



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Advanced Computer Architecture

## 高级计算机体系结构

---

### 第8讲：DLP and GPU (3)

profiling

翁跃

DCS5367, 11/23/2021

# Overview

---

- What is profile tool ?
- Why we need profile tool ?
- How to use nvprof ?

# Overview

---

- What is profile tool ? 🤔
- Why we need profile tool ?
- How to use nvprof ?

# What is profile tool ?

---

- A profiler can be applied to an individual method or at the scale of a module or program, to identify performance **bottlenecks** by making long-running code obvious.
- A profiler can be used to understand code from a timing point of view, with the objective of **optimizing** it to handle various runtime conditions or various loads.

— Wikipedia [1]

# What is profile tool ?

---

## Classification of profile tool:

- **Hardware-based:** [基于硬件]
  - rely on hardware performance counters to grant users the access to low-level activities, such as nvprof[2], rocprof[3] ...
- **Software-based:** [基于软件]
  - by leveraging binary rewriters or performance monitoring units and debug registers available only in CPU/GPU architectures, such as Intel Pin[4], debuggers GDB[5], NVBit[6], SASSI[7] ...
- **Compiler-based:** [基于编译器]
  - by adding instrumentation while compiling, such as CUDAAAdvisor[8], CUDA Flux[9] ...

# What is profile tool ?

---

## Classification of profile tool:

- **Hardware-based:** [基于硬件]

Lower overhead

more transparent

- rely on hardware performance counters to grant users the access to low-level activities, such as nvprof[2], rocprof[3] ...

- **Software-based:** [基于软件]

- by leveraging binary rewriters or performance monitoring units and debug registers available only in CPU architectures, such as Intel Pin[4], debuggers GDB[5], NVBit[6], SASSI[7] ...

- **Compiler-based:** [基于编译器]

- by adding instrumentation while compiling, such as CUDAAdvisor[8], CUDA Flux[9] ...

# Overview

---

- What is profile tool ?
- Why we need profile tool ? 🤨
- How to use nvprof ?

# Why we need profile tool ?

---

When I finish my code: 😎

When I find my code takes a long time to finish: 🙄

And it consumes very few system resources: 😞

- top/htop/nvidia-smi ...



PROFILE TOOL!





# Why we need profile tool ?

---

Program analysis tools are extremely important for understanding program behavior.

Computer architects need such tools to evaluate how well programs will perform on new architectures. [对计算机体系结构而言]







Software writers need tools to analyze their programs and identify critical sections of code. [对软件编程人员]

Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing...[对编译器设计者]

— ATOM, PLDI, '94

# Why we need profile tool ?

---

- Fledgling programmer and defective program 
- Identify limiters and optimization clues 
- Identify the most cost-effective optimization 
- Assess the impact changes 
- Cyclical modification and tuning 
- Make full use of machine performance 

# Overview

---

- What is profile tool ?
- Why we need profile tool ?
- How to use nvprof ? 😊

# nvprof overview

---

- The nvprof profiling tool enables you to collect and view profiling data from the **command-line**. [基于命令行]
- Profiling **options** are provided to nvprof through command-line options. [选项设置]
- nvprof enables the collection of a timeline of **CUDA-related activities on both CPU and GPU**, including kernel execution, memory transfers, memory set and CUDA API calls and events or metrics for CUDA kernels. [收集CUDA相关的活动]
- Profiling results are displayed in the **console** after the profiling data is collected, and may also be saved for later viewing by either nvprof or the **Visual Profiler (nvvp)**. [数据展现形式]

# Profiling modes

---

- Four profiling modes [4种性能剖析模式]
  - Summary mode: (default mode)
  - GPU-Trace and API-Trace mode: (`--print-gpu-trace`)
  - Event/metric summary mode: (`--events/--metrics`)
  - Event/metrics trace mode: (`--aggregate-mode off --events/--metrics`)

# (1) Summary mode

---

- Four profiling modes [4种性能剖析模式]
  - Summary mode: (default mode)
    - A single result line for each kernel function and each type of CUDA memory copy/set performed by the application.
    - For each kernel, nvprof outputs the total time of all instances of the kernel or type of memory copy as well as the average, minimum, and maximum time.
    - By default, nvprof also prints a summary of all the CUDA runtime/driver API calls.
  - GPU-Trace and API-Trace mode: (--print-gpu-trace)
  - Event/metric summary mode: (--events/--metrics)
  - Event/metrics trace mode: (--aggregate-mode off --events/--metrics)

# (1) Summary mode

- Four profiling modes [4种性能剖析模式]
  - Summary mode: (default mode)

```
$ nvprof matrixMul
[Matrix Multiply Using CUDA] - Starting...
==27694== NVPROF is profiling process 27694, command: matrixMul
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 35.35 GFlop/s, Time= 3.708 msec, Size= 131072000 Ops, WorkgroupSize= 1024
Checking computed result for correctness: OK

Note: For peak performance, please refer to the matrixMulCUBLAS example.
==27694== Profiling application: matrixMul
==27694== Profiling result:
Time(%)      Time          Calls          Avg           Min           Max           Name
 99.94%     1.11524s       301    3.7051ms    3.6928ms    3.7174ms    void matrixMulCUDA<int=32>(f
  0.04%     406.30us        2    203.15us    136.13us    270.18us    [CUDA memcpy HtoD]
  0.02%     248.29us        1    248.29us    248.29us    248.29us    [CUDA memcpy DtoH]

==27964== API calls:
Time(%)      Time          Calls          Avg           Min           Max           Name
 49.81%     285.17ms        3    95.055ms    153.32us    284.86ms    cudaMalloc
 25.95%     148.57ms        1    148.57ms    148.57ms    148.57ms    cudaEventSynchronize
 22.23%     127.28ms        1    127.28ms    127.28ms    127.28ms    cudaDeviceReset
  1.33%      7.6314ms       301    25.353us    23.551us    143.98us    cudaLaunch
  0.25%      1.4343ms        3    478.09us    155.84us    984.38us    cudaMemcpy
  0.11%      601.45us        1    601.45us    601.45us    601.45us    cudaDeviceSynchronize
```

## (2) GPU/API-Trace mode

---

- Four profiling modes [4种性能剖析模式]
  - Summary mode: (default mode)
  - GPU-Trace and API-Trace mode: (--print-gpu-trace/print-api-trace)
    - GPU-Trace mode provides a timeline of all activities taking place on the GPU in chronological order.
    - Each kernel execution and memory copy/set instance is shown in the output. For each kernel or memory copy, detailed information such as kernel parameters, shared memory usage and memory transfer throughput are shown.
  - Event/metric summary mode: (--events/--metrics)
  - Event/metrics trace mode: (--aggregate-mode off --events/--metrics)



# (2) GPU/API-Trace mode

- Four profiling modes [4种性能剖析模式]
  - GPU-Trace and API-Trace mode: (--print-gpu-trace/print-api-trace)

```
$ nvprof --print-gpu-trace matrixMul  
==27706== NVPROF is profiling process 27706, command: matrixMul
```

```
==27706== Profiling application: matrixMul  
[Matrix Multiply Using CUDA] - Starting...  
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0
```

```
MatrixA(320,320), MatrixB(640,320)  
Computing result using CUDA Kernel...  
done  
Performance= 35.36 GFlop/s, Time= 3.707 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block  
Checking computed result for correctness: OK
```

Note: For peak performance, please refer to the matrixMulCUBLAS example.

```
==27706== Profiling result:
```

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSMem*	Size	Throughput	Device	Context	Stream	Name
133.81ms	135.78us	-	-	-	-	-	409.60KB	3.0167GB/s	GeForce GT 640M	1	2	[CUDA memcpy HtoD]
134.62ms	270.66us	-	-	-	-	-	819.20KB	3.0267GB/s	GeForce GT 640M	1	2	[CUDA memcpy HtoD]
134.90ms	3.7037ms	(20 10 1)	(32 32 1)	29	8.1920KB	0B	-	-	GeForce GT 640M	1	2	void matrixMulCUDA<int
float*, float*, int, int) [94]												
138.71ms	3.7011ms	(20 10 1)	(32 32 1)	29	8.1920KB	0B	-	-	GeForce GT 640M	1	2	void matrixMulCUDA<int
float*, float*, int, int) [105]												
<...more output...>												
1.24341s	3.7011ms	(20 10 1)	(32 32 1)	29	8.1920KB	0B	-	-	GeForce GT 640M	1	2	void matrixMulCUDA<int
float*, float*, int, int) [2191]												
1.24711s	3.7046ms	(20 10 1)	(32 32 1)	29	8.1920KB	0B	-	-	GeForce GT 640M	1	2	void matrixMulCUDA<int
float*, float*, int, int) [2195]												
1.25089s	248.13us	-	-	-	-	-	819.20KB	3.3015GB/s	GeForce GT 640M	1	2	[CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.  
SSMem: Static shared memory allocated per CUDA block.  
DSMem: Dynamic shared memory allocated per CUDA block.

## (2) GPU/API-Trace mode

- Four profiling modes [4种性能剖析模式]
  - GPU-Trace and API-Trace mode: (--print-gpu-trace/print-api-trace)

```
$nvprom --print-api-trace matrixMul  
==27722== NVPROF is profiling process 27722, command: matrixMul
```

```
==27722== Profiling application: matrixMul  
[Matrix Multiply Using CUDA] - Starting...  
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0  
  
MatrixA(320,320), MatrixB(640,320)  
Computing result using CUDA Kernel...  
done  
Performance= 35.35 GFlop/s, Time= 3.708 msec, Size= 131072000 Ops, Workg  
Checking computed result for correctness: OK
```

Note: For peak performance, please refer to the matrixMulCUBLAS example.

```
==27722== Profiling result:
```

Start	Duration	Name
108.38ms	6.2130us	cuDeviceGetCount
108.42ms	840ns	cuDeviceGet
108.42ms	22.459us	cuDeviceGetName
108.45ms	11.782us	cuDeviceTotalMem
108.46ms	945ns	cuDeviceGetAttribute
149.37ms	23.737us	cudaLaunch (void matrixMulCUDA<int=32>(float*, float
149.39ms	6.6290us	cudaEventRecord
149.40ms	1.10156s	cudaEventSynchronize
<...more output...>		

# (3) Event/metric summary mode

---

- Four profiling modes [4种性能剖析模式]
  - Summary mode: (default mode)
  - GPU-Trace and API-Trace mode: (`--print-gpu-trace/print-api-trace`)
  - Event/metric summary mode: (`--events/--metrics`)
    - To see a list of all available events/metrics on a particular NVIDIA GPU
  - Event/metrics trace mode: (`--aggregate-mode off --events/--metrics`)

