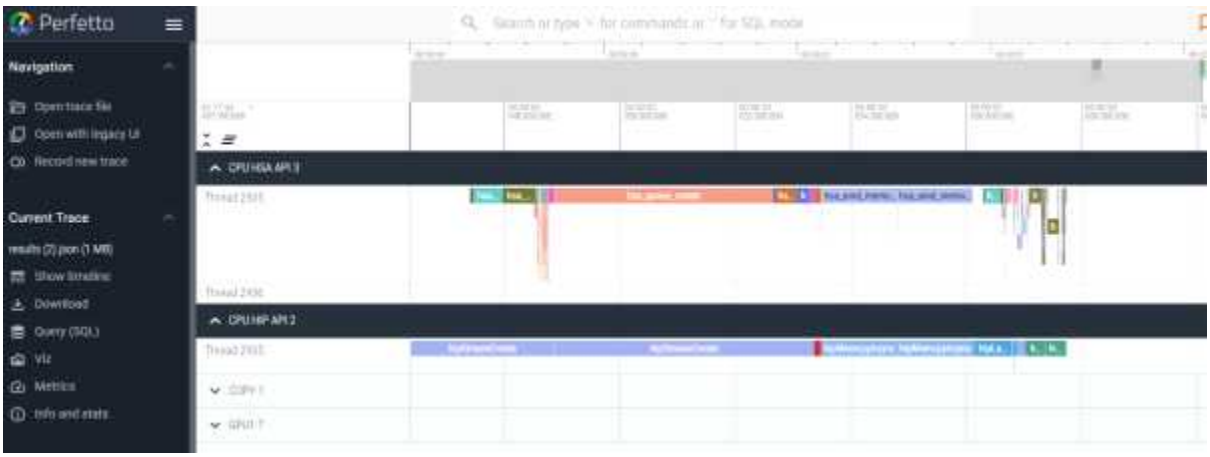
- ROCprofiler 도구를 이용하여 간단한 프로파일링이 가능함. **rocprof** 를 이용하여 프로파일링 데이터를 수집하고, perfetto UI를 통하여 관찰할 수 있음. Perfetto[25]는 구글이 개발한 오픈 소스 시스템 프로파일링, 앱 추적 및 추적 분석 도구임. 리눅스 및 안드로이드 시스템 프로파일링을 위해 개발됨. Perfetto UI[26]는 웹 기반 인터페이스로 트레이스 정보를 탐색 가능함.

```
$ rocprof -h          도움말
ROCm Profiling Library (RPL) run script, a part of ROCprofiler library package.
Full path: /opt/rocm-6.2.0/bin/rocprof
Metrics definition: /opt/rocm-6.2.0/lib/rocprofiler/metrics.xml

Usage:
  rocprof [-h] [--list-basic] [--list-derived] [-i <input .txt/.xml file>] [-o <output CSV fil
e>] <app command line>
[...]
```

25) https://perfetto.dev/
26) https://ui.perfetto.dev/

```
$ rocprof  --list-basic
$ rocprof  --list-derived
$ rocprof  --hip-trace  --hsa-trace --verbose ./streams
RPL: profiling '"./streams"'
RPL: input file ''
RPL: output dir '/tmp/rpl_data_240829_025044_2884'
------------ ------------ ------------
WARNING: rocprof(v1) is not supported on this device. Recommended use: rocprofv2
Please refer project's README for a list of supported architecures.
------------ ------------ ------------
RPL: result dir '/tmp/rpl_data_240829_025044_2884/input_results_240829_025044'
ROCtracer (2905):
    HSA-trace(*)
    HSA-activity-trace()
    HIP-trace(*)
[...]
dump json 28:29
File '/home/ubuntu/linux-drill/rocm/results.json' is generating
```

- 생성된 results.json 을 perfettoUI를 통해 열어 볼 수 있음. 단축키 a,d(좌우이동), s,w(확대,축소).



3.6 관리형 메모리

- 관리형 메모리(managed memory)는 CPU와 GPU간에 메모리를 자동으로 관리하는 기능임. 이 기능을 이용하면 데이터 이동을 수동으로 관리하지 않아도 됨. HIP 런타임이 자동으로 필요한 시점에 메모리를 관리하여 CPU와 GPU 간에 데이터를 자동으로 관리한다. `hipMallocManaged()` 함수를 통해 할당한다.

```
// managed.hip
#include <hip/hip_runtime.h>
#include <iostream>
#define N 1024
__global__ void addKernel(float* a, float* b, float* c) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {
    c[idx] = a[idx] + b[idx];
  }
}
void print_vec(float *data, int n) {
  for (int i = 0; i < n; ++i) std::cout << data[i] << " ";
  std::cout << std::endl;
```

```
}
int main() {
  float *a, *b, *c;
  hipMallocManaged(&a, N * sizeof(float));
  hipMallocManaged(&b, N * sizeof(float));
  hipMallocManaged(&c, N * sizeof(float));

  for (int i = 0; i < N; i++) {  // 데이터 초기화
    a[i] = i;       // 0,1,2,3,...,1023
    b[i] = i * 2;   // 0,2,4,6,...,2046
  }
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  hipLaunchKernelGGL(addKernel, dim3(numBlocks), dim3(blockSize), 0, 0, a, b, c);
  hipDeviceSynchronize();
  print_vec(c, N);
  hipFree(a); hipFree(b); hipFree(c);
  return 0;
}
$ hipcc -w managed.hip -o managed  ; ./managed
0 3 6 9 12 15 18 21 24 27 30 33 36 [...] 3060 3063 3066 3069
```

- hipMallocManaged() 없이 __managed__ 키워드를 사용하여 변수를 정의하여도 효과는 같음. 단, __managed__ 키워드는 ROCm 5.2.0 버전부터 지원됨.

```
// managed2.hip
[...]
#define N 1024
__managed__ float a[N], b[N], c[N];  // managed memory variables
__global__ void addKernel(float* a, float* b, float* c) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {
    c[idx] = a[idx] + b[idx];
  }
}
[...]
```

3.7 워프내 셔플

- 워프(warp) 안에 있는 스레드들을 레인(lane)이라고 함. 레인은 0 ~ warpSize-1 까지 존재. warp cross-lane 함수는 워프안의 모든 레인에 대하여 동작함. 대표적으로 __shfl 함수는 지정된 소스레인(srcLane)으로 부터 모든 스레드로 값을 브로드캐스트 함.

```
형식: T __shfl  (T var, int srcLane, int width=warpSize);
 var: 전달할 값     srcLane: 소스 레인의 lane ID
 width: 연산에 참여할 스레드 그룹의 크기. 기본값은 warpSize인 32.이 보다 클수는 없음
 리턴값은 지정된 소스레인으로 부터 전달 받은 변수의 값임.
```

```
// shfl1.hip
#include <hip/hip_runtime.h>
#include <iostream>
// #define N 16
__global__ void shflKernel(float* in, float* out, int mod) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

