GPU Programming

CUDA Nitty Details

Christian Lessig

Atomics

- Analogous to atomics in C++
- For global and shared memory
- Specific versions for entire GPU and thread block
- atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc(), atomicAnd(), ...

© Christian Lessig, 2017

Data-types

- Default precision is IEEE 32-bit (float and int)
  - Device is designed for it

© Christian Lessig, 2017

Data-types

- Default precision is IEEE 32-bit (float and int)
  - Device is designed for it
- Many applications require 64-bit precision
  - e.g. most simulations in scientific computing

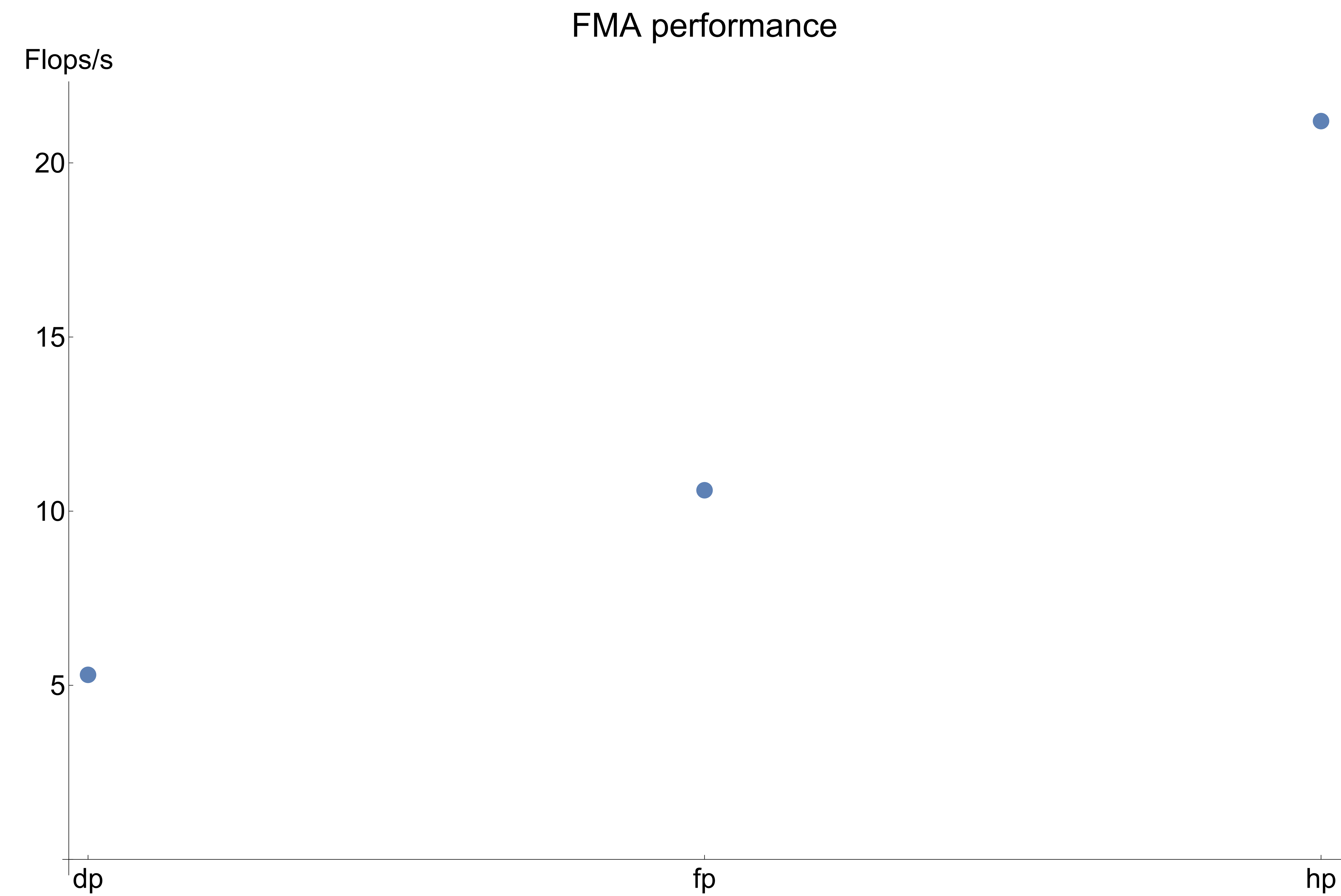
© Christian Lessig, 2017

Data-types

- Default precision is IEEE 32-bit (float and int)
  - Device is designed for it
- Many applications require 64-bit precision
  - e.g. most simulations in scientific computing
- Some applications require less precision
  - e.g. graphics, deep learning

© Christian Lessig, 2017

Data-types



Data from https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/

© Christian Lessig, 2017

Data-types

```
int n2 = n/2;
half2 *x2 = (half2*)x, *y2 = (half2*)y;

for (int i = start; i < n2; i+= stride)
  y2[i] = __hfma2(__halves2half2(a, a), x2[i], y2[i]);
```

© Christian Lessig, 2017

Warp shuffle functions

- Communicate values in a warp without using shared memory

© Christian Lessig, 2017

Warp shuffle functions

- Communicate values in a warp without using shared memory
- E.g.:

```
__shfl_sync( unsigned mask, T var, int srcLane,
             int width=warpSize);
```

© Christian Lessig, 2017

Warp shuffle functions

- Communicate values in a warp without using shared memory
- E.g.:

```
__shfl_sync( unsigned mask, T var, int srcLane,
             int width=warpSize);

value = __shfl_sync(0xffffffff, value, 0);
```

© Christian Lessig, 2017

Pinned host memory

- Data transfer from host to device copies from host RAM to device RAM.

Pinned host memory

- Data transfer from host to device copies from host RAM to device RAM.