# (3) Event/metric summary mode

- Four profiling modes [4种性能剖析模式]
  - Event/metric summary mode: (--events/--metrics)

```
$ nvprof --events warps_launched,local_load --metrics ipc matrixMul  
[Matrix Multiply Using CUDA] - Starting...  
==6461== NVPROF is profiling process 6461, command: matrixMul
```

```
GPU Device 0: "GeForce GTX TITAN" with compute capability 3.5
```

```
MatrixA(320,320), MatrixB(640,320)
```

```
Computing result using CUDA Kernel...
```

```
==6461== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.  
done
```

```
Performance= 6.39 GFlop/s, Time= 20.511 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
```

```
Checking computed result for correctness: Result = PASS
```

```
NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```

```
==6461== Profiling application: matrixMul
```

```
==6461== Profiling result:
```

```
==6461== Event result:
```

```
Invocations
```

```
Device "GeForce GTX TITAN (0)"
```

```
Kernel: void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
```

Event Name	Min	Max	Avg
warps_launched	6400	6400	6400
local_load	0	0	0

```
==6461== Metric result:
```

```
Invocations
```

```
Device "GeForce GTX TITAN (0)"
```

```
Kernel: void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
```

Metric Name	Metric Description	Min
ipc	Executed IPC	1.282576

# (4) Event/metric trace mode

---

- Four profiling modes [4种性能剖析模式]
  - Summary mode: (default mode)
  - GPU-Trace and API-Trace mode: (`--print-gpu-trace/print-api-trace`)
  - Event/metric summary mode: (`--events/--metrics`)
  - Event/metrics trace mode: (`--aggregate-mode off --events/--metrics`)
    - In event/metric trace mode, event and metric values are shown for each kernel execution. By default, event and metric values are aggregated across all units in the GPU.

# (4) Event/metric trace mode

- Four profiling modes [4种性能剖析模式]
  - Event/metrics trace mode: (`--aggregate-mode off --events/--metrics`)

```
$ nvprof --aggregate-mode off --events local_load --print-gpu-trace matrixMul  
[Matrix Multiply Using CUDA] - Starting...  
==6740== NVPROF is profiling process 6740, command: matrixMul
```

```
GPU Device 0: "GeForce GTX TITAN" with compute capability 3.5
```

```
MatrixA(320,320), MatrixB(640,320)  
Computing result using CUDA Kernel...  
done
```

```
Performance= 16.76 GFlop/s, Time= 7.822 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block  
Checking computed result for correctness: Result = PASS
```

```
NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```

```
==6740== Profiling application: matrixMul
```

```
==6740== Profiling result:
```

Device	Context	Stream	Kernel	local_load (0)	local_load (1)	...
GeForce GTX TIT	1	7	void matrixMulCUDA<i	0	0	...
GeForce GTX TIT	1	7	void matrixMulCUDA<i	0	0	...

<...more output...>

# Profiling controls

---

- Profiling controls:
  - Timeout [超时]: -t. A timeout (in seconds) can be provided to nvprof. The CUDA application being profiled will be killed by nvprof after the timeout. Profiling result collected before the timeout will be shown.
  - Profiling scope [范围]: --devices <device IDs>, --kernels, --events, --metrics, --query-events, --query-metrics
  - Multiprocess profiling [多线程]: To profile all processes launched by an application, use the --profile-child-processes option.

# Profiling output

---

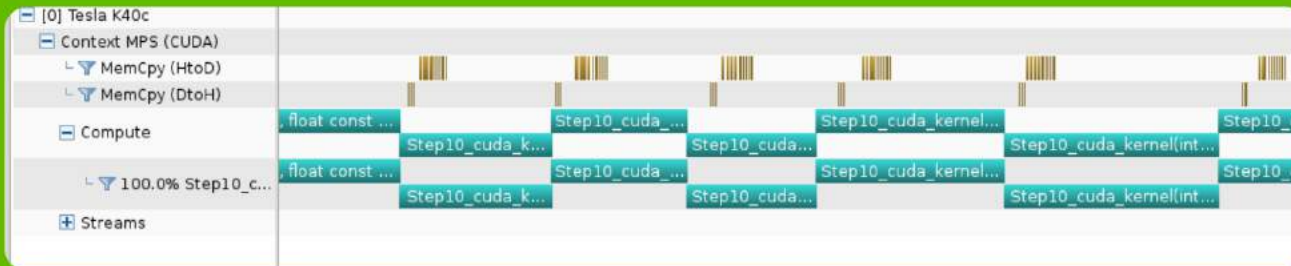
- Output:

- CSV: For each profiling mode, option `--csv` can be used to generate output in comma-separated values (CSV) format. The result can be directly imported to spreadsheet software such as Excel.
- Export/Import: For each profiling mode, option `--export-profile` can be used to generate a result file. This file is **not human-readable**, but can be imported back to `nvprof` using the option `--import-profile`, or into the **Visual Profiler**.



# Visual Profiler

## Timeline [时间线]



## Guided System

[分析]

### 1. CUDA Application Analysis

### 2. Performance-Critical Kernels

### 3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

Perform Latency Analysis

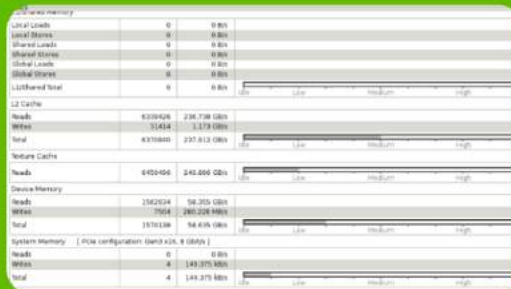
Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

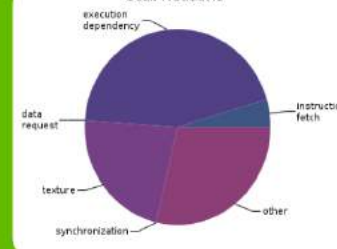
Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

## Analysis [分析]



### Stall Reasons



# nvprof vs Visual Profiler

---

## NVPROF

- Command-line Data Gathering
- Simple, high-level text output
- Gather hardware metrics
- Export data to other tools

## VISUAL PROFILER

- Graphical display of nvprof data
- “Big picture” analysis
- Very good visualization of data movement and kernel interactions
- Best run locally from your machine

# Example

---

## A simple example: vector addition

```
#include <stdio.h>
#define N 1048576

__global__ void add_vectors(int *a, int *b, int *c){
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < N) c[id] = a[id] + b[id];
}

int main(){
    size_t bytes = N*sizeof(int);

    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    for(int i=0; i<N; i++){
        A[i] = 1;
        B[i] = 2;
    }

    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    int thr_per_blk = 256;
    int blk_in_grid = ceil(float(N) / thr_per_blk);
    add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

# Example: code structure

```
#include <stdio.h>
#define N 1048576
```

```
__global__ void add_vectors(int *a, int *b, int *c){
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < N) c[id] = a[id] + b[id];
}
```

Vector addition kernel (GPU)

[核函数]

```
int main(){
    size_t bytes = N*sizeof(int);
```

```
int *A = (int*)malloc(bytes);
int *B = (int*)malloc(bytes);
int *C = (int*)malloc(bytes);
```

Allocate memory on CPU

[分配host端内存]

```
int *d_A, *d_B, *d_C;
cudaMalloc(&d_A, bytes);
cudaMalloc(&d_B, bytes);
cudaMalloc(&d_C, bytes);
```

Allocate memory on GPU

[分配device端内存]

```
for(int i=0; i<N; i++){
    A[i] = 1;
    B[i] = 2;
}
```

Initialize arrays on CPU

[数据赋值]

```
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
```

Copy data from CPU to GPU

[数据拷贝]

```
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);
```

Set configuration parameters and launch kernel

[核函数计算]

```
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

Copy data from GPU to CPU

[数据拷贝]

```
free(A);
free(B);
free(C);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Free memory on CPU and GPU

[释放内存]

```
return 0;
```

# Example: compile and run

---

## A simple example: vector addition

- Compile: `$ nvcc vector_add.cu -o vec_add`

- Run `nvprof`:

```
$ nvprof -s -o vec_add_cuda.nvvp ./vec_add
```

- `-s`: Print summary of profiling results
- `-o`: Export timeline file (to be opened later in NVIDIA Visual Profiler)

# Example: output

- Output on the terminal

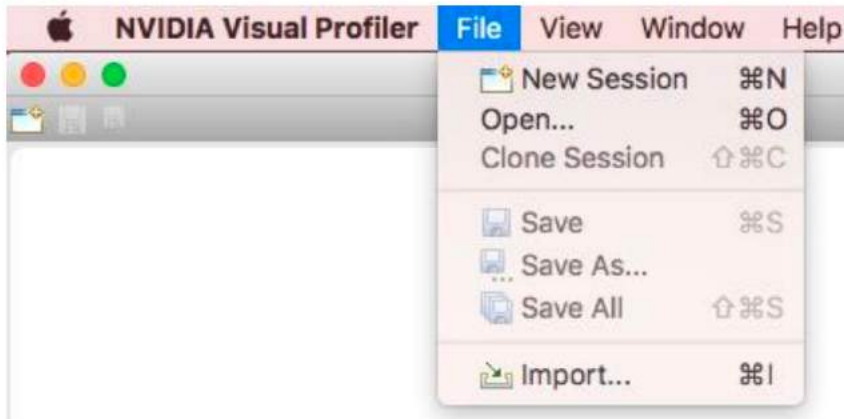
```
==174655== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	56.25%	463.36us	2	231.68us	229.66us	233.70us	[CUDA memcpy HtoD]
	41.59%	342.56us	1	342.56us	342.56us	342.56us	[CUDA memcpy DtoH]
	2.16%	17.824us	1	17.824us	17.824us	17.824us	add_vectors(int*, int*, int*)
API calls:	99.35%	719.78ms	3	239.93ms	1.1351ms	717.50ms	cudaMalloc
	0.23%	1.6399ms	96	17.082us	224ns	670.19us	cuDeviceGetAttribute
	0.17%	1.2559ms	3	418.64us	399.77us	454.40us	cudaFree
	0.16%	1.1646ms	3	388.18us	303.13us	550.07us	cudaMemcpy
	0.06%	412.85us	1	412.85us	412.85us	412.85us	cuDeviceTotalMem
	0.03%	182.11us	1	182.11us	182.11us	182.11us	cuDeviceGetName
	0.00%	32.391us	1	32.391us	32.391us	32.391us	cudaLaunchKernel
	0.00%	3.8960us	1	3.8960us	3.8960us	3.8960us	cuDeviceGetPCIBusId
	0.00%	2.2920us	3	764ns	492ns	1.1040us	cuDeviceGetCount
	0.00%	1.4090us	2	704ns	423ns	986ns	cuDeviceGet