```
  if (idx < N) {
    float val = in[idx];
    val = __shfl(val, idx % mod);    // srcLand 값이 idx % mod 임에 주의
 // eg. idx=5, mod=2 일때 srcLand = idx % mod = 1, 즉 idx가 5인 스레드는 srcLane=1 로 부터 전달
받은 값을 리턴받음
    out[idx] = val;
  }
}
void print_vec(std::string title, float *data, int n) {
  if (title.length()) std::cout << title ;
  std::cout << " [ ";
  for (int i = 0; i < n; ++i) std::cout << data[i] << " ";
  std::cout << " ]" << std::endl;
}
void initialize_vector(float *vec, int n, float value) {
  for (int i = 0; i < n; i++) {
    vec[i] = i * value;
  }
}
__managed__ float a[N], b[N];   // managed memory
int main() {
  initialize_vector(a, N, 1);  // 0,1,2,3,4,...,N-1
  print_vec("a", a, N);

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;

  int mod = 2;
  shflKernel <<<dim3(numBlocks), dim3(blockSize), 0, 0>>>(a, b, mod);
  hipDeviceSynchronize();
  std::cout << " mod=" << mod << std::endl;
  print_vec("b", b, N);

  mod = 6;
  shflKernel <<<dim3(numBlocks), dim3(blockSize), 0, 0>>>(a, b, mod);
  hipDeviceSynchronize();
  std::cout << " mod=" << mod << std::endl;
  print_vec("b", b, N);
  return 0;
}
```

```
$ hipcc -w shfl1.hip -DN=16  -o shfl1 ; ./shfl1
a [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  ]
 mod=2
b [ 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1  ]
 mod=6
b [ 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3  ]
$ hipcc -w shfl1.hip -DN=64     -o shfl1 ; ./shfl1
a [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63  ]
 mod=2         warpSize(32) 안에서만 셔플이 이루어 짐에 주의
    0, 1 ==> 브로드케스트                                          32,33 ==> 브로드케스트
b [ 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 32 33 32 33 32 33 32 33 32
33 32 33 32 33 32 33 32 33 32 33 32 33 32 33 32 33 32 33 32 33 32 33 32 33  ]
```

```
 mod=6
b [ 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 34 35 36 37 32 33 34 35 36
37 32 33 34 35 36 37 32 33 34 35 36 37 32 33 34 35 36 37 32 33 34 35  ]
```

```
// shfl2.hip
[...]
__global__ void shflKernel(float* in, float* out, int mod) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {
    float val = in[idx];
    val = __shfl(val, idx % mod, WIDTH);   // WIDTH를 매크로에서 정의
    out[idx] = val;
  }
}
[...]
```

```
$ hipcc -w shfl2.hip  -o shfl2 -DN=64 -DWIDTH=8 ; ./shfl2
a [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63  ]
 mod=2    WIDTH=8 단위로 셔플리 이루어짐
b [ 0 1 0 1 0 1 0 1 8 9 8 9 8 9 8 9 16 17 16 17 16 17 16 17 24 25 24 25 24 25 24 25 32 33 32 33
32 33 32 33 40 41 40 41 40 41 40 41 48 49 48 49 48 49 48 49 56 57 56 57 56 57 56 57  ]
```

- 유사함수. CUDA에도 동일한 이름의 함수가 있음. _sync 함수는 향상된 버전으로 ROCm 6.1.x 버전까지는 아직 제공되고 있지 않음.

```
T __shfl_up   (T var, unsigned int delta, int width=warpSize);
T __shfl_down (T var, unsigned int delta, int width=warpSize);
T __shfl_xor  (T var, int laneMask, int width=warpSize);
T __shfl_sync      (unsigned long long mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync   (unsigned long long mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync (unsigned long long mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync  (unsigned long long mask, T var, int laneMask, int width=warpSize);
```

- shfl_up 은 워프내에서 상위 스레드로 shift 이동한다.

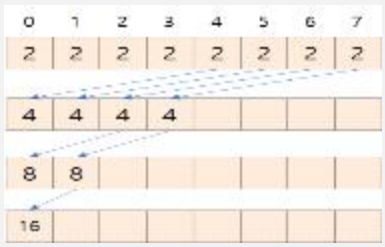
```
// shfl3.hip
[...]
__global__ void shflKernel(float* in, float* out, int mod) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {
    float val = in[idx];
    val = __shfl_up(val, 2, WIDTH);    // warp 내에서 2칸 위로 이동
    out[idx] = val;
  }
}
```

```
$ hipcc -w shfl3.hip  -o shfl3 -DN=32 -DWIDTH=16 ; ./shfl3
a [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31  ]
 mod=2    2칸씩 위로 이동됨. 워프 단위로 작동함
b [ 0 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17 16 17 18 19 20 21 22 23 24 25 26 27 28 29  ]
```

- shfl_down 을 이용하여 합계 구하기

```
// shfl4.hip
#define N 32
#define WIDTH 16
```

```
__global__ void shflKernel(float* in, float* out, int mod) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) {
    float val = in[idx];
    val += __shfl_down(val, 8, WIDTH);
    val += __shfl_down(val, 4, WIDTH);
    val += __shfl_down(val, 2, WIDTH);
    val += __shfl_down(val, 1, WIDTH);
    out[idx] = val;
  }
} [...]
```




```
$ hipcc -w shfl4.hip  -o shfl4 -DN=32 -DWIDTH=16 ; ./shfl4
a [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31  ]
 mod=2    120은 0~15 까지의 합                                    376 은 16 ~ 31까지의 합
b [ 120 128 136 144 152 160 168 176 184 192 200 208 216 224 232 240 376 384 392 400 408 416 424
432 440 448 456 464 472 480 488 496  ]
```

3.8 원자적 함수

- 원자적(atomic) 함수는 글로벌 메모리 또는 LDS 메모리에 대한 read-write 연산을 충돌없이 (conflict-free) 수행함.

형식:
```
atomicAdd(address, val)     덧셈
atomicSub(address, val)     뺄셈
atomicExch(address, val)    값을 교환
atomicMin(address, val)     최솟값
atomicMax(address, val)     최대값
atomicAnd(address, val)     비트AND 연산
atomicOr (address, val)     비트OR 연산
atomicXor(address, val)     비트XOR 연산
```

```
// atomic1.hip
#include <hip/hip_runtime.h>
#include <stdio.h>
#define N 1024
#define THREADS_PER_BLOCK 256
__global__ void sumKernel(int *input, int *sum1, int *sum2) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = gridDim.x * blockDim.x;
  for (int i = tid; i < N; i += stride) {
    sum1 += input[i];           // non atomic
    atomicAdd(sum2, input[i]);  // atomic
  }
}
int main() {
  int *d_input, *d_sum1, *d_sum2;
  int h_input[N], h_sum1 = 0, h_sum2 = 0;
  for (int i = 0; i < N; i++) h_input[i] = 1; // 0,1,2,3,...,N-1
  hipMalloc(&d_input, N * sizeof(int));
  hipMalloc(&d_sum1, sizeof(int));
  hipMalloc(&d_sum2, sizeof(int));
  hipMemcpy(d_input, h_input, N * sizeof(int), hipMemcpyHostToDevice);
  hipMemcpy(d_sum1, &h_sum1, sizeof(int), hipMemcpyHostToDevice);
```

