EuroCC@Turkey
Parallel Computing on GPUs with CUDA

Dr. Özcan DÜLGER

Computer Engineering, Middle East Technical University

Computer Engineering, Artvin Coruh University

27 April 2022

EURO

TRUBA
Turkish Science e-Infrastructure

TÜBİTAK
ULAKBİM

Sabancı Üniversitesi

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

Summary of Topics

- Debugging and Profiling Performance
  - Debuggers:
    - CUDA Memcheck
    - CUDA GDB
  - Profilers:
    - Nvprof
- Performance Optimization and Efficiency
  - Memory Coalescing Access to Global Memory
  - Warp Divergence
  - Device Occupancy and SM Efficiency
  - Efficient Parallel Reduction Algorithm

Summary of Topics

- Some Libraries and Remaining Issues
  - Thrust Library
  - CUB Library
  - Comparison of three parallel reduction algorithms
  - Remaining Issues
    - Usage of cudaMemset
    - Initializing cost of the states of RNGs
    - L1 cache in Tesla K40
- CUDA Samples
  - Memory Coalescing Implementation of Metropolis Resampling
    - Coalesced variants of Metropolis Resampling
      - Metropolis-C1
      - Metropolis-C2
    - Comparisons of their performance
      - Bias and Mean Squared Error (MSE)
      - Target Tracking

Contents

- **Debugging and Profiling Performance**
- Performance Optimization and Efficiency
- Some Libraries and Remaining Issues
- CUDA Samples

TESLA K40

Property	Value	Property	Value
Architecture	Kepler	Global Memory	11520 MB
Number of SMX	15	Shared Memory	49152 Byte
CUDA Core	2880	L2 Cache	1572864 Byte
Core Clock	745 MHz	Segment Size	128 Byte
Max. Thread / SMX	2048	Warp Size	32
Max. Thread / Block	1024	Max. Block / SMX	16

Debugging and Profiling Performance

- CUDA Debugging Tools:
  - cuda-memcheck
  - cuda-gdb
  - Nsight

CUDA-Memcheck

- is used to detect memory access errors
- to detect run time errors

- is installed as part of the CUDA toolkit
- is supported on all CUDA supported platforms such as Windows, Linux and Android
- is supported on all CUDA capable GPUs with SM versions 3.5 and above
- is not supported on Virtual GPUs

- https://docs.nvidia.com/cuda/cuda-memcheck/

CUDA-Memcheck

- CUDA Memcheck Tools:
  - *Memcheck:* The memory access error and leak detection tool
  - *Racecheck:* The shared memory data access hazard detection tool
  - *Initcheck:* The unitialized device global memory access detection tool
  - *Synccheck:* The thread synchronization hazard detection tool

CUDA-Memcheck

- **memcheck tool** is a run time error detection tool for CUDA applications
- Error types:
  - *Memory access error:* due to out of bounds or misaligned accesses to memory (global, local, shared, global atomic accesses)
  - *Hardware exception:* are reported by the hardware error reporting mechanism
  - *Malloc/Free errors:* due to incorrect use of malloc()/free(). Allocated device memory not been freed
  - *Memory leaks error*: allocations of device memory using cudaMalloc() or using malloc() in device code not been freed
  - *CUDA API errors:* when a CUDA API call in the application returns a failure

Memcheck Tool

- *Memory access error: Out of bounds error*

```
__global__ void memory_access_error(int *A)
{
    *(A + 3) = 100;
}

void run_access_error(int *A_GPU)
{
    printf("Running memory_access_error kernel\n");
    memory_access_error<<<1,1>>>(A_GPU);
    printf("Ran memory_access_error kernel\n");
    printf("cudaGetLastError()-1 = %s\n",cudaGetErrorString(cudaGetLastError()));
    printf("Error: %s\n", cudaGetErrorString(cudaDeviceSynchronize()));
    printf("cudaGetLastError()-2 = %s\n",cudaGetErrorString(cudaGetLastError()));
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    run_access_error(A_GPU);
    cudaFree(A_GPU);
}
```

out of bounds error

Memcheck Tool

- *Memory access error: Out of bounds error*

```
$ nvcc -o memcheck1 memcheck1.cu
$ ./memcheck1
```

```
Running memory_access_error kernel
Ran memory_access_error kernel
cudaGetLastError()-1 = no error
Error: no error
cudaGetLastError()-2 = no error
```