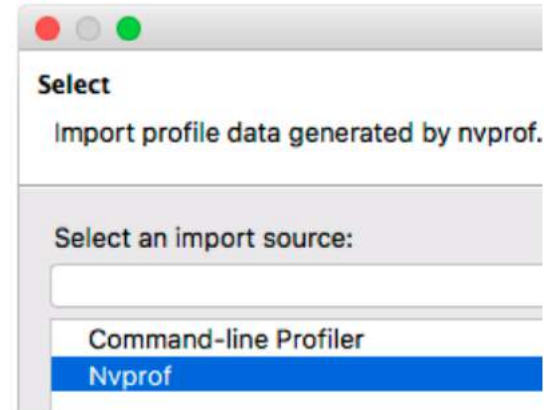
# Example: output

- Import output file to nvvp

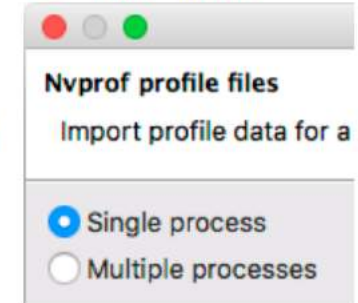
① File->Import



② Select "Nvprof" then "Next >"

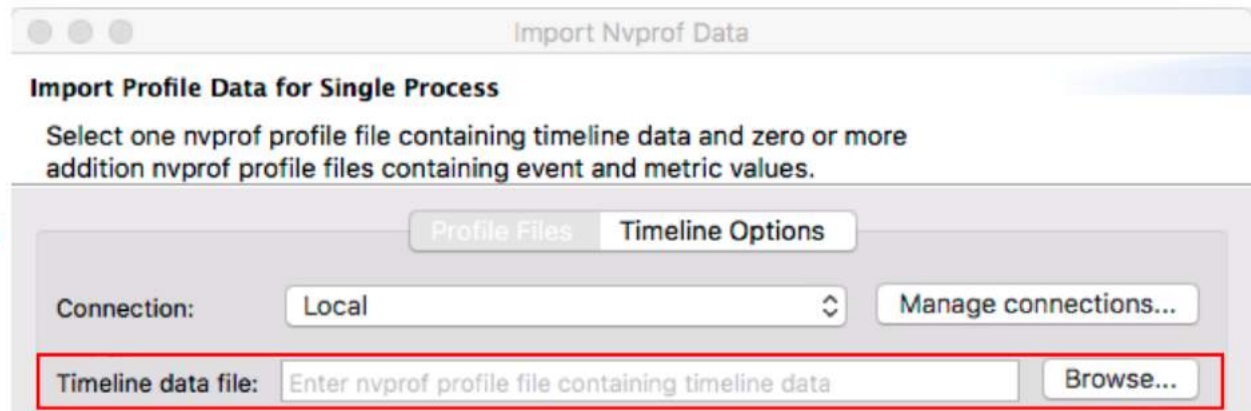


③ Select "Single Process" then "Next >"



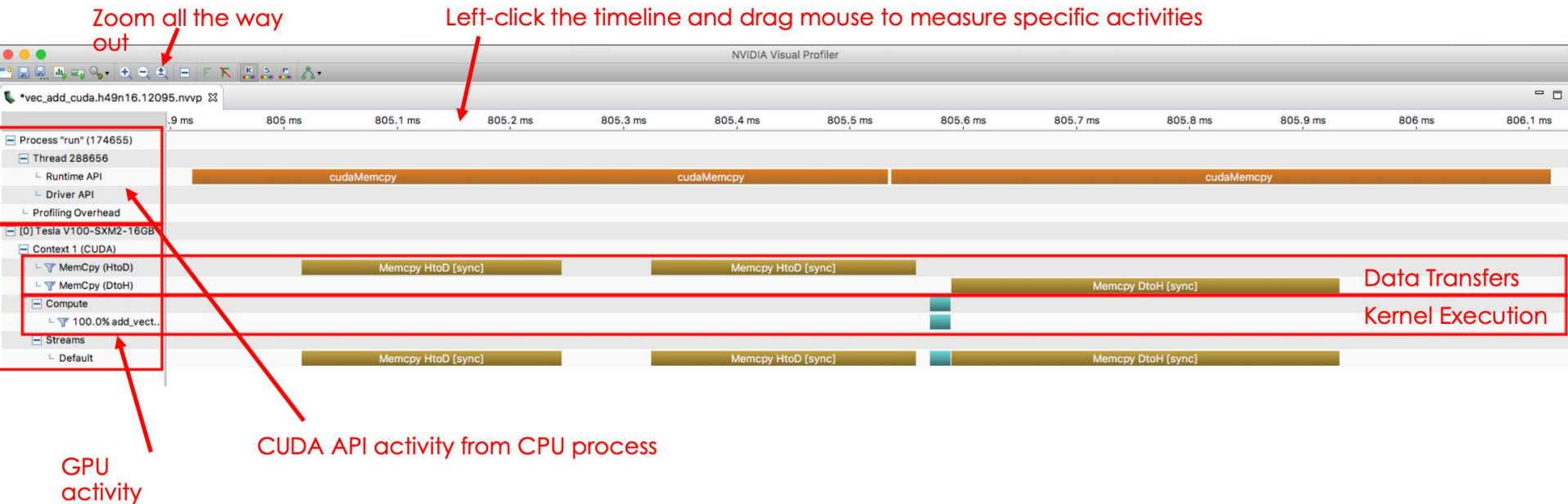
④

Click "Browse" next to "Timeline data file" to locate the .nvvp file on your local system, then click "Finish"