```
  hipMemcpy(d_sum2, &h_sum2, sizeof(int), hipMemcpyHostToDevice);
  int blocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
  sumKernel<<<blocks, THREADS_PER_BLOCK>>>(d_input, d_sum1, d_sum2);

  hipMemcpy(&h_sum1, d_sum1, sizeof(int), hipMemcpyDeviceToHost);
  hipMemcpy(&h_sum2, d_sum2, sizeof(int), hipMemcpyDeviceToHost);
  printf("sum1: %d  sum2: %d\n", h_sum1, h_sum2);
  hipFree(d_input); hipFree(d_sum1); hipFree(d_sum2);
  return 0;
}
$ hipcc -w atomic1.hip   -o atomic1 ; ./atomic1
sum1: 0  sum2: 1024     atomic 함수를 써야 제대로된 결과를 얻음
```

- compare-and-swap 함수는 메모리에 저장된 값이 기대한 값과 같으면 새로운 값으로 대체시킴

형식: int atomicCAS(int *address, int expected, int new_val);
    address에 있는 값을 expected와 비교하여 일치하면 new_val로 저장하고 기존 저장된 값을 리턴.

```
// atomic2.hip
#include <hip/hip_runtime.h>
#include <stdio.h>
#define N 500
#define THREADS_PER_BLOCK 256
__global__ void incrementKernel(int* counter) {
  int old = *counter; // 기존의 값을 읽음
  int assumed;
  do {
    assumed = old;     // 기존의 값과 같으면 1을 증가한 값을 기록
    old = atomicCAS(counter, assumed, assumed + 1);
  } while (assumed != old); // 만일 다른 스레드가 값을 변경했으면, 이 과정을 성공할때까지 반복
}
int main() {
  int *d_counter;
  int h_counter = 0;
  hipMalloc(&d_counter, sizeof(int));
  hipMemcpy(d_counter, &h_counter, sizeof(int), hipMemcpyHostToDevice);
  int blocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
  incrementKernel<<<blocks, THREADS_PER_BLOCK>>>(d_counter);
  hipMemcpy(&h_counter, d_counter, sizeof(int), hipMemcpyDeviceToHost);
  printf("Final counter value: %d\n", h_counter);
  hipFree(d_counter);
  return 0;
}
$ hipcc -w atomic2.hip   -o atomic2 ; ./atomic2
Final counter value: 512       N은 500이지만, 2개의 블럭에서 총 512개 스레드가 실행되었음
```

- Compare-and-Swap 연산을 수행하여 락(lock)을 구현할 수 있음.

```
__device__ void lock(int* mutex) {
    while (atomicCAS(mutex, 0, 1) != 0);
}
__device__ void unlock(int* mutex) {
    atomicExch(mutex, 0);
}
```

3.9 워프 단위 리덕션

- 워프내부의 레인은 0 ~ warpSize-1 까지 존재. 워프내에서 평가된 값이 브로드캐스트되어 전체 워프를 대상으로 리덕션(reduction)됨. __all()은 워프레인 전체 대하여 근거(predicate)가 0이 아니면 1을 리턴. __any()는 워프레인 전체 중 어느 하나가 predicate가 0이 아니면 1을 리턴.

형식: int __all(int predicate)
int __any(int predicate)

```
// allany.hip
#include <hip/hip_runtime.h>
#include <iostream>

__global__ void checkAllPositive(int* data, int* result, int size) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int warpId = idx / 32;
  bool isPositive = (idx < size) ? (data[idx] > 0) : true;
  bool allPositive = __all(isPositive); // 워프 내의 모든 스레드가 양수를 가지고 있는지 확인
  if (threadIdx.x % 32 == 0) {  // 워프의 첫 번째 스레드만 결과를 기록
    result[warpId] = allPositive;
  }
}
__global__ void checkAnyPositive(int* data, int* result, int size) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int warpId = idx / 32;
  bool isPositive = (idx < size) ? (data[idx] > 0) : false;
  bool anyPositive = __any(isPositive); // 워프 내의 어느 스레드가 양수를 가지고 있는지 확인
  if (threadIdx.x % 32 == 0) {  // 워프의 첫 번째 스레드만 결과를 기록
    result[warpId] = anyPositive;
  }
}
void print_vec(std::string title, int *data, int n) {
  if (title.length()) std::cout << title ;
  std::cout << " [ ";
  for (int i = 0; i < n; ++i) std::cout << data[i] << " ";
  std::cout << " ]" << std::endl;
}
int main() {
  constexpr int size = 128;
  int h_data[size];
  int h_result[size/32];
  int *d_data, *d_result;
  for (int i = 0; i < size; i++) {
    if (i >= 40 & i <= 123) h_data[i] = -1; // 인덱스 40~123 이면 음수
    else h_data[i] = i + 1; // 양수
  }
  print_vec("h_data", h_data, size);

  hipMalloc(&d_data, size * sizeof(int));
  hipMalloc(&d_result, (size / 32) * sizeof(int));
  hipMemcpy(d_data, h_data, size * sizeof(int), hipMemcpyHostToDevice);

  constexpr int threadsPerBlock = 256;
  constexpr int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

  checkAllPositive<<<blocksPerGrid, threadsPerBlock>>>(d_data, d_result, size);
```

```
  hipMemcpy(h_result, d_result, (size / 32) * sizeof(int), hipMemcpyDeviceToHost);
  print_vec("checkAllPositive", h_result,  size/32);

  checkAnyPositive<<<blocksPerGrid, threadsPerBlock>>>(d_data, d_result, size);
  hipMemcpy(h_result, d_result, (size / 32) * sizeof(int), hipMemcpyDeviceToHost);
  print_vec("checkAnyPositive", h_result,  size/32);

  hipFree(d_data);  hipFree(d_result);  return 0;
}
$ hipcc -w allany.hip   -o allany ; ./allany
h_data [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 125 126 127
128  ]  // 인덱스 40~123 이면 음수   [1~32][33~64][66~96][97~128] 4개로 나뉨
checkAllPositive [ 1 0 0 0  ]   128개를 32개씩 묶음으로 첫번째 묶음만 all positive임
checkAnyPositive [ 1 1 0 1  ]                          세번째 묶음만 any positive가 0임
```

- __ballot은 조건을 만족하는가에 대한 비트마스크(bit mark)정보를 제공한다.

  형식: uint64_t __ballot(int predicate)