Memcheck Tool

- *Memory access error: Out of bounds error*

$ cuda-memcheck memcheck1

Note: Depending on the SM type of your GPU, your system output may vary

```
========= CUDA-MEMCHECK
Running memory_access_error kernel
Ran memory_access_error kernel
cudaGetLastError()-1 = no error
========= Invalid __global__ write of size 4
=========     at 0x00000030 in memory_access_error(int*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7047a000c is out of bounds
```

Memory segment, type of access and size of access

PC of the instruction and kernel name

Address of access and error type

Thread id and block id

Memcheck Tool

- *Memory access error: Out of bounds error*

```
__global__ void memory_access_error(int *A)
{
    *(A + 3) = 100;
}

void run_access_error(int *A_GPU)
{
    printf("Running memory_access_error kernel\n");
    memory_access_error<<<1,1>>>(A_GPU);
    printf("Ran memory_access_error kernel\n");
    printf("cudaGetLastError()-1 = %s\n",cudaGetErrorString(cudaGetLastError()));
    printf("Error: %s\n", cudaGetErrorString(cudaDeviceSynchronize()));
    printf("cudaGetLastError()-2 = %s\n",cudaGetErrorString(cudaGetLastError()));
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    run_access_error(A_GPU);
    cudaFree(A_GPU);
}
```

out of bounds error

not capture the error yet

fail with an error code of unspecified launch failure

fail with an error code of unspecified launch failure and kernel terminates

fail with an error code of unspecified launch failure

Memcheck Tool

- *Memory access error: Out of bounds error*

```
========= CUDA-MEMCHECK
Running memory_access_error kernel
Ran memory_access_error kernel
cudaGetLastError()-1 = no error
========= Invalid __global__ write of size 4
=========     at 0x00000030 in memory_access_error(int*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7047a000c is out of bounds
```

```
Error: unspecified launch failure
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaDeviceSynchronize.
```

```
cudaGetLastError()-2 = unspecified launch failure
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaGetLastError.
```

```
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaFree.
```

Memcheck Tool

- *Memory access error: Misaligned accesses error*

```
__global__ void memory_access_error(int *A)
{
    *(int*) ((char*)&A[0] + 1) = 100;
}

void run_access_error(int *A_GPU)
{
    printf("Running memory_access_error kernel\n");
    memory_access_error<<<1,1>>>(A_GPU);
    printf("Ran memory_access_error kernel\n");
    printf("cudaGetLastError()-1 = %s\n",cudaGetErrorString(cudaGetLastError()));
    printf("Error: %s\n", cudaGetErrorString(cudaDeviceSynchronize()));
    printf("cudaGetLastError()-2 = %s\n",cudaGetErrorString(cudaGetLastError()));
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    run_access_error(A_GPU);
    cudaFree(A_GPU);
}
```

misaligned accesses

must be multiple of 4

Memcheck Tool

- *Memory access error: Misaligned accesses error*

```
========= CUDA-MEMCHECK
Running memory_access_error kernel
Ran memory_access_error kernel
cudaGetLastError()-1 = no error
========= Invalid __global__ write of size 4
=========     at 0x00000030 in memory_access_error(int*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7047a0001 is misaligned
```

```
Error: unspecified launch failure
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaDeviceSynchronize.
```

```
cudaGetLastError()-2 = unspecified launch failure
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaGetLastError.
```

```
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API call to cudaFree.
```

Memcheck Tool

- *Memory access error: Malloc/free error*

```
__global__ void malloc_free_error()
{
    int* array = (int*)malloc(10*sizeof(int));
    free(array);
    free(array);
}

int main()
{
    malloc_free_error<<<1,2>>>();
    cudaDeviceSynchronize();
}
```

Each thread allocates a new device heap memory

Error! Trying to free allocated memory that has already been freed

Memcheck Tool

- *Memory access error: Malloc/free error*

```
__global__ void malloc_free_error()
{
    int* array = (int*)malloc(10*sizeof(int));
    free(array);
    free(array);
}

int main()
{
    malloc_free_error<<<1,2>>>();
    cudaDeviceSynchronize();
}
```

Type of error

```
CUDA-MEMCHECK
Malloc/Free error encountered : Double free
    at 0x000691c0
    by thread (1,0,0) in block (0,0,0)
    Address 0x70509f970
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x60)

Malloc/Free error encountered : Double free
    at 0x000691c0
    by thread (0,0,0) in block (0,0,0)
    Address 0x70509f920
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x60)

========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API cal
l to cudaDeviceSynchronize.
```