# Example: output

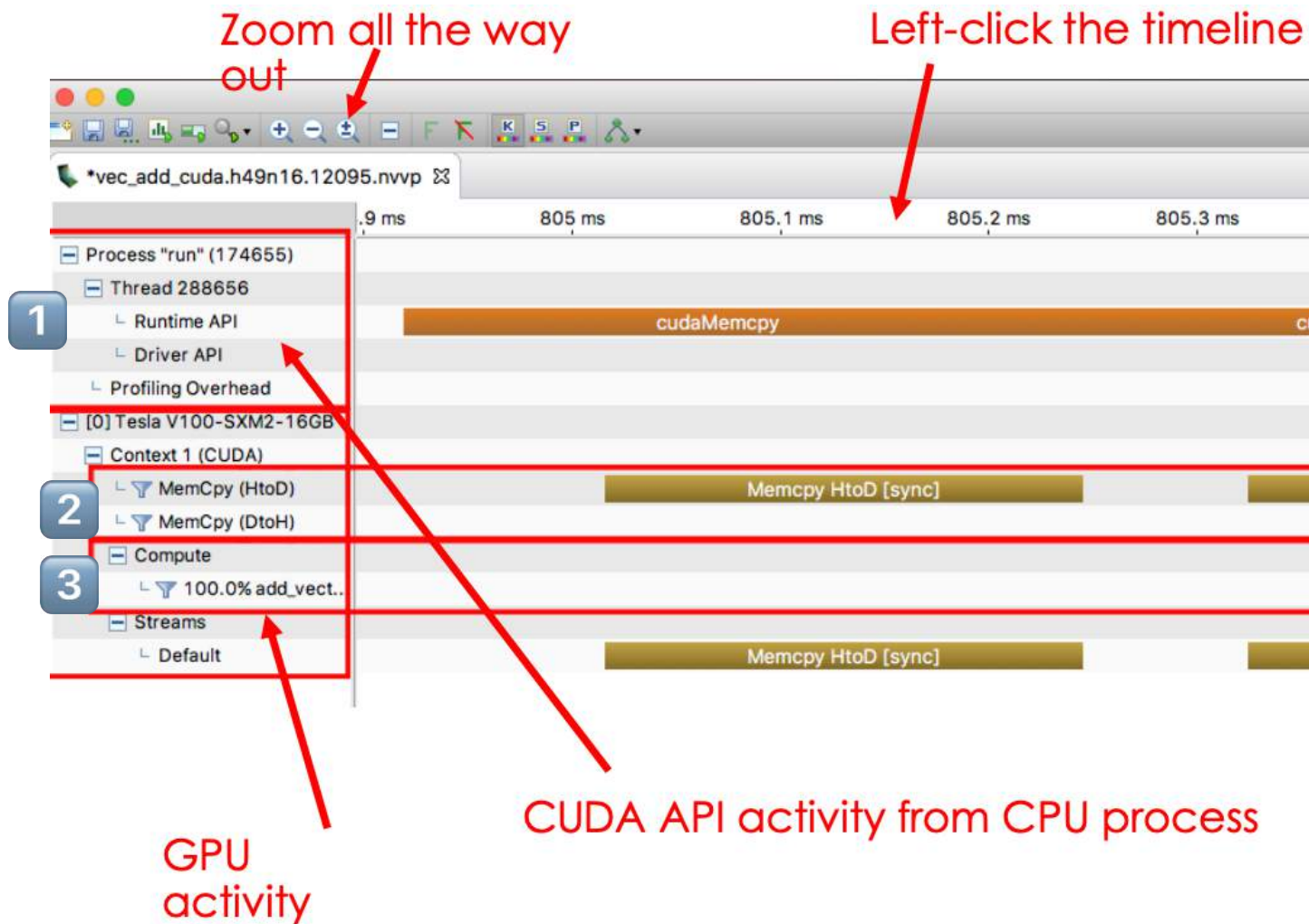
- Interface overview





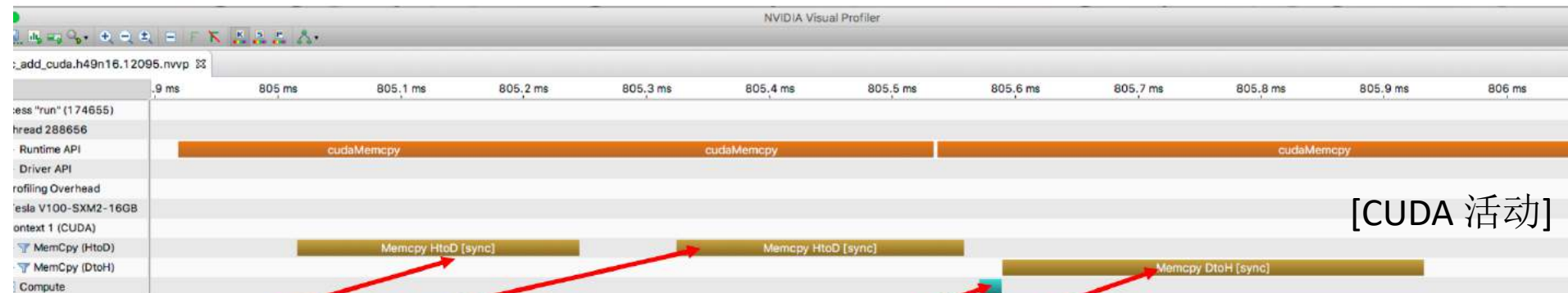
# Output on visual profile

- Interface description



# Output on visual profile

- Associate with code



```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
```

```
// Set execution configuration parameters
//   thr_per_blk: number of CUDA threads per grid block
//   blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );
```

```
// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);
```

```
// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	56.25%	463.36us	2	231.68us	229.66us	233.70us	[CUDA memcpy HtoD]
	41.59%	342.56us	1	342.56us	342.56us	342.56us	[CUDA memcpy DtoH]
	2.16%	17.824us	1	17.824us	17.824us	17.824us	add_vectors(int*, int*, int*)

# Output on visual profile

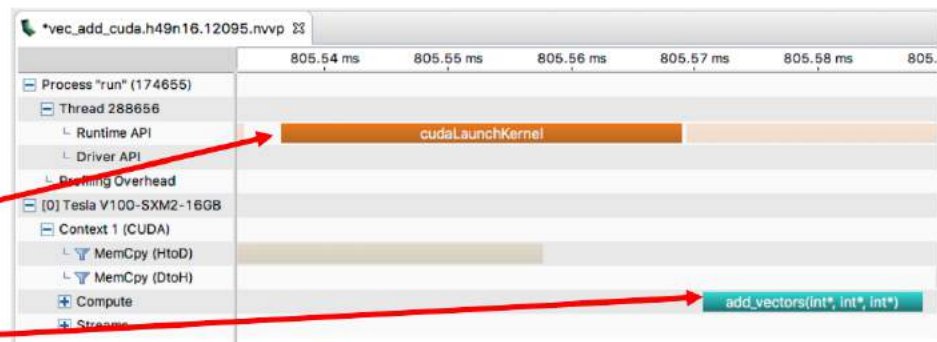
- Details information about operations

```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//   thr_per_blk: number of CUDA threads per grid block
//   blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```



Details about the kernel execution

[详细信息]

The screenshot shows the 'Properties' window for the 'add\_vectors(int\*, int\*, int\*)' kernel. The window has a title bar with a close button and a maximize button. The main content is a table with the following data:

add_vectors(int*, int*, int*)	
Queued	n/a
Submitted	n/a
Start	805.571 ms (805,570,598...
End	805.588 ms (805,588,422...
Duration	17.824 μs
Stream	Default
Grid Size	[ 4096,1,1 ]
Block Size	[ 256,1,1 ]
Registers/Thread	16
Shared Memory/Block	0 B
Launch Type	Normal
▼ Occupancy	
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

# Move forward (1): Specific metrics

```
$ nvprof -m [metrics list] ./app_name
```

[指定特定的指标]

```
Example: $ nvprof -m dram_utilization,l2_utilization,\  
double_precision_fu_utilization,achieved_occupancy ./redundant_mm 2048 100
```

```
==13250== NVPROF is profiling process 13250, command: ./redundant_mm 2048 100  
==13250== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.  
==13250== Profiling application: ./redundant_mm 2048 100  
(N = 2048) Max Total Time: 10.532436 Max GPU Time: 8.349185  
Rank 000, HWThread 002, GPU 0, Node h49n16 - Total Time: 10.532436 GPU Time: 8.349185  
==13250== Profiling result:  
==13250== Metric result:
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla V100-SXM2-16GB (0)"					
Kernel: volta_dgemm_64x64_nn					
100	dram_utilization	Device Memory Utilization	Low (1)	Low (2)	Low (1)
100	l2_utilization	L2 Cache Utilization	Low (2)	Low (2)	Low (2)
100	double_precision_fu_utilization	Double-Precision Function Unit Utilization	Max (10)	Max (10)	Max (10)
100	achieved_occupancy	Achieved Occupancy	0.114002	0.120720	0.118229

Ideally, something will be "High" or "Max". If everything is "Low", check you have enough work and check occupancy.

[核函数]

[指标名称]

[指标描述]

[最大/小/平均值]

# Common metrics

---

- Metrics [指标]

**inst\_executed**: # of instructions executed

**cf\_executed**: # of executed control-flow instructions

**ipc**: instructions executed per cycle

**sm\_efficiency**: % of time at least one warp is active

**achieved\_occupancy**: ratio of avg active warps per active cycle relative to the max # of warps supported on a SM

**l2\_utilization**: utilization level of L2 relative to peak

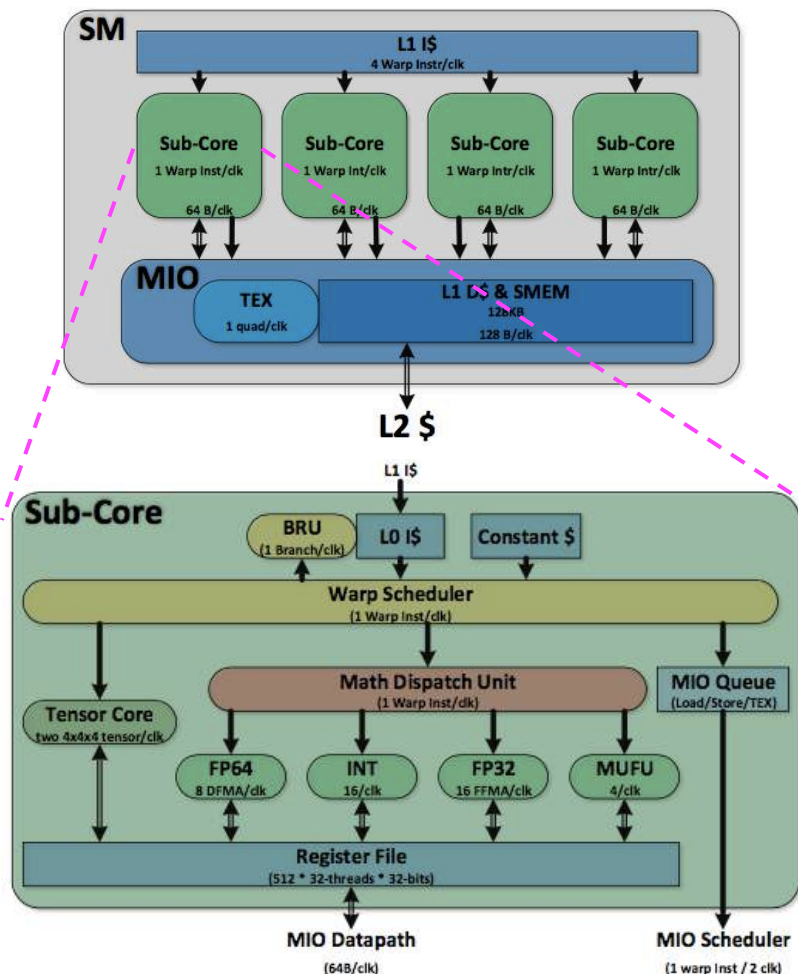
**dram\_utilization**: utilization level of DRAM to peak

**dram\_read\_throughput**: DRAM read throughput

**dram\_write\_throughput**: DRAM write throughput

# Common metrics

- Compute metrics [和计算相关的指标]



## Warp:

- sm\_efficiency
- achieved\_occupancy
- eligible\_warps\_per\_cycle
- warp{/\_nonpred}\_execution\_efficiency

## Instruction:

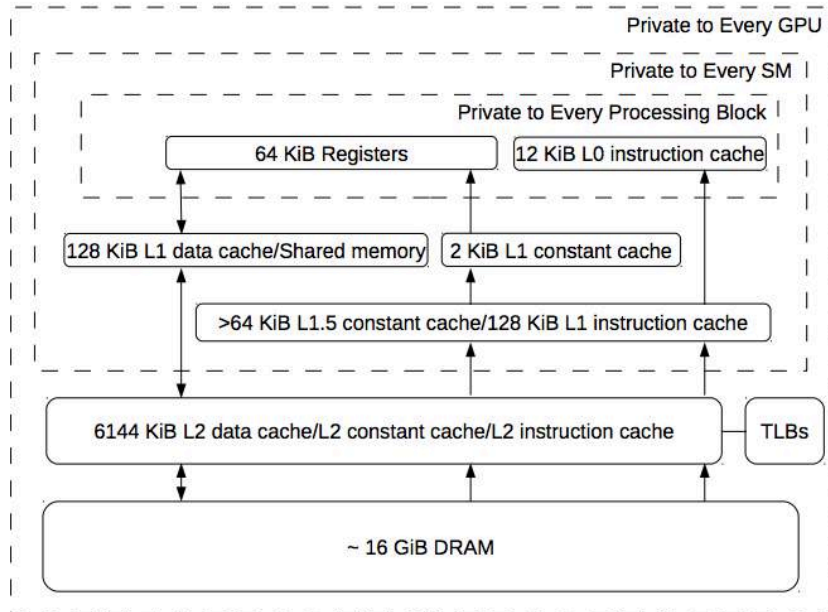
- ipc, issued\_ipc
- issue\_slot\_utilization
- stall\_\*
- branch\_efficiency

## Function Unit:

- {half/single/double}\_precision\_fu\_util
- {ldst/cf/special/tex}\_fu\_utilization

# Common metrics

- Memory metrics (1) [和访存相关的指标]



## SMEM:

- shared\_{load/store}\_transactions
- shared\_{load/store}\_throughput
- shared\_{efficiency/utilization}
- shared\_{load/store}\_trans\_per\_req

## L1 cache:

- tex\_cache\_transactions
- tex\_cache\_throughput
- tex\_cache\_hit\_rate
- tex\_utilization

## LMEM:

- local\_{load/store}\_transactions
- local\_{load/store}\_throughput
- local\_hit\_rate
- local\_memory\_overhead
- local\_{load/store}\_requests
- local\_{load/store}\_trans\_per\_req

# Common metrics

- Memory metrics (2) [和访存相关的指标]

## L2 cache:

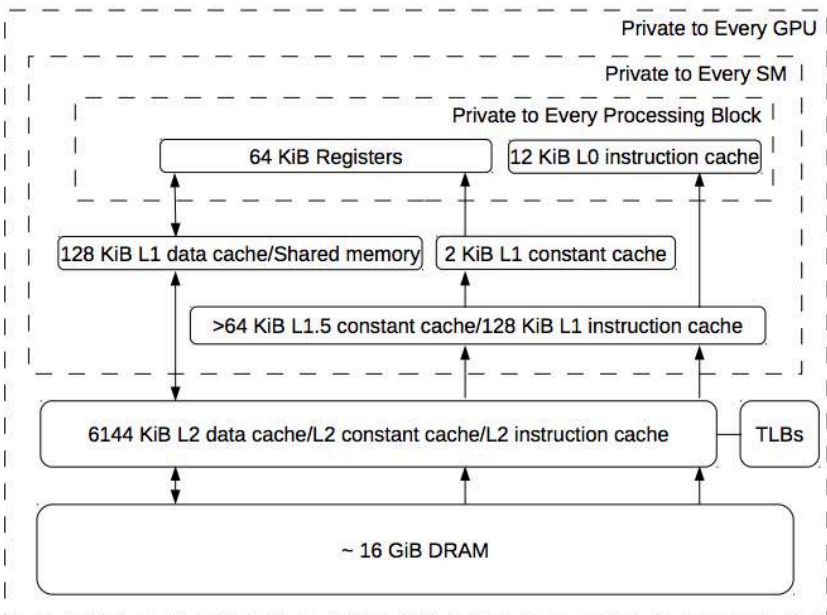
- $l2_{\{atomic/read/write\}}_{transactions}$ 
  - $l2_{tex}_{\{read/write\}}_{transactions}$
- $l2_{\{atomic/read/write\}}_{throughput}$ 
  - $l2_{tex}_{\{read/write\}}_{throughput}$
- $l2_{utilization}$ 
  - $l2_{tex}_{hit\_rate}$
- $l2_{\{global/local\}}_{load\_bytes}$
- $l2_{\{global\_atomic/local\_global\}}_{st\_Bs}$

## DRAM:

- $dram_{\{read/write\}}_{transactions}$
- $dram_{\{read/write\}}_{throughput}$
- $dram_{utilization}$
- $dram_{\{read/write\}}_{bytes}$

## Global:

- $\{gld/gst\}_{\{transactions/throughput\}}$
- $\{gld/gst\}_{requested\_throughput}$
- $\{gld/gst\}_{efficiency}$
- $\{gld/gst\}_{transactions\_per\_request}$





# More details about metrics

---

- More details can be found in ...

Metric Name	Description
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
atomic_transactions	Global memory atomic and reduction transactions
atomic_transactions_per_request	Average number of global memory atomic and reduction transactions performed for each atomic and reduction instruction
branch_efficiency	Ratio of branch instruction to sum of branch and divergent branch instruction
cf_executed	Number of executed control-flow instructions
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10
cf_issued	Number of issued control-flow instructions
double_precision_fu_utilization	The utilization level of the multiprocessor function units that execute double-precision floating-point instructions on a scale of 0 to 10

<https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference-7x>

# Move forward (2): Specific kernel(s)

---