```
// ballot.hip
#include <hip/hip_runtime.h>
#include <iostream>
__global__ void ballotExample(int* input, int* output, int size) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int warpId = idx / 32;
  int laneId = idx % 32;
  int predicate = (idx < size) ? (input[idx] > 0) : 0; // 조건 확인: 입력 값이 0보다 큰지
  unsigned int ballot = __ballot(predicate);
  if (laneId == 0) { // 워프의 첫 번째 스레드만 결과를 기록
    output[blockIdx.x * (blockDim.x / 32) + warpId] = ballot;
  }
}
[...]
int main() {
  constexpr int size = 128;
  int h_data[size];
  int h_result[size/32];
  int *d_data, *d_result;

  for (int i = 0; i < size; i++) {
    if (i >= 40 & i <= 123) h_data[i] = -1; // 인덱스 40~123 이면 음수
    else h_data[i] = i + 1; // 데이터 초기화 모두 양수
  }
  print_vec("h_data", h_data, size, 0);

  hipMalloc(&d_data, size * sizeof(int));
  hipMalloc(&d_result, (size / 32) * sizeof(int));
  hipMemcpy(d_data, h_data, size * sizeof(int), hipMemcpyHostToDevice);
  constexpr int threadsPerBlock = 256;
  constexpr int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;
  ballotExample<<<blocksPerGrid, threadsPerBlock>>>(d_data, d_result, size);
  hipMemcpy(h_result, d_result, (size / 32) * sizeof(int), hipMemcpyDeviceToHost);
  print_vec("ballotExample", h_result,  size/32, 1);
```

```
  hipFree(d_data);  hipFree(d_result);  return 0;
}
$ hipcc -w ballot.hip   -o ballot ; ./ballot
h_data [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 125 126 127
128  ]  // 인덱스 40~123 이면 음수. 음수에 해당하는 비트 값이 0임.
          2번째 블럭은 하위 8비트가 1임.  4번째 블럭은 상위 4비트가 1임
ballotExample [ 0xffffffff 0x000000ff 0x00000000 0xf0000000  ]
```

3.10 로깅 및 추적

- 애플리케이션 런타임에 HIP API에 관한 로깅 기능을 제공함. 로깅 레벨과 로깅 마스크를 이용하여 로깅할 정보를 선택할 수 있음. AMD_LOG_LEVEL 환경변수를 설정하여 로깅 레벨을 지정, AMD_LOG_MASK 환경변수를 통하여 로깅할 기능을 선택가능.

```
enum LogLevel {
  LOG_NONE    = 0,
  LOG_ERROR   = 1,
  LOG_WARNING = 2,
  LOG_INFO    = 3,
  LOG_DEBUG   = 4
};
```

```
enum LogMask {
  LOG_API       = 0x00000001, //!< API call
  LOG_CMD       = 0x00000002, //!< Kernel and Copy Commands and Barriers
  LOG_WAIT      = 0x00000004, //!< Synchronization and waiting for command
s to finish
  LOG_AQL       = 0x00000008, //!< Decode and display AQL packets
  LOG_QUEUE     = 0x00000010, //!< Queue commands and queue contents
  LOG_SIG       = 0x00000020, //!< Signal creation, allocation, pool
  LOG_LOCK      = 0x00000040, //!< Locks and thread-safety code.
  LOG_KERN      = 0x00000080, //!< kernel creations and arguments, etc.
  LOG_COPY      = 0x00000100, //!< Copy debug
  LOG_COPY2     = 0x00000200, //!< Detailed copy debug
  LOG_RESOURCE  = 0x00000400, //!< Resource allocation, performance-impact
ing events.
  LOG_INIT      = 0x00000800, //!< Initialization and shutdown
  LOG_MISC      = 0x00001000, //!< misc debug, not yet classified
  LOG_AQL2      = 0x00002000, //!< Show raw bytes of AQL packet
  LOG_CODE      = 0x00004000, //!< Show code creation debug
  LOG_CMD2      = 0x00008000, //!< More detailed command info, including b
arrier commands
  LOG_LOCATION  = 0x00010000, //!< Log message location
  LOG_ALWAYS    = 0xFFFFFFFF, //!< Log always even mask flag is zero
};
```

```
$ hipcc   -o bandwidth  bandwidth.hip  -DDSIZE=1048576   # 1MB
$ AMD_LOG_LEVEL=3 ./bandwidth
:3:rocdevice.cpp           :468 : 14275948620 us: [pid:2285  tid:0x733637a8e180] Initializing HSA stack.
:3:rocdevice.cpp           :554 : 14275956317 us: [pid:2285  tid:0x733637a8e180] Enumerated GPU agents =
1
:3:rocdevice.cpp           :232 : 14275956358 us: [pid:2285  tid:0x733637a8e180] Numa selects cpu agent
[0]=0x1ed8210(fine=0x1ed8400,coarse=0x1ed88f0) for gpu agent=0x1ed8e30 CPU<->GPUXGMI=0
:3:comgrctx.cpp            :33  : 14275956378 us: [pid:2285  tid:0x733637a8e180] Loading COMGR library.
:3:comgrctx.cpp            :126 : 14275956409 us: [pid:2285  tid:0x733637a8e180] Loaded COMGR library ve
rsion 2.8.
:3:rocdevice.cpp           :1809: 14275956897 us: [pid:2285  tid:0x733637a8e180] Gfx Major/Minor/Steppin
g: 10/1/1
[...]
```

```
:3:rocdevice.cpp           :285 : 14276249578 us: [pid:2285  tid:0x733637a8e180] Deleting hardware queue
0x73352c300000 with refCount 0
  각 컬럼의 의미는 다음과 같다.
  Log Level : File and Line Number :  Timestamp : ProcessID, ThreadID : Function and Message
  메시지 부분만 잘라서 출력하기
$ (AMD_LOG_LEVEL=4 ./bandwidth &> a) ; (cat a ¦ awk '{ print substr($0,83) }')
[...]
Direct Dispatch: 1
 hipMalloc ( 0x7ffc4df8a678, 1048576 ) <----- 메모리 할당 1MB
Device=0x79be80, freeMem_ = 0x1f6f00000
hipMalloc: Returned hipSuccess : 0x78b54e200000: duration: 98 us <--- 메모리 할당 98us
 hipMemcpy ( 0x78b54e200000, 0x78b6598ff010, 1048576, hipMemcpyHostToDevice )
Number of allocated hardware queues with low priority: 0, with normal priority: 0, with high pr
iority: 0, maximum per priority is: 4
Created SWq=0x78b659d0d000 to map on HWq=0x78b54d100000 with size 16384 with priority 1, cooper
ative: 0
acquireQueue refCount: 0x78b54d100000 (1)
Using Code Object V5.
Host active wait for Signal = (0x78b6507ff780) for -1 ns
hipMemcpy: Returned hipSuccess : : duration: 285434 us <------ H2D 복사
 hipDeviceSynchronize (  )
hipDeviceSynchronize: Returned hipSuccess :
 hipMemcpy ( 0x7f86ecaff010, 0x7f85e1000000, 1048576, hipMemcpyDeviceToHost )
Host active wait for Signal = (0x7f86e35ff700) for -1 ns
hipMemcpy: Returned hipSuccess : : duration: 1474 us <---- D2H 복사
 hipDeviceSynchronize (  )
[...]
```

- ltrace를 이용한 추적

```
ltrace -C -e "hip*" ./launchkernel
```

3.11 난수

- 커널 내부에서 난수(random number)를 사용하기 위해서는 rocRAND[27] 기능을 활용함. 여러가지 유사난수(PRNG, pseudo-random number generator) 기능을 제공하고 있음. 본 예제에서는 XORWow PRNG 기능을 사용함