Memcheck Tool

- *Memory access error: Malloc/free error*

```
__global__ void malloc_free_error()
{
    int* array = (int*)malloc(10*sizeof(int));
    array = array + 1;
    free(array);

}

int main()
{
    malloc_free_error<<<1,2>>>();
    cudaDeviceSynchronize();
}
```

Each thread allocates a new device heap memory

Error! Trying to free pointer that was not returned by malloc()

Memcheck Tool

- *Memory access error: Malloc/free error*

```
__global__ void malloc_free_error()
{
    int* array = (int*)malloc(10*sizeof(int));
    array = array + 1;
    free(array);

}

int main()
{
    malloc_free_error<<<1,2>>>();
    cudaDeviceSynchronize();
}
```

Type of error

```
CUDA-MEMCHECK
Malloc/Free error encountered : Invalid pointer to free
    at 0x000691c0
    by thread (1,0,0) in block (0,0,0)
    Address 0x70509f974
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x40)
```

```
Malloc/Free error encountered : Invalid pointer to free
    at 0x000691c0
    by thread (0,0,0) in block (0,0,0)
    Address 0x70509f924
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x40)
```

```
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API cal
l to cudaDeviceSynchronize.
```

Memcheck Tool

- *Memory access error: Leak errors*

cuda-memcheck --leak-check full memcheck3

```
__global__ void error_free_error(int *A)
{
    A[0] = 1;
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    error_free_error<<<1,1>>>(A_GPU);
    cudaDeviceReset();
}
```

A_GPU is not freed. Causes leak error

Must be used in order to destroy the CUDA context

Memcheck Tool

- *Memory access error: Leak errors*

```
__global__ void error_free_error(int *A)
{
    A[0] = 1;
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    error_free_error<<<1,1>>>(A_GPU);
    cudaDeviceReset();
}
```

```
CUDA-MEMCHECK
Leaked 12 bytes at 0x7047a0000

LEAK SUMMARY: 12 bytes leaked in 1 allocations
ERROR SUMMARY: 0 errors
```

CUDA-Memcheck

- **initcheck tool** is a run time uninitialized device global memory access detector
- Can detect the uninitialized device global memory via:
  - device side writes
  - CUDA memcpy API call
  - CUDA memset API call
- only supports detecting accesses to device global memory
- must be used *--tool initcheck* option
- first run *cuda-memcheck* to be sure that your application is free of memory access errors

Initcheck Tool

- *Memset error*

```
__global__ void memset_error(int *A)
{
    A[threadIdx.x] += threadIdx.x;
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*8);
    cudaMemset(A_GPU,0,8);
    memset_error<<<1,8>>>(A_GPU);
    cudaDeviceSynchronize();
    cudaFree(A_GPU);
}
```

cuda-memcheck --tool initcheck initcheck1

Error occurred when initializing A_GPU. Third parameter must be given in terms of byte

Initcheck Tool

- *Memset error*

```
__global__ void memset_error(int *A)
{
    A[threadIdx.x] += threadIdx.x;
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*8);
    cudaMemset(A_GPU,0,8);
    memset_error<<<1,8>>>(A_GPU);
    cudaDeviceSynchronize();
    cudaFree(A_GPU);
}
```

```
CUDA-MEMCHECK
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (7,0,0) in block (0,0,0)
    Address 0x7047a001c

Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (6,0,0) in block (0,0,0)
    Address 0x7047a0018

Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (5,0,0) in block (0,0,0)
    Address 0x7047a0014

Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (4,0,0) in block (0,0,0)
    Address 0x7047a0010

Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (3,0,0) in block (0,0,0)
    Address 0x7047a000c

Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (2,0,0) in block (0,0,0)
    Address 0x7047a0008

ERROR SUMMARY: 6 errors
```

CUDA-GDB

- is the NVIDIA tool for debugging CUDA applications on Linux and QNX
- extension of GNU gdb
- supports debugging C/C++ and Fortran CUDA applications
- allows the users to:
  - set breakpoints
  - single-step CUDA applications
  - inspect and modify the memory
  - inspect and modify the variables of any given thread running on the hardware
- https://docs.nvidia.com/cuda/cuda-gdb