```
$ nvprof -kernels :::1 output_name.nvprof ./app_name
```



[指定特定的核函数]

(context:stream:kernel:invocation)

Record metrics for only the **first** invocation of each kernel.

# Move forward (3): Analyse metrics

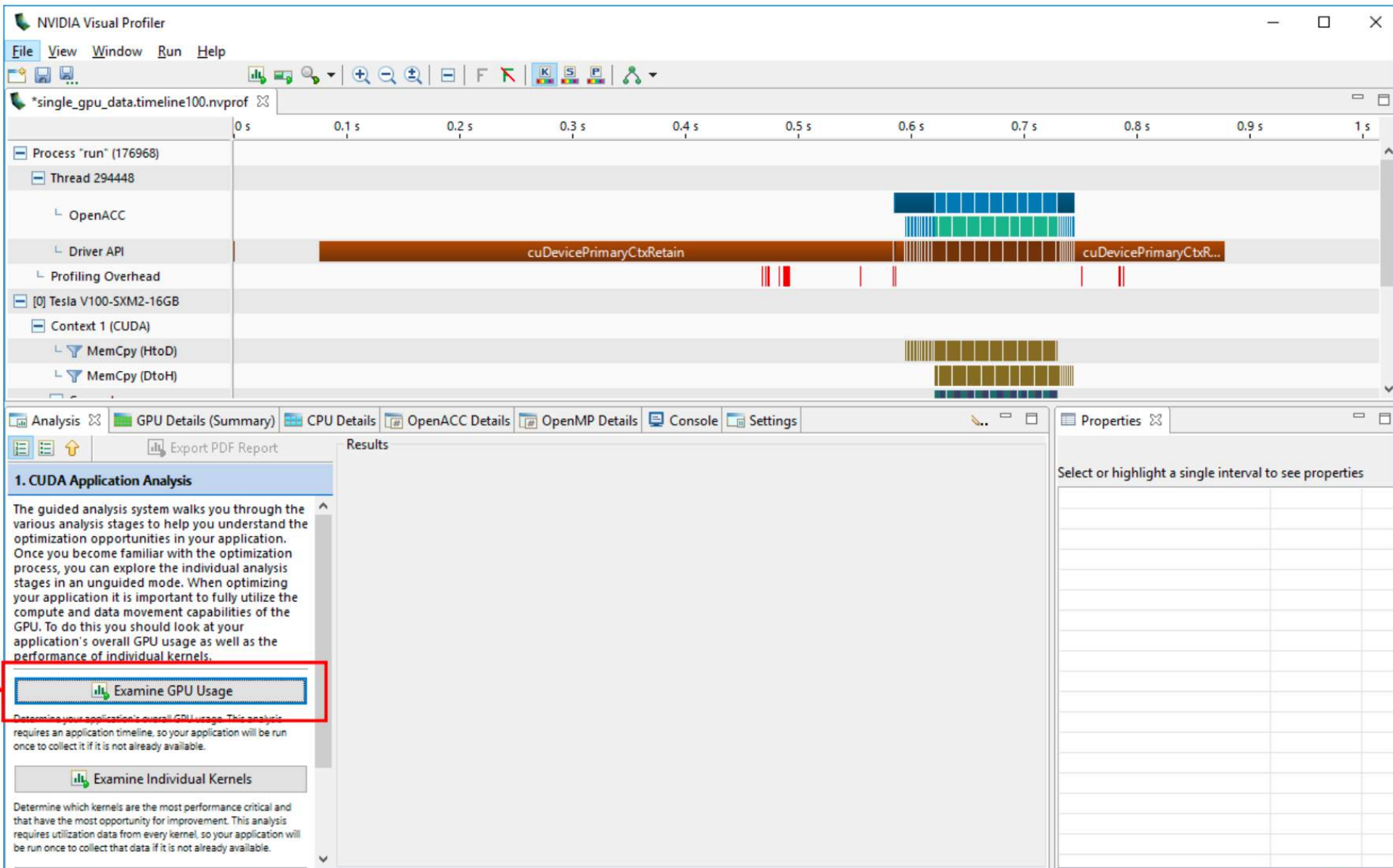
---