  - malloc / new allocated memory is pageable, i.e. not necessarily in RAM

© Christian Lessig, 2017

Pinned host memory

- Data transfer from host to device copies from host RAM to device RAM.
  - malloc / new allocated memory is pageable, i.e. not necessarily in RAM
  - Allocate **pinned** (non-pageable) memory:

© Christian Lessig, 2017

Pinned host memory

- Data transfer from host to device copies from host RAM to device RAM.
  - malloc / new allocated memory is pageable, i.e. not necessarily in RAM
  - Allocate **pinned** (non-pageable) memory:

```
// float* data = (float*) malloc(8*sizeof(float));
cudaMallocHost((void**)& data_pinned, 8*sizeof(float));
```

© Christian Lessig, 2017

Driver API

- Driver API for more fine grained control

© Christian Lessig, 2017

Driver API

- Driver API for more fine grained control

```
// cuMemcpyHtoD(d_B, h_B, size);
cudaMemcpy( d_B, d_h, size, cudaMemcpyHostToDevice)

// vecAdd<< blocks, threads>>( d_A, d_B, d_C)
void* args[] = { &d_A, &d_B, &d_C, &N };
cuLaunchKernel( vecAdd,  blocksPerGrid, 1, 1,
                threadsPerBlock, 1, 1, 0,
                args, NULL);
```

© Christian Lessig, 2017

NVRTC

- On-the-fly (runtime) compilation of code

```
compileFileToPTX("vecAdd.cu", 0, NULL, &ptx, &ptxSize);
CUmodule module = loadPTX(ptx, argc, argv);

CUfunction kernel_addr;
cuModuleGetFunction(&kernel_addr, module, "vecAdd"));
```

© Christian Lessig, 2017

Asynchronous Execution

- Motivation:
  - Processing large data sets
  - Smaller tasks that do not utilize full device
  - Latency critical applications
- Executes multiple kernels (streams) concurrently
  - Typically combined with asynchronous data transfer

© Christian Lessig, 2017

Asynchronous Execution

```
for (int i = 0; i < nStreams; ++i) {
  cudaStreamCreate( &stream[i]);
  cudaMemcpyAsync( &dd[i], &dh[i], i,
                    cudaMemcpyHostToDevice,
                   stream[i]);
  kernel<<<block, threads, 0, stream[i]>>>(dd);
  cudaMemcpyAsync( &dh[i], &dd[i], size,
                   cudaMemcpyDeviceToHost, stream[i]);
}
```