CUDA-GDB

```cpp
#include <iostream>

using namespace std;

__global__ void vector_add(float *A,float *B,float *C)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    C[tid] = A[tid] + B[tid];//Vector addition
}

int main(int argc, char **argv)
{
    int size = 32768;//Data size
    float *A_host = new float[size];//Host Array
    float *B_host = new float[size];//Host Array
    float *C_host = new float[size];//Host Array

    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
    cudaMalloc((void**)&A_GPU,sizeof(float)*size);
    cudaMalloc((void**)&B_GPU,sizeof(float)*size);
    cudaMalloc((void**)&C_GPU,sizeof(float)*size);

    dim3 threadsPerBlock(1024);//Number of threads in a block
    dim3 numBlocks(size/1024);//Number of blocks in a grid

    for(int counter = 0;counter < size; counter++)
    {
        A_host[counter] = counter+1;//Assigning numbers from 1 to size
        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
    }

    cudaMemcpy(A_GPU,A_host,sizeof(float)*size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*size,cudaMemcpyHostToDevice);

    vector_add<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);
    cudaDeviceSynchronize();//Waits until vector_add kernel completes its run

    cudaMemcpy(C_host,C_GPU,sizeof(float)*size,cudaMemcpyDeviceToHost);

    cudaError_t err = cudaGetLastError();
    if ( err != cudaSuccess )
        cout << "CUDA Error: " << cudaGetErrorString(err) << endl;

    delete[] A_host;
    delete[] B_host;
    delete[] C_host;
    cudaFree(A_GPU);
    cudaFree(B_GPU);
    cudaFree(C_GPU);
}
```

- Compile as **nvcc -g -G vector_add.cu -o vector_add**
  - forces -O0 compilation, very limited optimizations
  - makes the compiler include debug information in the executable
  - -g is for host debug, -G is for device debug

```cpp
#include <iostream>

using namespace std;

__global__ void vector_add(float *A,float *B,float *C)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    C[tid] = A[tid] + B[tid];//Vector addition
}

int main(int argc, char **argv)
{
    int size = 32768;//Data size
    float *A_host = new float[size];//Host Array
    float *B_host = new float[size];//Host Array
    float *C_host = new float[size];//Host Array

    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
    cudaMalloc((void**)&A_GPU,sizeof(float)*size);
    cudaMalloc((void**)&B_GPU,sizeof(float)*size);
    cudaMalloc((void**)&C_GPU,sizeof(float)*size);

    dim3 threadsPerBlock(1024);//Number of threads in a block
    dim3 numBlocks(size/1024);//Number of blocks in a grid

    for(int counter = 0;counter < size; counter++)
    {
        A_host[counter] = counter+1;//Assigning numbers from 1 to size
        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
    }

    cudaMemcpy(A_GPU,A_host,sizeof(float)*size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*size,cudaMemcpyHostToDevice);

    vector_add<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);
    cudaDeviceSynchronize();//Waits until vector_add kernel completes its run

    cudaMemcpy(C_host,C_GPU,sizeof(float)*size,cudaMemcpyDeviceToHost);

    cudaError_t err = cudaGetLastError();
    if ( err != cudaSuccess )
        cout << "CUDA Error: " << cudaGetErrorString(err) << endl;

    delete[] A_host;
    delete[] B_host;
    delete[] C_host;
    cudaFree(A_GPU);
    cudaFree(B_GPU);
    cudaFree(C_GPU);
}
```

CUDA-GDB

Starts debugger

- cuda-gdb vector_add
- (cuda-gdb) break main

  Breakpoint 1 at 0x402877: file vector_add.cu, line 13.
- (cuda-gdb) break vector_add

  Breakpoint 2 at 0x402c3b: file vector_add.cu, line 6.
- (cuda-gdb) break 9

  Breakpoint 3 at 0x402c52: file vector_add.cu, line 9.
- (cuda-gdb) run → Starts application

  Starting program:

  Breakpoint 1, main (argc=1, argv=0x7fffffffde38) at vector_add.cu:13
  13 int size = 32768;//Data size

CUDA-GDB

```
1 #include <iostream>
2
3 using namespace std;
4
5 __global__ void vector_add(float *A,float *B,float *C)
6 {
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8     C[tid] = A[tid] + B[tid];//Vector addition
9 }
10
```

- (cuda-gdb) continue ⟶ Advances the execution

```
Continuing.
[New Thread 0x7ffff5c12700 (LWP 11105)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0, sm 14, warp 0, lane 0]
Breakpoint 2, vector_add<<<(32,1,1),(1024,1,1)>>> (A=0x7047a0000, B=0x7047c0000, C=0x7047e0000) at vector_add.cu:7
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
```