```
$ nvprof --analysis-metrics output_name.nvprof ./app_name [分析指标]
```

Example: `nvprof --analysis-metrics single_gpu_data.metrics100.nvprof ./run`

# Move forward (3): Analyse metrics

- High-level analysis



The screenshot displays the NVIDIA Visual Profiler interface. The top section shows a timeline for a process named 'run' (176968), with a thread 294448. The timeline includes categories like OpenACC, Driver API, Profiling Overhead, and MemCpy (HtoD/DtoH). A specific kernel, 'cuDevicePrimaryCbRetain', is highlighted in orange, spanning from approximately 0.1s to 0.8s. The bottom section shows the 'Analysis' tab, with 'GPU Details (Summary)' selected. The '1. CUDA Application Analysis' section is visible, containing a red-bordered box around the 'Examine GPU Usage' button. The text below the button reads: 'Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.'

**1. CUDA Application Analysis**

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.

**Examine GPU Usage**

Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

**Examine Individual Kernels**

Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.

# Move forward (3): Analyse metrics

- High-level analysis

**Most applications will see these.**

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

**Examine Individual Kernels**

You can also examine the performance of individual kernels to expose additional optimization opportunities.

**May indicate insufficient amount of work.**

**Results**

- Low Memcpy/Kernel Overlap** [ 0 ns / 8.93188 ms = 0% ]  
The percentage of time when memcpy is being performed in parallel with kernel is low.
- Low Kernel Concurrency** [ 0 ns / 97.2522 ms = 0% ]  
The percentage of time when two kernels are being executed in parallel is low.
- Low Memcpy Throughput** [ 6.775 MB/s avg, for memcpys accounting for 3.5% of all memcpy time ]  
The memory copies are not fully using the available host to device bandwidth.
- Low Memcpy Overlap** [ 0 ns / 3.0515 ms = 0% ]  
The percentage of time when two memory copies are being performed in parallel is low.
- Low Compute Utilization** [ 97.2522 ms / 877.80852 ms = 11.1% ]  
The multiprocessors of one or more GPUs are mostly idle.

**Compute Utilization**  
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels ex

**NVLink Analysis**  
The following NVLink topology diagram shows logical NVLink connections between GPUs and CPUs. A logical N

# Move forward (3): Analyse metrics

- Analyse individual kernels

The screenshot displays the NVIDIA Visual Profiler interface. The left sidebar contains a navigation menu with the following items:

- Analysis
- GPU Details (Summary)
- CPU Details
- OpenACC Details
- OpenMP Details
- Console
- Settings

The main content area is divided into two sections:

**1. CUDA Application Analysis**

**2. Check Overall GPU Usage**

The text under section 2 reads: "The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization."

A red box highlights a button labeled "Examine Individual Kernels" with a red arrow pointing to it from the left. Below this button, a note states: "You can also examine the performance of individual kernels to expose additional optimization opportunities."

The right pane, titled "Results", lists several performance metrics:

- Low Memcpy/Kernel Overlap** [ 0 ns / 8.93188 ms = 0% ]  
The percentage of time when memcopy is being performed in parallel with kernel is low.
- Low Kernel Concurrency** [ 0 ns / 97.2522 ms = 0% ]  
The percentage of time when two kernels are being executed in parallel is low.
- Low Memcpy Throughput** [ 6.775 MB/s avg, for memcpys accounting for 3.5% of all memcopy time ]  
The memory copies are not fully using the available host to device bandwidth.
- Low Memcpy Overlap** [ 0 ns / 3.0515 ms = 0% ]  
The percentage of time when two memory copies are being performed in parallel is low.
- Low Compute Utilization** [ 97.2522 ms / 877.80852 ms = 11.1% ]  
The multiprocessors of one or more GPUs are mostly idle.

Below the metrics, there are sections for "Compute Utilization" and "NVLink Analysis".

# Move forward (3): Analyse metrics

- Kernel optimization priorities [核函数优化优先级]

The screenshot displays the NVIDIA Visual Profiler interface. The top navigation bar includes tabs for Analysis, GPU Details (Summary), CPU Details, OpenACC Details, OpenMP Details, Console, and Settings. The left sidebar shows a tree view with '1. CUDA Application Analysis' and '2. Performance-Critical Kernels'. The main content area is titled 'Results' and contains a section for 'Kernel Optimization Priorities'. This section includes a descriptive paragraph and a table of kernel instances. A red box highlights the table, and a red arrow points from the 'Perform Kernel Analysis' button to the table. Another red box highlights the 'Perform Additional Analysis' button.

**1. CUDA Application Analysis**

**2. Performance-Critical Kernels**

The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.

[Perform Kernel Analysis](#)

Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.

[Perform Additional Analysis](#)

You can collect additional information to help identify kernels with potential performance problems. After running this analysis, select any of the new results at right to highlight the individual kernels for which the analysis applies.

**Kernel Optimization Priorities**

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[ 100 kernel instances ] main_123_gpu
66	[ 100 kernel instances ] main_134_gpu
37	[ 100 kernel instances ] main_127_gpu_red
5	[ 100 kernel instances ] main_148_gpu
2	[ 100 kernel instances ] main_142_gpu

# Move forward (3): Analyse metrics

- Memory bandwidth analysis [内存带宽分析]

The screenshot displays the NVIDIA Nsight Compute interface. On the left, a sidebar lists analysis steps: 1. CUDA Application Analysis, 2. Performance-Critical Kernels, and 3. Compute, Bandwidth, or Latency Bound. The third step is selected, and a red box highlights the 'Perform Memory Bandwidth Analysis' button. Below this, a text box explains that the first step is to determine if performance is bounded by computation, memory bandwidth, or latency, and that the results indicate the performance of kernel 'main\_123\_gpu' is most likely limited by memory bandwidth. Further down, another red box highlights the 'Perform Memory Bandwidth Analysis' button again, with a red arrow pointing to it. Below this are buttons for 'Perform Compute Analysis' and 'Perform Latency Analysis'. The main area shows the 'Results' tab with a title 'Kernel Performance Is Bound By Memory Bandwidth'. The text explains that for device 'Tesla V100-SXM2-16GB', the kernel's compute utilization is significantly lower than its memory utilization, indicating that performance is limited by the memory system. A bar chart shows utilization percentages for 'Compute' and 'Memory (Device)'. The 'Compute' bar is composed of three segments: Memory operations (purple, ~10%), Control-flow operations (cyan, ~5%), and Arithmetic operations (blue, ~10%). The 'Memory (Device)' bar is a single blue segment representing ~85% utilization. A legend on the right identifies the colors: Memory operations (purple), Control-flow operations (cyan), Arithmetic operations (blue), and Memory (Device) (blue).

**1. CUDA Application Analysis**

**2. Performance-Critical Kernels**

**3. Compute, Bandwidth, or Latency Bound**

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "main\_123\_gpu" is most likely limited by memory bandwidth.