© Christian Lessig, 2017

Asynchronous Execution

```
for (int i = 0; i < nStreams; ++i) {
  cudaStreamCreate( &stream[i]);
  cudaMemcpyAsync( &dd[i], &dh[i], i,
                   cudaMemcpyHostToDevice,
                   stream[i]);
  kernel<<<block, threads, 0, stream[i]>>>(dd);
  cudaMemcpyAsync( &dh[i], &dd[i], size,
                   cudaMemcpyDeviceToHost, stream[i]);
}
```

© Christian Lessig, 2017

Unified memory

- Unified address space for data
- Memory management (host <-> device transfers) are handled by the API / driver

© Christian Lessig, 2017

Memory allocation on the device

- Memory can be allocated dynamically on the device from a pre-defined “heap“ area (that, however, resides in in global memory)

```
__global__ void kernel( int size) {
     char* ptr = (char*) malloc(size);
     memset(ptr, 0, size);
     free(ptr);
}
```

© Christian Lessig, 2017

Dynamic parallelism

- Enables to generate new thread grid from within an existing one
- Avoids significant overhead of host <-> device synchronization when parallelism is data dependent

Warp matrix operations

- Core of the deep learning hardware on CUDA devices.
- Provides dedicated hardware to implement:

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,\dots} & A_{0,15} \\ A_{1,0} & A_{1,1} & A_{1,\dots} & A_{1,15} \\ A_{\dots,0} & A_{\dots,1} & A_{\dots,\dots} & A_{\dots,15} \\ A_{15,0} & A_{15,1} & A_{15,\dots} & A_{15,15} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,\dots} & B_{0,15} \\ B_{1,0} & B_{1,1} & B_{1,\dots} & B_{1,15} \\ B_{\dots,0} & B_{\dots,1} & B_{\dots,\dots} & B_{\dots,15} \\ B_{15,0} & B_{15,1} & B_{15,\dots} & B_{15,15} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,\dots} & C_{0,15} \\ C_{1,0} & C_{1,1} & C_{1,\dots} & C_{1,15} \\ C_{\dots,0} & C_{\dots,1} & C_{\dots,\dots} & C_{\dots,15} \\ C_{15,0} & C_{15,1} & C_{15,\dots} & C_{15,15} \end{pmatrix}$$