- (cuda-gdb) info cuda threads ⟶ 15 SMx, each has 2048 threads

BlockIdx	ThreadIdx	To BlockIdx	ThreadIdx	Count	Filename
(0,0,0)	(0,0,0)	(29,0,0)	(1023,0,0)	30720	vector_add.cu

- (cuda-gdb) info cuda devices

Dev	PCI Bus/Dev ID	Name	Description	SM Type	SMs	Warps/SM	Lanes/Warp
0	05:00.0	Tesla K40c	GK110B	sm_35	15	64	32
1	04:00.0	Tesla K20c	GK110	sm_35	13	64	32

- (cuda-gdb) info cuda kernels

	Kernel	Parent	Dev	Grid	Status	GridDim	BlockDim	Invocation
*	0	- 0		1	Active	(32,1,1)	(1024,1,1)	vector_add(A=0x7047a0000, B=0x7047c0000, C=0x7047e0000)

CUDA-GDB

```
1 #include <iostream>
2
3 using namespace std;
4
5 __global__ void vector_add(float *A,float *B,float *C)
6 {
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8     C[tid] = A[tid] + B[tid];//Vector addition
9 }
10
```

- (cuda-gdb) cuda device sm warp lane block thread. ⟶ Current focus

  block (0,0,0), thread (0,0,0), device 0, sm 14, warp 0, lane 0
- (cuda-gdb) print blockDim

  $6 = {x = 1024, y = 1, z = 1}
- (cuda-gdb) print threadIdx

  $7 = {x = 0, y = 0, z = 0}
- (cuda-gdb) print A[1]

  $8 = 2
- (cuda-gdb) print tid

  $9 = <optimized out> ⟶ Instruction not executed yet
- (cuda-gdb) next

  8 C[tid] = A[tid] + B[tid];//Vector addition
- (cuda-gdb) print tid

  $10 = 0

CUDA-GDB

```
1 #include <iostream>
2
3 using namespace std;
4
5 __global__ void vector_add(float *A,float *B,float *C)
6 {
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8     C[tid] = A[tid] + B[tid];//Vector addition
9 }
10
```

- Notes:
- Software coordinates of the focus:
  - Thread
  - Block
  - Kernel
- Hardware coordinates of the focus:
  - Lane
  - Warp
  - SM
  - Device

- (cuda-gdb) print B[5]

  $11 = 7

- (cuda-gdb) print A[tid]

  $12 = 1

- (cuda-gdb) print C[tid]

  $13 = 0 → Instruction not executed yet

- (cuda-gdb) cuda kernel 0 block 1,0,0 thread 35,0,0 → Software coordinates

  [Switching focus to CUDA kernel 0, grid 1, block (1,0,0), thread (35,0,0), device 0, sm 13, warp 1, lane 3]

  7 int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

- (cuda-gdb) cuda device 0 sm 13 warp 1 lane 4

  [Switching focus to CUDA kernel 0, grid 1, block (1,0,0), thread (36,0,0), device 0, sm 13, warp 1, lane 4]

  7 int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

- (cuda-gdb) next

  8 C[tid] = A[tid] + B[tid];//Vector addition

- (cuda-gdb) print tid

  $14 = 1060

CUDA-GDB

```
1 #include <iostream>
2
3 using namespace std;
4
5 __global__ void vector_add(float *A,float *B,float *C)
6 {
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8     C[tid] = A[tid] + B[tid];//Vector addition
9 }
10
```

- (cuda-gdb) next

  Breakpoint 3, vector_add<<<(32,1,1),(1024,1,1)>>> (A=0x7047a0000, B=0x7047c0000, C=0x7047e0000) at vector_add.cu:9

  9 }

- (cuda-gdb) print C[tid]

  No symbol "tid" in current context.

- (cuda-gdb) print C[1060]

  $15 = 2123

- (cuda-gdb) print blockDim.x*blockIdx.x+threadIdx.x

  $16 = 1060

- (cuda-gdb) delete breakpoints

  Delete all breakpoints? (y or n) y

- (cuda-gdb) continue

  Continuing.