```
// rng.hip
#include <hip/hip_runtime.h>
#include <rocrand/rocrand_kernel.h>
#define N  10      스레드 개수
#define MAX 10     난수발생 0 ~ MAX 사이의 정수
[...]
struct MyStruct {   // 스레드 내부의 인덱스 값을 저장
[...]
  int number;
  rocrand_state_xorwow  rng; // 난수 발생기 상태 저장
};
__managed__  unsigned long long   seed;  // 난수 발생기 seed
__global__ void myKernel(MyStruct *kinfo, int n) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx >= n) return;
```

27) https://rocm.docs.amd.com/projects/rocRAND/en/docs-6.1.2/index.html

```
  kinfo[idx].idx = idx;  // 현재 스레드의 정보를 저장
  kinfo[idx].thread_id   = threadIdx.x;
  kinfo[idx].block_id    = blockIdx.x;
  kinfo[idx].blockdim_id = blockDim.x;

  rocrand_init(seed, idx, 0, &(kinfo[idx].rng)); // XORWOW 상태 초기화
  int rand = floor(rocrand_uniform (&(kinfo[idx].rng)) * (MAX+1));
  kinfo[idx].number = rand;
}
int main() {
  MyStruct *d_krnInfo, *h_krnInfo;
  h_krnInfo = new MyStruct[N];
  int krninfo_size = N * sizeof(struct MyStruct);
  hipMalloc((void**)&d_krnInfo, krninfo_size); // 디바이스 메모리 할당(kernel info)
  // 현재 시간을 seed로 사용
  seed = std::chrono::high_resolution_clock::now().time_since_epoch().count();
  // Kernel 실행
  constexpr unsigned int block_size = 256; // 블록당 스레드 개수
  unsigned int grid_size = (N + block_size - 1) / block_size; // 블럭 개수
  myKernel<<<grid_size, block_size>>>(d_krnInfo, N);
  hipDeviceSynchronize();
  hipMemcpy(h_krnInfo, d_krnInfo, krninfo_size, hipMemcpyDeviceToHost); // 호스트로 복사
  std::cout << "idx  .number " << std::endl;
  for (int i = 0; i < N; i++) {
    std::cout << ""
       << "" << h_krnInfo[i].idx
       << "  " << h_krnInfo[i].number
       << std::endl;
  }
  hipFree(d_krnInfo);   return 0;
}
```

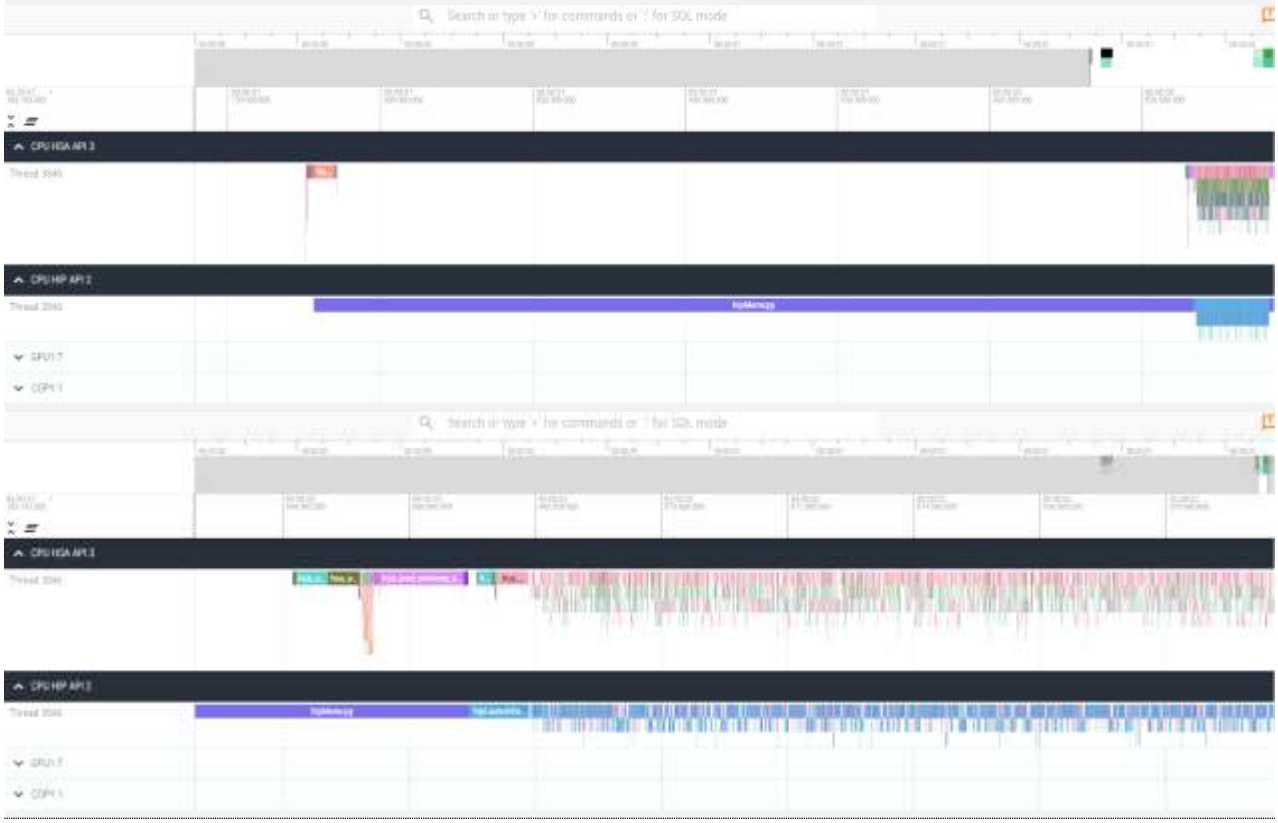
```
$ hipcc -w rng.hip -o rng ; ./rng
idx  .number    여러번 실행할 때마다 결과가 다르게 나와야 함
0  0
1  6
2  8
3  9
4  0
5  6
6  5
7  0
8  6
9  5
```

4. HIP 응용 코드 예제

4.1 버블 정렬

```
// sort1.hip
[...]
const int ARRAY_SIZE = 1024;  // 정렬할 배열의 크기
__global__ void bubbleSortKernel(int* data, int n, int step) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;
  for (int i = idx; i < n - 1; i += stride) {   // bubble sort
    if (i % 2 == step) {
      if (data[i] > data[i + 1]) {
        int temp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = temp;
      }
    }
  }
}
[...]
bool is_sorted(int *arr, int size) {
  for (int i = 0; i < size - 1; i++) {
    if (arr[i] > arr[i + 1]) {      return false;    }
  }
  return true;
}
void hipSort(int* data, int n) {
  int* d_data;
  HIP_CHECK(hipMalloc(&d_data, n * sizeof(int)));
  HIP_CHECK(hipMemcpy(d_data, data, n * sizeof(int), hipMemcpyHostToDevice));
  constexpr int BLOCK_SIZE = 256;
  dim3 block(BLOCK_SIZE);
  dim3 grid((n + block.x - 1) / block.x);
  for (int i = 0; i < n; i++) {
    bubbleSortKernel<<<grid, block>>>(d_data, n, i % 2);  // kernel executions
    HIP_CHECK(hipGetLastError());
  }
  HIP_CHECK(hipMemcpy(data, d_data, n * sizeof(int), hipMemcpyDeviceToHost));
  HIP_CHECK(hipFree(d_data));
}
int main() {
  constexpr unsigned int n = ARRAY_SIZE;
  int data[n];
  initialize_vector(data, n); // random number initialize
  print_vec("before ", data, n);
  hipSort(data, ARRAY_SIZE); // HIP을 사용한 정렬 수행
  print_vec("after ", data, n);
  if (is_sorted(data, n)) std::cout << "Sorting success" << std::endl;
  else std::cout << "Sorting FAIL!!" << std::endl;
  return 0;
}
$ hipcc -w sort1.hip  -o sort1 ; ./sort1
before  [ 6418 3005 6475 4004 7732 5509 [...]  8690 1728  ]
```