FP16 or FP32 | FP16 | FP16 | FP16 or FP32

from https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

© Christian Lessig, 2017

Warp matrix operations

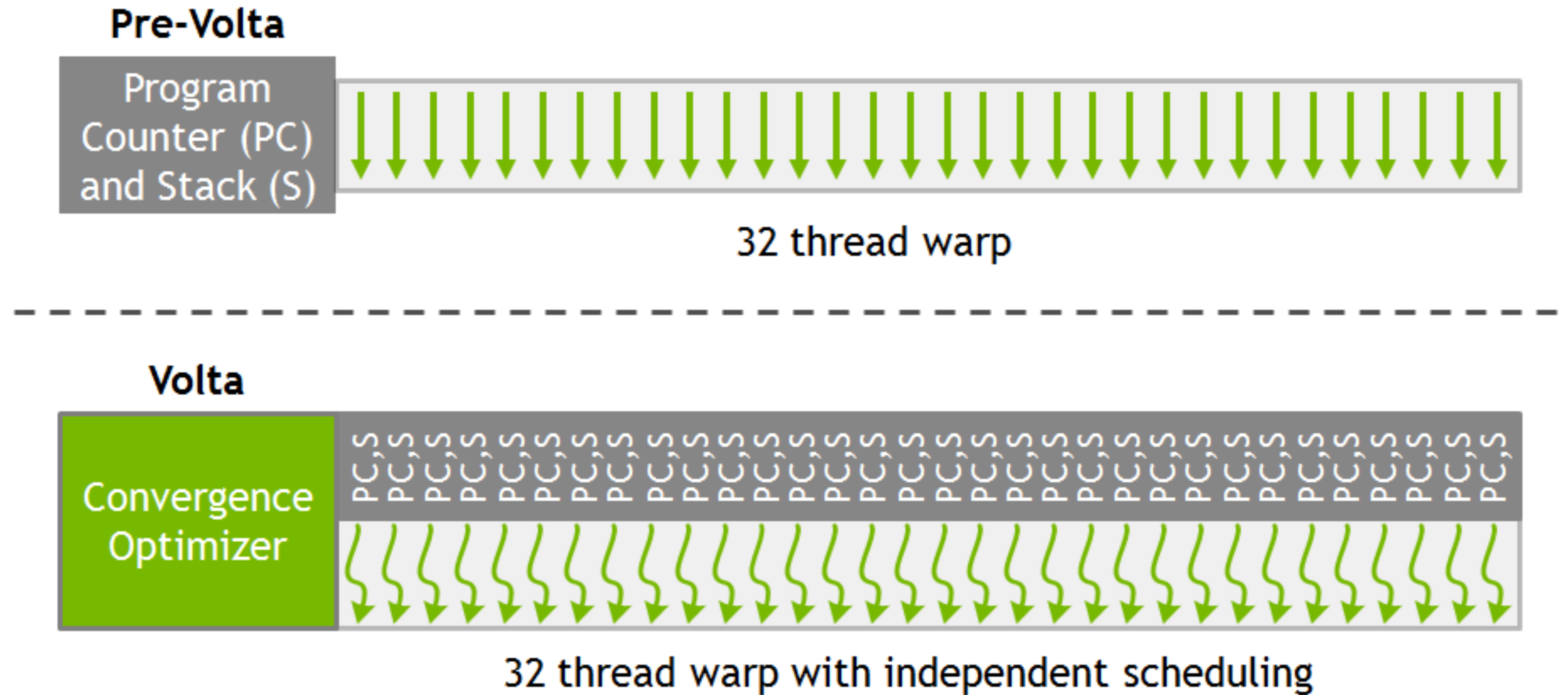
- Core of the deep learning hardware on CUDA devices.

```
// Load the inputs
wmma::load_matrix_sync(a_frag, a, lda);
wmma::load_matrix_sync(b_frag, b, ldb);

// Perform the matrix multiplication
wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
```

© Christian Lessig, 2017

Independent thread scheduling



from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

© Christian Lessig, 2017

Graphics inter-op

- Enables to share data with OpenGL / DirectX
- Useful for example for global illumination or simulation of fluids that are more efficiently computed in Cuda but must be displayed using a graphics API.

© Christian Lessig, 2017

Cooperative groups

- Extends the concept of a warp / thread block to user specified groups of threads that can interact
  - Synchronization of sets of blocks or entire grid (with-out host intervention)
  - Communication of values within a group of threads

© Christian Lessig, 2017

Cuda libraries

- cuFFT
- cuSparse
- cuBLAS
- cuDNN
- CUTLASS
- thrust
- ...

© Christian Lessig, 2017

nvprof / nvvp

- nvprof: command line profiler

© Christian Lessig, 2017

nvprof / nvvp

- nvprof: command line profiler

```
bauhaus:build lessig$ /Developer/NVIDIA/CUDA-8.0/bin/nvprof ./main
==1714== NVPROF is profiling process 1714, command: ./main
Execution time: 381.35 ms.
==1714== Profiling application: ./main
==1714== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.93%  380.73ms        99  3.8458ms  3.7730ms  4.4032ms  transposeMatrix3(float*,
float*, unsigned int)
  6.00%  25.692ms         1  25.692ms  25.692ms  25.692ms  [CUDA memcpy HtoD]
  5.07%  21.701ms         1  21.701ms  21.701ms  21.701ms  [CUDA memcpy DtoH]

==1714== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 66.50%  380.68ms         2  190.34ms  18.900us  380.67ms  cudaDeviceSynchronize
 24.95%  142.84ms         2  71.419ms  194.69us  142.64ms  cudaMalloc
 ...
```