  [Thread 0x7ffff5c12700 (LWP 11105) exited]

  [Inferior 1 (process 11091) exited normally]

- Notes:
- disable command prevents hitting the breakpoint by additional threads
- kill command or pressing ‘CTRL+C’ kill the application instantly

Instead of this *disable* command can also be used here

Program exited normally

Nsight

- Provides debugging visually
- NVIDIA Nsight Visual Studio Edition
  - https://docs.nvidia.com/nsight-visual-studio-edition/cuda-debugger
- NVIDIA Nsight Eclipse Edition
  - https://developer.nvidia.com/nsight-eclipse-edition

Debugging and Profiling Performance

- Profiling tools enable you to understand and optimize the performance of your CUDA applications
- CUDA Profiling Tools:
  - Visual Profiler
    - allows to visualize and optimize the performance of your application
    - displays a timeline of your application's activity
    - analyzes your application to detect potential performance bottlenecks
    - the standalone version of it is ***nvvp***
  - nvprof
    - enables you to collect and view profiling data from the command-line
- https://docs.nvidia.com/cuda/profiler-users-guide

NVPROF

- enables you to collect and view profiling data from the command-line
- enables the collection of a timeline of CUDA-related activities on both CPU and GPU
  - kernel executions
  - memory transfers
  - memory set and CUDA API calls
  - events or metrics for CUDA kernels

NVPROF

- nvprof [options] [application] [application-arguments]
- Some of the options:
  - **query-events**: List all the events available on the device(s)
  - **query-metrics**: List all the metrics available on the device(s)
  - **print-api-summary**: Print a summary of CUDA runtime/driver API calls
  - **print-api-trace**: Print CUDA runtime/driver API trace
  - **print-gpu-summary**: Print a summary of the activities on the GPU (including CUDA kernels and memcpy's/memset's)
  - **print-gpu-trace**: Print individual kernel invocations (including CUDA memcpy's/memset's) and sort them in chronological order. In event/metric profiling mode, show events/metrics for each kernel invocation
  - **log-file**: Make nvprof send all its output to the specified file, or one of the standard channels

NVPROF

- Profiling modes:
  - **Summary Mode**: Default operating mode. Outputs a single result line for each kernel function and each type of CUDA memory copy/set operations
  - **GPU-Trace and API-Trace Modes**:
    - GPU-Trace mode provides a timeline of all activities taking place on the GPU in chronological order
    - API-trace mode shows the timeline of all CUDA runtime and driver API calls invoked on the host in chronological order
  - **Event/metric Summary Mode**: Events or metrics are profiled across all kernel executions
  - **Event/metric Trace Mode**: Event and metric values are shown for each kernel execution

NVPROF
An Example Code

```
5  __global__ void vector_add(float *A,float *B,float *C)
6  {
7      int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8      C[tid] = A[tid] + B[tid];//Vector addition
9  }
10
11 __global__ void vector_sub(float *A,float *B,float *C)
12 {
13     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
14     C[tid] = A[tid] - B[tid];//Vector subtraction
15 }
16
17 int main(int argc, char **argv)
18 {
19     int size = 32768;//Data size
20     float *A_host = new float[size];//Host Array
21     float *B_host = new float[size];//Host Array
22     float *C_host = new float[size];//Host Array
23
24     float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
25     cudaMalloc((void**)&A_GPU,sizeof(float)*size);
26     cudaMalloc((void**)&B_GPU,sizeof(float)*size);
27     cudaMalloc((void**)&C_GPU,sizeof(float)*size);
28
29     dim3 threadsPerBlock(1024);//Number of threads in a block
30     dim3 numBlocks(size/1024);//Number of blocks in a grid
31
32     for(int counter = 0;counter < size; counter++)
33     {
34         A_host[counter] = counter+1;//Assigning numbers from 1 to size
35         B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
36     }
37
38     cudaMemcpy(A_GPU,A_host,sizeof(float)*size,cudaMemcpyHostToDevice);
39     cudaMemcpy(B_GPU,B_host,sizeof(float)*size,cudaMemcpyHostToDevice);
40
41     vector_add<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);
42     cudaMemcpy(C_host,C_GPU,sizeof(float)*size,cudaMemcpyDeviceToHost);
43
44     vector_sub<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);
45     cudaMemcpy(C_host,C_GPU,sizeof(float)*size,cudaMemcpyDeviceToHost);
46
47     delete[] A_host;
48     delete[] B_host;
49     delete[] C_host;
50     cudaFree(A_GPU);
51     cudaFree(B_GPU);
52     cudaFree(C_GPU);
53 }
```