```
after  [ 8 15 33 43 60 65 74 75 82 123 [...] 9981 9990  ]
Sorting success
$ rocprof  --hip-trace  --hsa-trace --verbose ./sort1
 results.json을 perfettoUI 에서 읽기 (n 번의 커널 호출이 발생함)
```



4.2 문자열 패턴 매칭

- 문자열 패턴 매칭은 주어진 문자열에서 특정 패턴을 찾는 알고리즘임. 단순 비교 알고리즘 부터, KMP(Knuth-Morris-Pratt, 커누스-모리스-프랫) 알고리즘[28], 라빈-카프(Rabin-Karp), 보이어-무어(Boyer-Moore)등 여러 알고리즘이 존재함. 다음 코드는 단순 비교 알고리즘을 사용하여 GPU에서 패턴을 검색한다. 일정한 크기의 chunksize로 나누어 각 스레드에서 처리함.

```
// pattern3.hip
[...]
#define PATTERN_SIZE 3     // 패턴 길이
#define CHUNK_SIZE  64     // chunk 단위로 잘라서 스레드에서 처리
#define DATA_SIZE    1024  // 전체 데이터 크기
struct MyStruct {
  int idx;
  int thread_id;
  int block_id;
  int blockdim_id;
  int n_count;  // 각 스레드에서 패턴을 찾은 카운터
};
char pattern[PATTERN_SIZE];   // 패턴 데이터
// pattern at device (managed variable)
__managed__ char    d_pattern[PATTERN_SIZE];
```

28) https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

```
__global__ void patternKernel(
   MyStruct *kinfo, char *d_data, int ds, int chunksize, int n_chunk, char *pattern, int ps)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < n_chunk) {
    kinfo[idx].idx = idx;
    kinfo[idx].thread_id   = threadIdx.x;
    kinfo[idx].block_id    = blockIdx.x;
    kinfo[idx].blockdim_id = blockDim.x;
    kinfo[idx].n_count = 0;
    unsigned int start = idx * chunksize;
    // find pattern in data
    int count = 0;
    for (int i = start; i < start+chunksize; i++) { // start 부터 chunksize 만큼
      bool flag = true;
      for (int j = 0; j < ps; j++) { // 단순비교 문자열 매칭 알고리즘
        if (d_data[i+j] != pattern[j]) {
          flag = false;
          break;
        }
      }
      if (flag) {        count++;      }
    }
    kinfo[idx].n_count = count;
  }
}
[...]
void print_krn_info(MyStruct *h_krnInfo, int n) {
  std::cout << "idx threadID  blockID  blockdimID count" << std::endl;
  int sum = 0;
  for (int i = 0; i < n; i++) {
    std::cout << ""
       << "" << h_krnInfo[i].idx
       << "\t" << h_krnInfo[i].thread_id
       << "\t" << h_krnInfo[i].block_id
       << "\t" << h_krnInfo[i].blockdim_id
       << "\t" << h_krnInfo[i].n_count
       << std::endl;
    sum += h_krnInfo[i].n_count; // 각 스레드에서 찾은 횟수를 합산
  }
  std::cout << "sum : " << sum << std::endl; // 총 매칭된 횟수
}
int main(void) {
  constexpr unsigned int ds = DATA_SIZE;
  constexpr unsigned int ps = PATTERN_SIZE;
  constexpr unsigned int chunksize = CHUNK_SIZE ; // data를 chunk 단위로 각 스레드에 할당
  constexpr unsigned int n_chunk = ds/chunksize; // chunk 개수
  MyStruct *d_krnInfo;
  MyStruct *h_krnInfo;

  char *h_data = new char[ds];
  initialize_vector_random(h_data, ds);
```

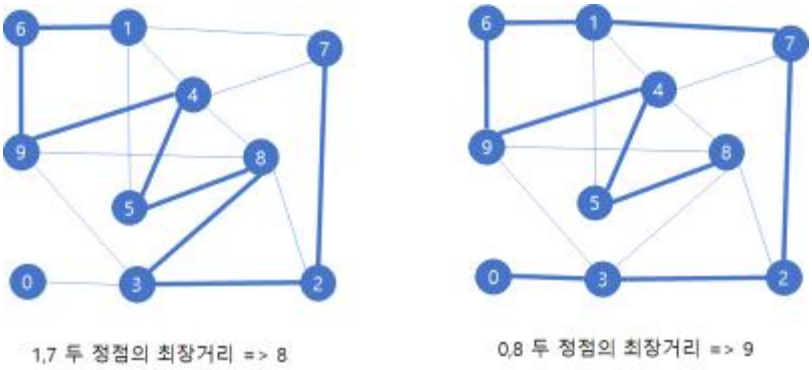
```
  initialize_vector_random(pattern, ps);

  print_vec2("data", h_data, ds, chunksize); // 데이터를 출력
  std::cout << "  data length : " << ds << std::endl;
  print_vec("pattern", pattern, ps);
  std::cout << "  pattern length : " << ps << std::endl;
  copy_vec(pattern, d_pattern, ps); // 패턴을 디바이스 메모리로 복사

  char *d_data;
  HIP_CHECK(hipMalloc(&d_data, ds * sizeof(char))); // 디바이스 메모리 할당

  h_krnInfo = new MyStruct[n_chunk];      // 호스트 메모리 할당 (kernel info)
  hipMalloc((void**)&d_krnInfo, n_chunk*sizeof(MyStruct)); // 디바이스 메모리 할당(kernel info)

  // 호스트에서 디바이스로 데이터 복사
  HIP_CHECK(hipMemcpy(d_data, h_data, ds * sizeof(char), hipMemcpyHostToDevice));

  constexpr unsigned int blocksize = 256;
  dim3 blockSize(blocksize);
  dim3 gridSize((n_chunk + blockSize.x - 1) / blockSize.x);
  std::cout
        << "  chunk size: " << chunksize
        << "  chunks: " << n_chunk
        << "  blockSize.x: " << blockSize.x
        << "  gridSize.x : " << gridSize.x
        << std::endl;