**Perform Memory Bandwidth Analysis**

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

**Perform Compute Analysis**

**Perform Latency Analysis**

Compute and instruction and memory latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

**Results**

**i Kernel Performance Is Bound By Memory Bandwidth**

For device "Tesla V100-SXM2-16GB" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.

**Utilization**

Category	Memory operations	Control-flow operations	Arithmetic operations	Memory (Device)
Compute	~10%	~5%	~10%	~0%
Memory (Device)	~0%	~0%	~0%	~85%

Legend:

- Memory operations
- Control-flow operations
- Arithmetic operations
- Memory (Device)



# Move forward (3): Analyse metrics

- Optimization suggestions [优化建议]

Export PDF Report

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results at right indicate that the kernel is limited by the bandwidth available to the device memory.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Results

**⚠ Global Memory Alignment and Access Pattern**

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. The array per assembly instruction.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

Line / File	poisson2d.c - \gpfs\wolf\gen110\scratch\j2k\nvidia_profilers\jacobi\3_single_gpu_data
126	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4712194 L2 transactions for 524032 total executions]
126	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4712194 L2 transactions for 524032 total executions]
126	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4712194 L2 transactions for 524032 total executions]
126	Global Store L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4712194 L2 transactions for 524032 total executions]
126	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4712194 L2 transactions for 524032 total executions]
127	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4712194 L2 transactions for 524032 total executions]

**⚠ GPU Utilization Is Limited By Memory Bandwidth**

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows utilization of each memory type relative to the maximum throughput supported by the memory. The results show that the kernel's performance is potentially limited by the bandwidth available from one or more of the memories on the device.

Optimization: Try the following optimizations for the memory with high bandwidth utilization.

- Shared Memory - If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieved 2x throughput.
- L2 Cache - Align and block kernel data to maximize L2 cache efficiency.
- Unified Cache - Reallocate texture data to shared or global memory. Resolve alignment and access pattern issues for global loads and stores.
- Device Memory - Resolve alignment and access pattern issues for global loads and stores.
- System Memory (via PCIe) - Make sure performance critical data is placed in device or shared memory.

# Move forward (3): Analyse metrics

- Compute analysis [计算资源分析]

The screenshot displays the NVIDIA Nsight Compute application interface. The sidebar on the left contains the following sections:

- 1. CUDA Application Analysis
- 2. Performance-Critical Kernels
- 3. Compute, Bandwidth, or Latency Bound

The text under section 3 explains the analysis process: "The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel 'volta\_dgemm\_64x64\_nn' is most likely limited by compute." Below this text are three buttons: "Perform Compute Analysis" (highlighted with a red box and arrow), "Perform Latency Analysis", and "Perform Memory Bandwidth Analysis". A "Rerun Analysis" button is at the bottom.

The main results area, titled "Results", contains the following information:

- Kernel Performance Is Bound By Compute**
- For device "Tesla V100-SXM2-16GB" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

A bar chart shows the utilization for two categories:

Category	Utilization (%)
Function Unit (Double)	~95%
Memory (Texture)	~25%

# Move forward (3): Analyse metrics

- Optimization suggestions [优化建议]

**GPU Utilization Is Limited By Function Unit Usage**

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is potentially limited by overuse of the following function units: Double.

- Load/Store - Load and store instructions for shared and constant memory.
- Texture - Load and store instructions for local, global, and texture memory.
- Half - Half-precision floating-point arithmetic instructions.
- Single - Single-precision integer and floating-point arithmetic instructions.
- Double - Double-precision floating-point arithmetic instructions.
- Special - Special arithmetic instructions such as sin, cos, popc, etc.
- Control-Flow - Direct and indirect branches, jumps, and calls.

**Instruction Execution Counts**

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The

Instruction Class	Utilization Level
Load/Store	Low
Texture	Low
Half	Low
Single	Low
Double	High
Special	Low
Control-Flow	Low

# Move forward (3): Analyse metrics

- Latency analysis [延迟分析]

The screenshot displays a software interface for performance analysis. On the left, a sidebar contains a list of analysis steps: 1. CUDA Application Analysis, 2. Performance-Critical Kernels, and 3. Compute, Bandwidth, or Latency Bound. The third step is selected, and a sub-section titled 'Perform Latency Analysis' is highlighted with a red box. Below this, there are buttons for 'Perform Compute Analysis', 'Perform Memory Bandwidth Analysis', and 'Rerun Analysis'. The main area on the right, titled 'Results', contains a red-bordered box with the following text: 'Kernel Performance Is Bound By Instruction And Memory Latency. This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla V100-SXM2-16GB". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.' Below the text is a bar chart showing utilization percentages for 'Compute' and 'Memory (Device)'. The 'Compute' bar is composed of segments for Memory operations (purple), Control-flow operations (cyan), Arithmetic operations (dark blue), and Memory (Device) (blue). The 'Memory (Device)' bar is also composed of these segments. The y-axis is labeled 'Utilization' and ranges from 0% to 100%.

Category	Utilization (%)
Compute	~10%
Memory (Device)	~5%

# Move forward (3): Analyse metrics

- Optimization suggestions [优化建议]

The screenshot displays the NVIDIA Nsight Systems interface. The left sidebar shows a navigation menu with four items: 1. CUDA Application Analysis, 2. Performance-Critical Kernels, 3. Compute, Bandwidth, or Latency Bound, and 4. Instruction and Memory Latency. A red arrow points from item 3 to a warning message in the main results pane. The warning message is titled "Grid Size Too Small To Hide Compute And Memory Latency" and contains the following text: "The kernel does not execute enough blocks to hide memory and operation latency. Typically the kernel grid size must be large enough to fill the GPU with multiple 'waves' of blocks. Based on theoretical occupancy, device 'Tesla V100-SXM2-16GB' can simultaneously execute 8 blocks on each of the 80 SMs, so the kernel may need to execute a multiple of 640 blocks to hide the compute and memory latency. If the kernel is executing concurrently with other kernels then fewer blocks will be required because the kernel is sharing the SMs with those kernels." Below the text is an optimization suggestion: "Optimization: Increase the number of blocks executed by the kernel." and a "More..." link. The bottom of the interface shows two buttons: "Examine Occupancy" and "Show Kernel Profile - PC Sampling".