NVPROF

- Summary mode example: **nvprof ./vector_add_sub**

```
==22396== NVPROF is profiling process 22396, command: ./vector_add_sub
==22396== Profiling application: ./vector_add_sub
==22396== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.64%  31.552us         2  15.776us  15.424us  16.128us  [CUDA memcpy HtoD]
 43.64%  30.848us         2  15.424us  15.264us  15.584us  [CUDA memcpy DtoH]
  6.02%  4.2560us         1  4.2560us  4.2560us  4.2560us  vector_add(float*, float*, float*)
  5.70%  4.0320us         1  4.0320us  4.0320us  4.0320us  vector_sub(float*, float*, float*)

==22396== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.63%  80.621ms         3  26.874ms  2.1980us  80.616ms  cudaMalloc
  0.72%  585.52us       166  3.5270us       74ns  139.78us  cuDeviceGetAttribute
  0.30%  243.76us         4  60.939us  39.615us  90.819us  cudaMemcpy
  0.15%  123.96us         3  41.320us  3.3530us  111.96us  cudaFree
  0.09%  73.572us         2  36.786us  36.607us  36.965us  cuDeviceTotalMem
  0.07%  55.062us         2  27.531us  26.456us  28.606us  cuDeviceGetName
  0.04%  34.634us         2  17.317us  17.079us  17.555us  cudaLaunch
  0.00%  2.9150us         6     485ns      91ns  2.1350us  cudaSetupArgument
  0.00%  1.7640us         2     882ns     291ns  1.4730us  cuDeviceGetCount
  0.00%  1.2110us         2     605ns     264ns     947ns  cudaConfigureCall
  0.00%     917ns         4     229ns      94ns     530ns  cuDeviceGet
```

NVPROF

- Summary mode example: **nvprof --print-gpu-summary ./vector_add_sub**

```
==22759== NVPROF is profiling process 22759, command: ./vector_add_sub
==22759== Profiling application: ./vector_add_sub
==22759== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.75%  31.616us         2  15.808us  15.424us  16.192us  [CUDA memcpy HtoD]
 43.30%  30.592us         2  15.296us  15.168us  15.424us  [CUDA memcpy DtoH]
  6.16%  4.3520us         1  4.3520us  4.3520us  4.3520us  vector_add(float*, float*, float*)
  5.80%  4.0970us         1  4.0970us  4.0970us  4.0970us  vector_sub(float*, float*, float*)
```

- Summary mode example: **nvprof --print-api-summary ./vector_add_sub**

```
==22776== NVPROF is profiling process 22776, command: ./vector_add_sub
==22776== Profiling application: ./vector_add_sub
==22776== Profiling result:
==22776== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.36%  84.166ms         3  28.055ms  2.1710us  84.160ms  cudaMalloc
  1.03%  877.47us       166  5.2850us     191ns  208.59us  cuDeviceGetAttribute
  0.21%  182.32us         4  45.580us  23.194us  76.506us  cudaMemcpy
  0.14%  117.44us         3  39.145us  3.3210us  105.96us  cudaFree
  0.13%  108.66us         2  54.330us  53.118us  55.543us  cuDeviceTotalMem
  0.10%  82.461us         2  41.230us  38.965us  43.496us  cuDeviceGetName
  0.03%  23.811us         2  11.905us  6.1630us  17.648us  cudaLaunch
  0.00%  3.4820us         2  1.7410us     523ns  2.9590us  cuDeviceGetCount
  0.00%  2.9140us         6     485ns      92ns  2.1470us  cudaSetupArgument
  0.00%  2.0760us         4     519ns     238ns     916ns  cuDeviceGet
  0.00%  1.2160us         2     608ns     314ns     902ns  cudaConfigureCall
```

NVPROF

- GPU-Trace and API-Trace modes example: **nvprof --print-gpu-trace ./vector_add_sub**

```
==22852== NVPROF is profiling process 22852, command: ./vector_add_sub
==22852== Profiling application: ./vector_add_sub
==22852== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem* Name
2.23744s  16.160us                    -               -         -         -         - [CUDA memcpy HtoD]
2.23748s  15.393us                    -               -         -         -         - [CUDA memcpy HtoD]
2.23750s  4.3520us             (32 1 1)      (1024 1 1)        10        0B        0B vector_add(float*, float*, float*) [186]
2.23751s  15.169us                    -               -         -         -         - [CUDA memcpy DtoH]
2.23759s  4.0640us             (32 1 1)      (1024 1 1)        10        0B        0B vector_sub(float*, float*, float*) [192]
2.23759s  15.201us                    -               -         -         -         - [CUDA memcpy DtoH]
```

```
    Size  Throughput           Device   Context    Stream  Name
128.00KB  7.5539GB/s    Tesla K40c (0)         1         7  [CUDA memcpy HtoD]
128.00KB  7.9303GB/s    Tesla K40c (0)         1         7  [CUDA memcpy HtoD]
       -           -    Tesla K40c (0)         1         7  vector_add(float*, float*, float*) [186]
128.00KB  8.0474GB/s    Tesla K40c (0)         1         7  [CUDA memcpy DtoH]
       -           -    Tesla K40c (0)         1         7  vector_sub(float*, float*, float*) [192]
128.00KB  8.0304GB/s    Tesla K40c (0)         1         7  [CUDA memcpy DtoH]
```