  // kernel 호출
  patternKernel<<<gridSize, blockSize, 0, 0>>>
    (d_krnInfo, d_data, ds, chunksize, n_chunk, d_pattern, ps);
  hipMemcpy(h_krnInfo, d_krnInfo, n_chunk * sizeof(MyStruct), hipMemcpyDeviceToHost);
  print_krn_info(h_krnInfo, n_chunk);
  hipFree(d_data);
  hipFree(d_krnInfo);
  delete[] h_data;
  return 0;
}
```

```
$ hipcc -w pattern3.hip -o pattern3 ; ./pattern3
data
    0 CDBCBEDCEBEDEDEDDEEDDBCEDBBBCDEEDCCDEBEDBCDECDCBEBBCDBCDBECDCBDE <-- chunk 0
   64 BBBBBBEDDEECDCDDBCCECBCBDEBCBBCBCDEECCECBCCCDDBBBBBDBEDBEBEEEECC <-- chunk 1
  128 ECDEBECDDBDBCDBCDEEECBBCECEDBBCBEDDBDBDCCDCDDDDBCBECBDECBECCDEDD ...
  192 CDEDBCCCBDBCEDDCEEDEDBDEEDCEBCEBCEEBEBBDBDBDBBDBBDDDDBECDDDCDCB
  256 EDBCDECDCDCCECCDEDCEDDDBDCDCEEBBEDBECDEDCEDCBBDCBEDDBBCECBBECCDC
  320 DECBCCBCCECEBCBECEDDECECECDEDDDBBCBCDECCBDEDBDBCDBBDBCBEBBDDEEDE
  384 ECCDBEBCEEEDCEBBEBDBEDBDBCDBBEBEDEEEDDEDCDBDDCBEBDEDBBDBCDBCDCCC
  448 CEEBBBEDDBEBBDEDECCDBDBDEDDDDBBDCECBBEBCEEECBDDCCCEBDCDDBBDEEEBC
  512 EEEEBBDCBCEBDCCDEBEEDBEDECBDDBDDBBEDECCBCEBBDBBDBDEBECCDEBDEBBCC
  576 EEBDDCBCDBECCECDECBEEBDDECEDBCEDEEDDDCBBDDBDBCBCDDDDEBCCCCBECBED
  640 CDDCEBDEBDDBBECCEEEECECBBBEBDBDEEEBECDDEDDBBBDDBDBDCBBDDDBBBCECB
  704 EDEBCCDBEBBCBEBCBBEBBDBBDECCBBDCCDBDBBCDEEDECCDDBBEBBEEDBBBBEBDC
  768 EBCEDECCBCDCCDEEEEEBDDCDDBDDEECBDDCEDCECCDBEDEBBEEDCECBBBBDBEDEC
  832 BEDDDCCDCEDEDDCCBECDDDDDECCBECBDEBDDECECEEEEBECEDCBCDECCDBDECBDD
  896 CCDDEECECEBCDCEEDBBEBDCCCEEDEBCBEEDDCDECCEEBCEDEEDBBCDDDBCBCEDED
  960 CCDDDCBCCBCECCDECBEDBBBEDCCCCEBBDBDBEBBBCBBBBBBBDEECEEDCDBCBBEDE <-- n_chunk 16
```

```
  data length : 1024
pattern [ CEC ]  <------ 찾을 패턴
  pattern length : 3
  chunk size: 64  chunks: 16  blockSize.x: 256  gridSize.x : 1
idx threadID  blockID  blockdimID count   count : chunk에서 찾은 패턴의 개수
0       0       0       256     0
1       1       0       256     3
2       2       0       256     1
3       3       0       256     0
4       4       0       256     2
5       5       0       256     3
6       6       0       256     0
7       7       0       256     1
8       8       0       256     0
9       9       0       256     1
10      10      0       256     2
11      11      0       256     0
12      12      0       256     2
13      13      0       256     1
14      14      0       256     1
15      15      0       256     1
sum : 18
```

4.3 그래프 최장경로 문제

- 그래프 이론에서 최장 경로 문제[29]는 주어진 그래프에서 최대 길이의 단순 경로(반복되는 정점이 없는 경로)를 찾는 문제임.



- NetworkX[30]를 이용하여 랜덤 그래프를 생성하는 파이썬 프로그램

```
$ sudo apt install python3-networkx  python3-matplotlib
# graph_gen.py
import networkx as nx
import matplotlib.pyplot as plt
n_nodes = 10
probility = 0.2
random_graph = nx.gnp_random_graph(n=n_nodes, p=probility) # 랜덤 그래프
edges = random_graph.edges()  # edges 리스트
edges = list(edges)  # 파이썬 리스트로 변환
print(edges)
```

29) https://en.wikipedia.org/wiki/Longest_path_problem
30) https://networkx.org/

```
n_edges = len(edges)
print (n_nodes, n_edges) # 정점의 개수, 간선의 개수
for edge in edges:
  print (edge[0], edge[1])
```

```
$ python3  graph_gen.py
[(0, 3), (3, 2), (3, 8), (3, 9), (1, 4), (1, 5), (1, 6), (1, 7), (4, 5), (4, 7), (4, 8), (4,
9), (5, 8), (6, 9), (7, 2), (2, 8), (8, 9)]
10 17      10: 정점의 개수  17: 간선의 개수
0 3
3 2
3 8
3 9
1 4
1 5
[...]
```

```
// graph1.cpp      최장경로 길이를 구하는 프로그램
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Function to perform DFS and find the longest path
void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited, int currentLength, int& max
Length) {
  visited[node] = true;
  maxLength = max(maxLength, currentLength);
  for (int neighbor : adj[node]) {
    if (!visited[neighbor]) {
      dfs(neighbor, adj, visited, currentLength + 1, maxLength);
    }
  }
  visited[node] = false;
}
int main() {
  int n, m; // n: number of vertices, m: number of edges
  cin >> n >> m;  // 표준 입력으로 입력 받음
  vector<vector<int>> adj(n);  // node 간의 인접 리스트(adjacent list)
  for (int i = 0; i < m; ++i) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u); // undirected graph 이므로 양쪽 방향으로 인접 정보를 추가
  }
  int maxLength = 0;
  vector<bool> visited(n, false); // 노드를 방문했는지 여부
  for (int i = 0; i < n; ++i) {
    dfs(i, adj, visited, 0, maxLength);  // 재귀호출 함수
  }
  cout << "The longest path length is: " << maxLength << endl;
  return 0;
}
```

```
$ g++  graph1.cpp -o graph1 ; ./graph1  < graph1.txt
The longest path length is: 9
```

- 주어진 그래프 정보는 표준 입력으로 받음. 인접행렬(adj)을 구성하고, 그래프 정점의 개수만큼 커널 스

레드를 실행. 다음에 방문할 노드 정보를 랜덤 넘버를 생성하여 결정함. 각 스레드 별로 처음 방문한 노드 번호와 마지막 방문한 노드의 번호, 경로의 길이를 출력한다.