# Move forward (3): Analyse metrics

- Occupancy analysis

**1. CUDA Application Analysis**

**2. Performance-Critical Kernels**

**3. Compute, Bandwidth, or Latency Bound**

**4. Instruction and Memory Latency**

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

**Examine Occupancy**

Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

**Show Kernel Profile - PC Sampling**

The kernel profile shows the samples of various stall reasons collected at each step of the assembly instruction. Using this information you can pinpoint portions

**Results**

**i Occupancy Is Not Limiting Kernel Performance**

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. [More...](#)

Variable	Achieved	Theoretical	Device Limit	Grid Size: [ 4096,1,1 ] (4096 blocks)Block Size: [ 256,1
<b>Occupancy Per SM</b>				
Active Blocks		8	32	
Active Warps	53.8	64	64	
Active Threads		2048	2048	
Occupancy	84.1%	100%	100%	
<b>Warps</b>				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
<b>Registers</b>				
Registers/Thread		16	65536	

# nvprof

---

- Different choices [不同的选择]

- Short run:

- \$ nvprof output\_name.nvprof ./app\_name

- Specific metrics:

- \$ nvprof **-m** [metric list] -csv output\_name.csv ./app\_name

- Specific kernel(s):

- \$ nvprof **-kernels** :::1 output\_name.nvprof ./app\_name

- Analysis metrics:

- \$ nvprof **--analysis-metrics** output\_name.nvprof ./app\_name

# nvprof

---

- But in future ... 🤨

Note that **Visual Profiler** and **nvprof** will be deprecated in a future CUDA release. The NVIDIA **Volta** platform is the last architecture on which these tools are fully supported. [将不再完全支持]

It is recommended to use next-generation tools **NVIDIA Nsight Systems** for GPU and CPU sampling and tracing and **NVIDIA Nsight Compute** for GPU kernel profiling. [推荐使用]



# Nsight Systems

---



## NSIGHT SYSTEMS

### Overview



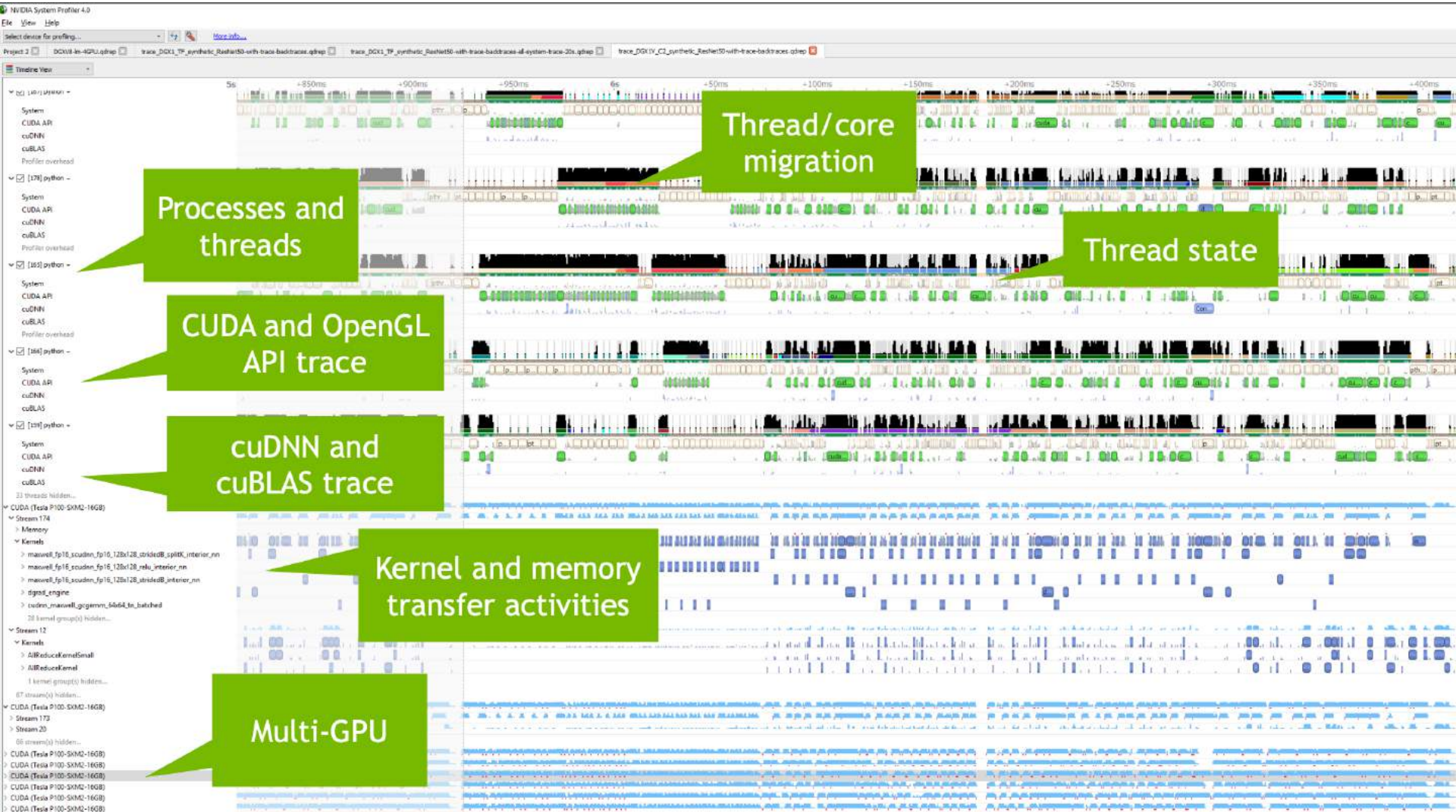
System-wide application algorithm tuning  
Multi-process tree support

Locate optimization opportunities  
Visualize millions of events on a very fast GUI timeline  
Or gaps of unused CPU and GPU time

Balance your workload across multiple CPUs and GPUs  
CPU algorithms, utilization, and thread state  
GPU streams, kernels, memory transfers, etc

OS: Linux x86\_64, Windows, MacOSX (host only)  
No plans for Linux Power

# Nsight Systems



# Nsight Compute



## NVIDIA NSIGHT COMPUTE

Next-Gen Kernel Profiling Tool

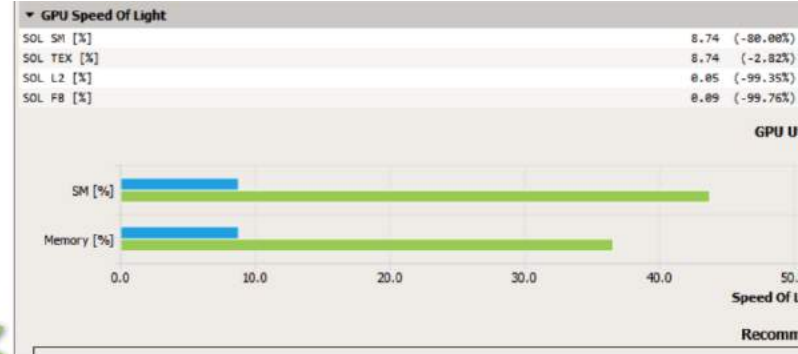


### Key Features:

- Interactive CUDA API debugging and kernel profiling
- Fast Data Collection
- Improved Workflow (Diff'ing Results)
- Fully Customizable (Programmable UI/Rules)
- Command Line, Standalone, IDE Integration

OS: Linux x86\_64, Windows, MacOSX (host only)  
Linux Power planned for Q2 2019

GPUs: Pascal, Volta, Turing



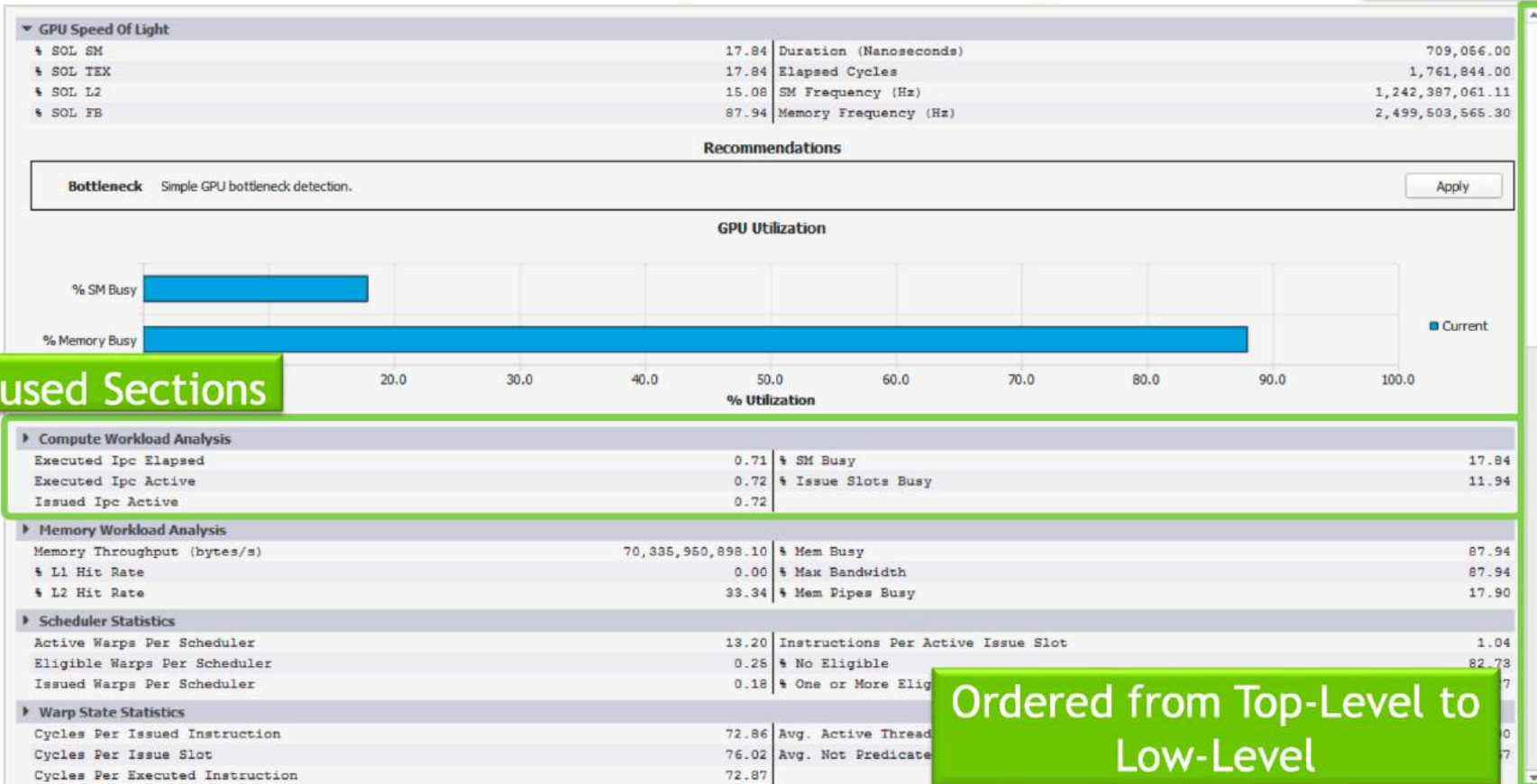
Metric	Value	Change
inst_executed [inst]	16,528,000	16,528,000
litex_sol_pct [%]	14.33	n/a
launch_block_size	128.00	128.00
launch_function_pcs	47,611,587,968.00	12,273,728.00
launch_grid_size	4,132.00	3,369.00
launch_occupancy_limit_blocks [block]	32.00	32.00
launch_occupancy_limit_registers [register]	21.00	21.00
launch_occupancy_limit_shared_mem [bytes]	384.00	384.00
launch_occupancy_limit_warps [warps]	16.00	16.00
launch_occupancy_per_block_size	3,638.00	3,638.00
launch_occupancy_per_register_count	5,792.00	5,792.00
launch_occupancy_per_shared_mem_size	2,260.00	2,260.00
launch_registers_per_thread [register/thread]	17.00	17.00
launch_shared_mem_config_size [bytes]	49,152.00	49,152.00
launch_shared_mem_per_block_dynamic [bytes/block]	0.00	0.00
launch_shared_mem_per_block_static [bytes/block]	20.00	20.00
launch_thread_count [thread]	528,896.00	431,232.00
launch_waves_per_multiprocessor	3.23	42.11
lit_sol_pct [%]	6.93	7.18
memory_access_size_type [bytes]	2.00; 32.00; 32.00; 32.00	2.00; 32.00; 32.00; 32.00

Source	Live Registers	Sampling Data (All)	Sampling Data (No Issue)
@IPT SHFL.IDX PT, RZ, RZ, RZ, RZ;	0	223	0
MOV R1, c[0x0][0x28];	1	13	44
S2R R0, SR_CTAID.X;	2	143	75
S2R R2, SR_TID.X;	3	0	38
IMAD R0, R0, c[0x0][0x0], R2;	3	599	94
ISETP.GE.AND P0, PT, R0, c[0x0][0x170]	2	125	26
@P0 EXIT;	2	259	86
MOV R2, R0;	3	386	29
@IPT SHFL.IDX PT, RZ, RZ, RZ, RZ;	2	0	0
MOV R4, 0x4;	3	0	0
IMAD.WIDE R4, R2, R4, c[0x0][0x160];	4	0	0
LDG.E.SYS R3, [R4];	3	0	0

# Nsight Compute

## NSIGHT COMPUTE Profile Report - Details Page

All Data on  
Single Page



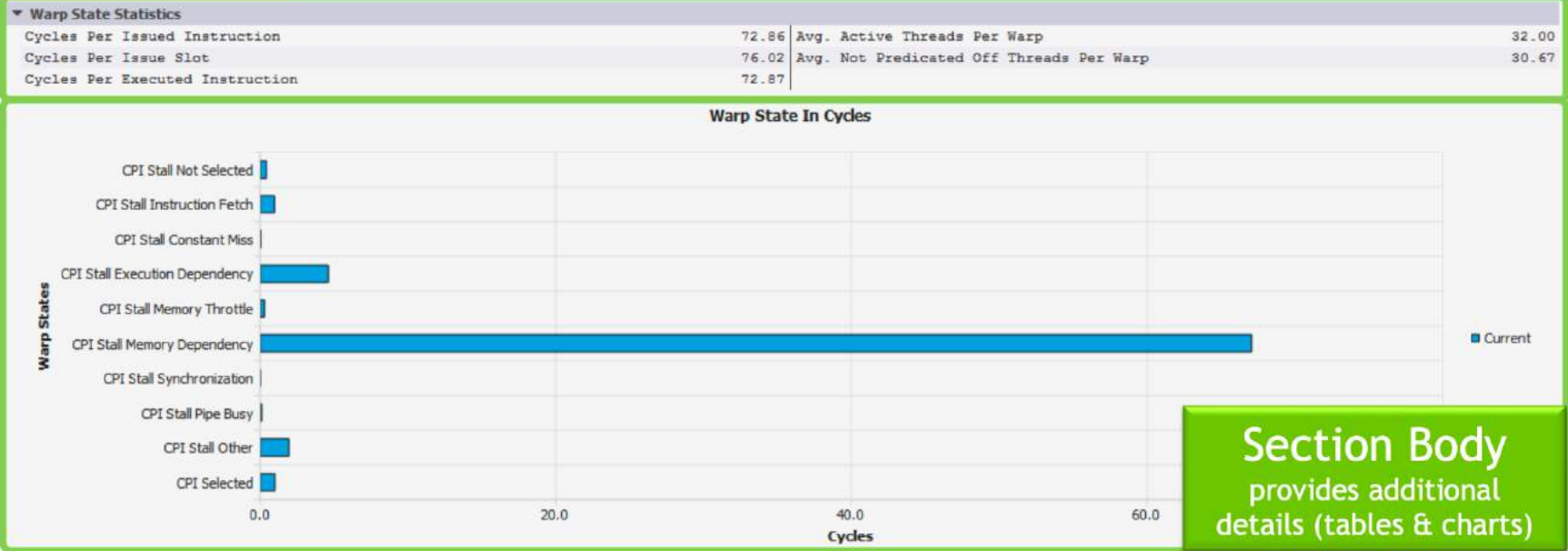
# Nsight Compute

## Section Header

provides overview & context for other sections

## NSIGHT COMPUTE

### Section Example



## Section Config

completely data driven  
add/modify/change sections

# Nsight Compute

## NSIGHT COMPUTE

### Unguided Analysis / Rules System

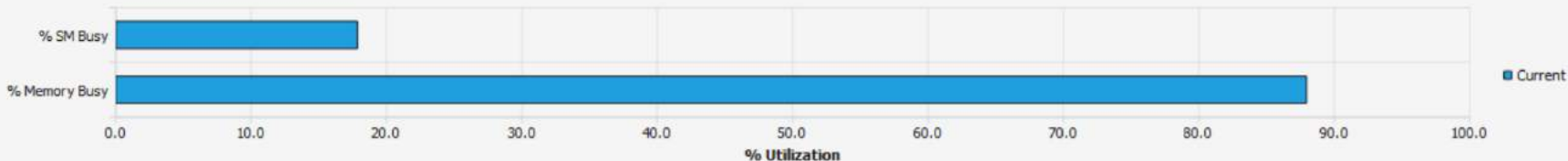
Analysis Rules  
recommendations from  
nvvp and more

17.84	Duration (Nanoseconds)	709,056.00
17.84	Elapsed Cycles	1,761,844.00
15.08	SM Frequency (Hz)	1,242,387,061.11
87.94	Memory Frequency (Hz)	2,499,503,565.30

#### Recommendations

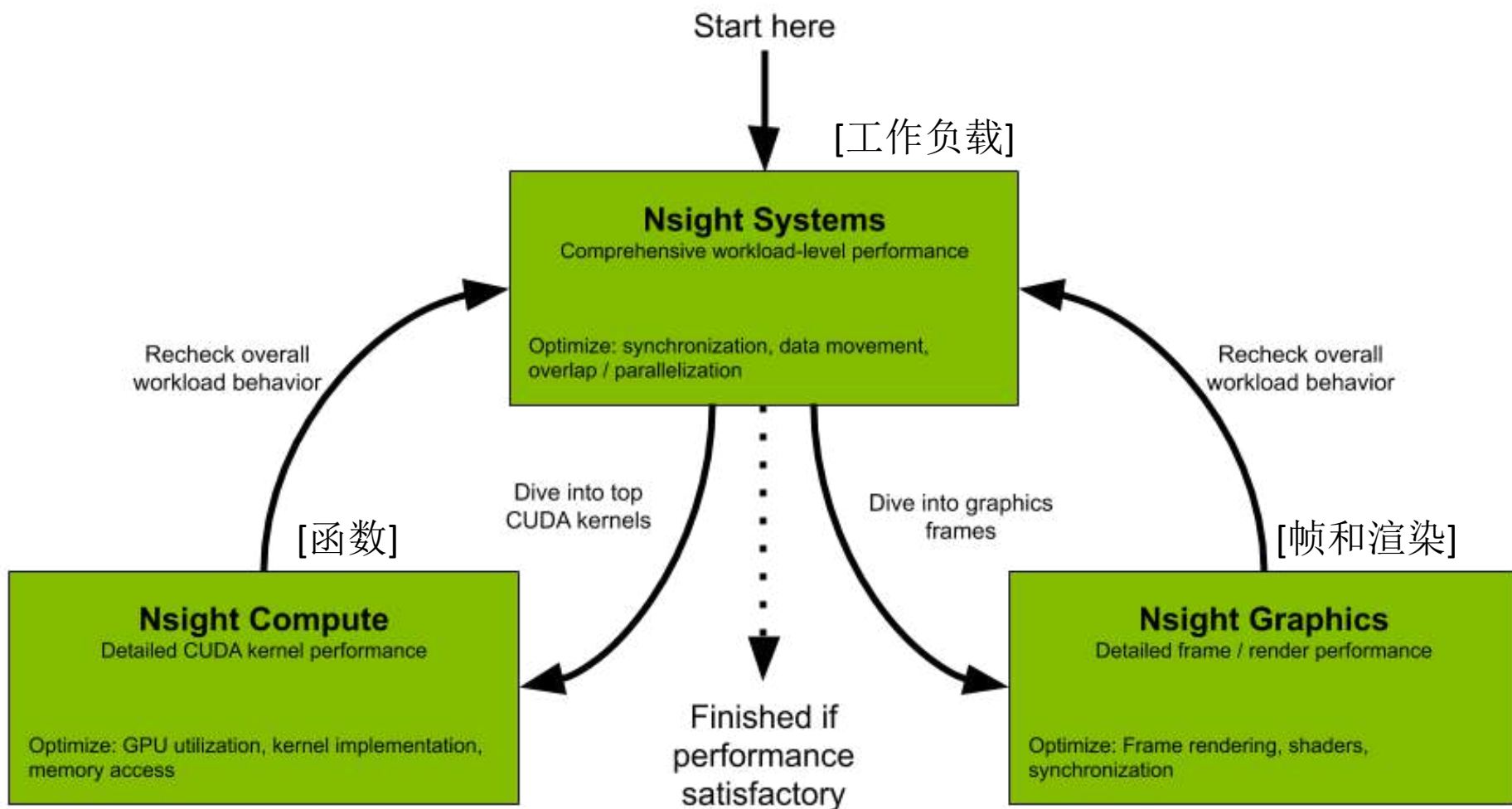
**⚠ Bottleneck** [Warning] Memory is more heavily utilized than Compute: Look at "Memory Workload Analysis" report section to see where the memory system bottleneck is. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can recompute.

#### GPU Utilization



Rules Config  
completely data driven  
add/modify/change rules

# NVIDIA Developer Tools Overview



# Conclusion

---

- nvprof is a hardware-based profile tool for the analysis and optimization of programs.
- You can customize the focus of profiling with different options, such as mode, metrics, kernel and so on.
- Visual profile makes your profiling result more intuitive.
- Nsight systems and Nsight compute will be a more sensible choice.



# References

---

- [1] Wikipedia. Profiling (computer programming)  
[https://en.wikipedia.org/wiki/Profiling\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming)).
- [2] Nvidia. 2020. CUDA Profiler. <https://docs.nvidia.com/cuda/profiler-users-guide/index.htm>
- [3] AMD. 2020. ROCm Profiler. <https://github.com/ROCm-Developer-Tools/rocprofiler/blob/master/doc/rocprof.md>
- [4] C.-K. Luk and R. Cohn , et al. 2005. " Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. " *In Conference on Programming Language Design and Implementation (PLDI)*. 190–200.
- [5] GNU. 2008. Debugging with GDB: The GNU Source-Level Debugger.  
<http://docs.adacore.com/live/wave/gdb-9/pdf/gdb/gdb.pdf>
- [6] O. Villa and M. Stephenson, et al. 2019. " NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. " *In IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–383.
- [7] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O’ Connor, and S. W. Keckler, " Flexible software pro- filing of gpu architectures, " *in ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.
- [8] D. Shen, S. L. Song, A. Li, and X. Liu, " Cudaadvisor: Llvm-based runtime profiling for modern gpus, " *in Proceedings of the 2018 Inter- national Symposium on Code Generation and Optimization*, 2018, pp. 214–227.
- [9] L. Braun and H. Froning, " Cuda flux: A lightweight instruction profiler for cuda applications, " *in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) Workshop, collocated with International Conference for High Performance Computing, Networking, Storage and Analysis (SC2019)*, 2019.

Thanks for your attention !