NVPROF

- GPU-Trace and API-Trace modes example: **nvprof --print-api-trace ./vector_add_sub**

```
==22887== NVPROF is profiling process 22887, command: ./vector_add_sub
==22887== Profiling application: ./vector_add_sub
==22887== Profiling result:
   Start  Duration  Name
2.32905s  1.3060us  cuDeviceGetCount
2.32906s     234ns  cuDeviceGet
2.32916s     805ns  cuDeviceGet
2.32932s     273ns  cuDeviceGetCount
2.32932s     105ns  cuDeviceGet
2.32932s  31.157us  cuDeviceGetName
2.32935s  38.126us  cuDeviceTotalMem
2.32939s     150ns  cuDeviceGetAttribute
2.32939s      84ns  cuDeviceGetAttribute
2.32939s     100ns  cuDeviceGetAttribute
2.32939s     101ns  cuDeviceGetAttribute
2.32939s      80ns  cuDeviceGetAttribute
2.32939s  25.346us  cuDeviceGetAttribute
2.32942s     103ns  cuDeviceGetAttribute
2.32942s      82ns  cuDeviceGetAttribute
2.32942s      81ns  cuDeviceGetAttribute
2.32942s      88ns  cuDeviceGetAttribute
2.32942s      90ns  cuDeviceGetAttribute
```

```
<...more output...>
```

NVPROF

- Event/metric mode:
- **Event:** is a countable activity, action, or occurrence on a device and collected during kernel execution
- **Metric:** statistic of an application that is calculated from one or more events
- **nvprof --query-events** command is used to get lists of available events
- **nvprof --query-metrics** command is used to get lists of available metrics

NVPROF

- Event/metric summary mode example:
- **nvprof --events warps_launched --metrics gld_transactions ./vector_add_sub**
  - warps_launched: Number of warps launched
  - gld_transactions: Number of global memory load transactions

```
==23567== Profiling result:
==23567== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: vector_add(float*, float*, float*)
          1                            warps_launched        1024        1024        1024
        Kernel: vector_sub(float*, float*, float*)
          1                            warps_launched        1024        1024        1024

==23567== Metric result:
Invocations                                Metric Name                    Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: vector_add(float*, float*, float*)
          1                            gld_transactions              Global Load Transactions        2048        2048        2048
        Kernel: vector_sub(float*, float*, float*)
          1                            gld_transactions              Global Load Transactions        2048        2048        2048
```

Note: Threads in the warp are access to the global memory as a coalesced way. So each warp performs one global memory load transaction. Otherwise, the value of this metric will be larger (an issue of the following lectures).

NVPROF

- Event/metric trace mode:
  - In summary mode, aggregate-mode is ‘on’ in default and SM specific events are collected across all SMs on the GPU and are shown as a single result.
  - In trace mode, by specifying aggregate-mode as ‘off’, the values of the events of each SM are shown separately.

NVPROF

- Some of the metrics are:
  - **l1_cache_global_hit_rate**: Hit rate in L1 cache for global loads
  - **gld_transactions:** Number of global memory load transactions
  - **gld_transactions_per_request**: Average number of global memory load transactions performed for each global memory load
  - **sm_efficiency**: The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
  - **achieved_occupancy**: Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
  - **warp_execution_efficiency:** Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage
  - https://docs.nvidia.com/cuda/profiler-users-guide/#metrics-reference-3x

SUMMARY

- CUDA Debuggers and Profilers Tools
  - Debuggers:
    - CUDA Memcheck
    - CUDA GDB
  - Profilers:
    - Nvprof

References

- https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
- https://docs.nvidia.com/cuda/cuda-memcheck/index.html
- https://docs.nvidia.com/cuda/cuda-gdb
- https://docs.nvidia.com/nsight-visual-studio-edition/cuda-debugger
- https://developer.nvidia.com/nsight-eclipse-edition
- https://docs.nvidia.com/cuda/profiler-users-guide