```
// graph2.hip
[...]
#define MAX_NODES 1024  // 최대 처리가능한 노드의 수
[...]
__device__ int get_next_random_adj_node( MyStruct *kinfo, int idx, int curr_node, int *adj, int
n, int *visited ) {   // 디바이스 함수  (다음 방문할 노드를 결정)
  int stack_data[MAX_NODES]; // stack
  int stack_top = 0;
  for (int i = 0; i < n; i++) {
    int index = curr_node * n + i; // i: next node
    if (!adj[index]) continue; // 인접하지 않음
    if (!visited[i]) { // 아직 방문하지 않음
      stack_data[stack_top++] = i; // 다음에 방문할 인접 노드들을 stack에 추가
    }
  }
  if (stack_top == 0) return -1; // 더이상 방문할 노드가 없음
  // stack에서 랜덤하게 하나를 골라서 방문
  int rand = floor(rocrand_uniform (&(kinfo[idx].rng)) * stack_top);
  return stack_data[rand];
}
// adj : 그래프 인접 노드 정보
// n : 노드 개수
__global__ void myKernel(MyStruct *kinfo, int *adj, int n) {  // 커널 함수
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx >= n) return;
  kinfo[idx].idx = idx;  // 현재 스레드의 정보를 저장
[...]
  kinfo[idx].first_visit = -1;
  kinfo[idx].last_visit = -1;
  kinfo[idx].length = 0;
  rocrand_init(seed, idx, 0, &(kinfo[idx].rng)); // XORWOW 상태 초기화
  int from_node = idx; // idx 값에 해당하는 노드에서 시작
  int currentLength = 0;
  int visited[MAX_NODES]; // 방문 여부를 표시하는 로컬 변수
  for (int i = 0; i < n; i++) visited[i] = 0; // 초기화

  int curr_node = from_node;
  visited[curr_node] = true;
  while (1) {
    int next = get_next_random_adj_node( kinfo, idx, curr_node, adj, n, visited );
    if (next == -1) break; // 더 방문할 노드가 없으면 종료
    if (kinfo[idx].first_visit == -1) kinfo[idx].first_visit = next;
    visited[next] = true;
    kinfo[idx].last_visit = next;
    currentLength++;   // 다음 노드로 이동하고 길이를 증가 시킴
    curr_node = next;
  }
  kinfo[idx].length = currentLength;
}
[...]
int main() {
  int n, m; // n: number of vertices, m: number of edges
```

```
  std::cin >> n >> m;
  int *adj;
  adj = (int *)malloc(n * n *sizeof(int)); // 호스트 메모리. node간의 연결을 표시. 2D matrix
  for (int i = 0; i < n*n; i++) adj[i] = 0; // 초기화
  for (int i = 0; i < m; ++i) { // edge 정보를 입력받음
    int u, v;
    std::cin >> u >> v;
    int a = u * n + v;  // v row, u column
    int b = v * n + u;  // u row, v column
    adj[a] = 1;
    adj[b] = 1;
  }
  print_matrix("adj matrix", adj, n, n);
[...]
  // Kernel 실행
  constexpr unsigned int block_size = 256; // 블록당 스레드 개수
  unsigned int grid_size = (n + block_size - 1) / block_size; // 블럭 개수
  myKernel<<<grid_size, block_size>>>(d_krnInfo, d_adj, n);
  hipDeviceSynchronize();
[...]
  std::cout << "idx  .first_visit .last_visit  .length" << std::endl;
  for (int i = 0; i < n; i++) {
    std::cout << ""
       << "" << h_krnInfo[i].idx
       << "  " << h_krnInfo[i].first_visit
       << "  " << h_krnInfo[i].last_visit
       << "  " << h_krnInfo[i].length
       << std::endl;
  }
  hipFree(d_krnInfo); hipFree(d_adj);  free(adj);  return 0;
}
```

```
$ hipcc -w graph2.hip -o graph2
$ ./graph2 < graph1.txt
adj matrix        그래프 인접 행렬
   0    0    0    1    0    0    0    0    0    0
   0    0    0    0    1    1    1    1    0    0
   0    0    0    1    0    0    0    1    1    0
   1    0    1    0    0    0    0    0    1    1
   0    1    0    0    0    1    0    1    1    1
   0    1    0    0    1    0    0    0    1    0
   0    1    0    0    0    0    0    0    0    1
   0    1    1    0    1    0    0    0    0    0
   0    0    1    1    1    1    0    0    0    1
   0    0    0    1    1    0    1    0    1    0
idx  .first_visit .last_visit  .length
0  3  5  9       0번 노드 방문 경로: 정점 3에서 시작 정점 5에서 종료. 길이는 9
1  7  6  4
2  7  0  9
3  0  0  1       3번 노드: 정점 0까지 길이 1
4  9  0  7
5  4  9  8
6  1  9  6
7  2  4  8
8  9  4  7
9  6  0  8
```

[참고자료]

- ROCm Documentation https://rocblas.readthedocs.io/en/rocm-6.0.0/what-is-rocm.html
- AMD RDNA3 ISA https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf
- Intro to HIP Programming, AMD, Youtube https://www.youtube.com/watch?v=hSwgh-BXx3E&list=PLx15eYqzJifehAxhWRD6T35GZwAqM9IK4
- GPU programming with HIP, Dec. 9-10, 2021, CSC-IT Center for Science Ltd. Expoo https://events.prace-ri.eu/event/1280/
- AMD HIP Tutorial, Yifan Sun, William & Mary, https://www.youtube.com/watch?v=9ZGpM3zXKoY&list=PLB1fSi1mbw6IKbZSPz9a2r2DbnHWnLbF-&index=1
- AMD ROCm documentation https://rocm.docs.amd.com/en/latest/
- AMD HIP Programming Guide https://raw.githubusercontent.com/RadeonOpenCompute/ROCm/rocm-4.5.2/AMD_HIP_Programming_Guide.pdf
- ROCm Examples https://github.com/ROCm/rocm-examples
- hip-tests, github https://github.com/ROCm/hip-tests
- hipBLAS API 2.2.0 https://rocm.docs.amd.com/projects/hipBLAS/en/latest/functions.html
- rocBLAS https://rocm.docs.amd.com/projects/rocBLAS/en/latest/how-to/Programmers_Guide.html
- ROCm/Tensile https://github.com/ROCm/Tensile
- Advanced HIP Workshop, Pawsey Supercomputing Research Centre https://www.youtube.com/playlist?list=PLmu61dgAX-aaqtQGzm5yEPg3Fm1pHSh3J
- Optimizing for the Radeon RDNA Architecture https://www.youtube.com/watch?v=7eEKLUhoTQs
- Perfetto https://ui.perfetto.dev/
- ROCProfiler documentation https://rocm.docs.amd.com/projects/rocprofiler/en/latest/index.html
- Developing HPC applications with AMD GPU https://www.hpc.rs/events/developing-hpc-applications-with-amd-gpus
- General-Purpose Graphics Processor Architecture, Tor M. Aamodt, Morgan&Claypool, 2018
- Using with AMD's HIP on Frontier, Frontier Application Readiness Kick-Off Workshop, Oct. 2019, Noel Clalmer 등
- KMP 알고리즘, 안경잡이개발자, https://blog.naver.com/ndb796/221240660061

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

ISBN 978-89-294-1716-1 (PDF)