© Christian Lessig, 2017

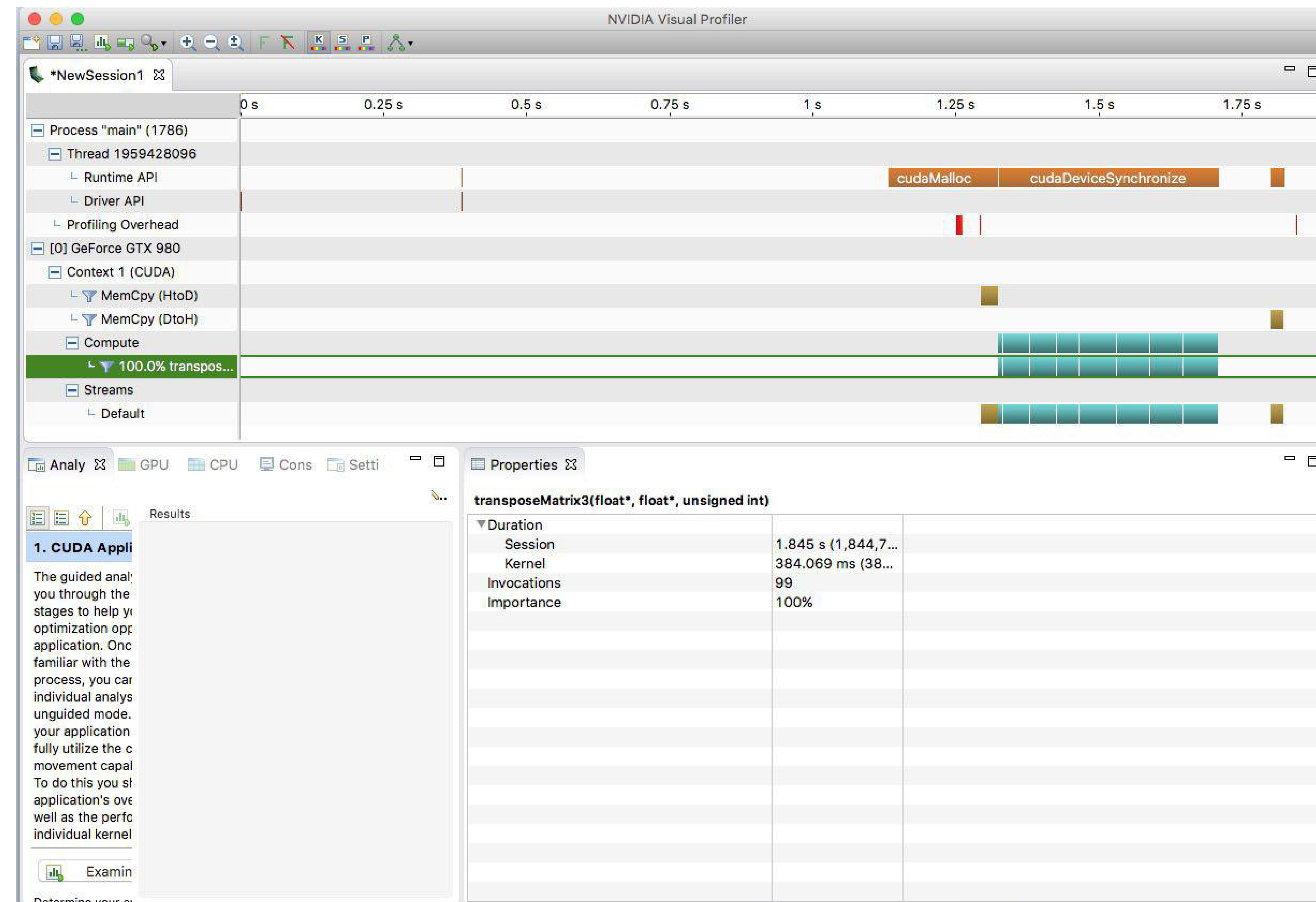
nvprof / nvvp

- nvvp: visual profiler

© Christian Lessig, 2017

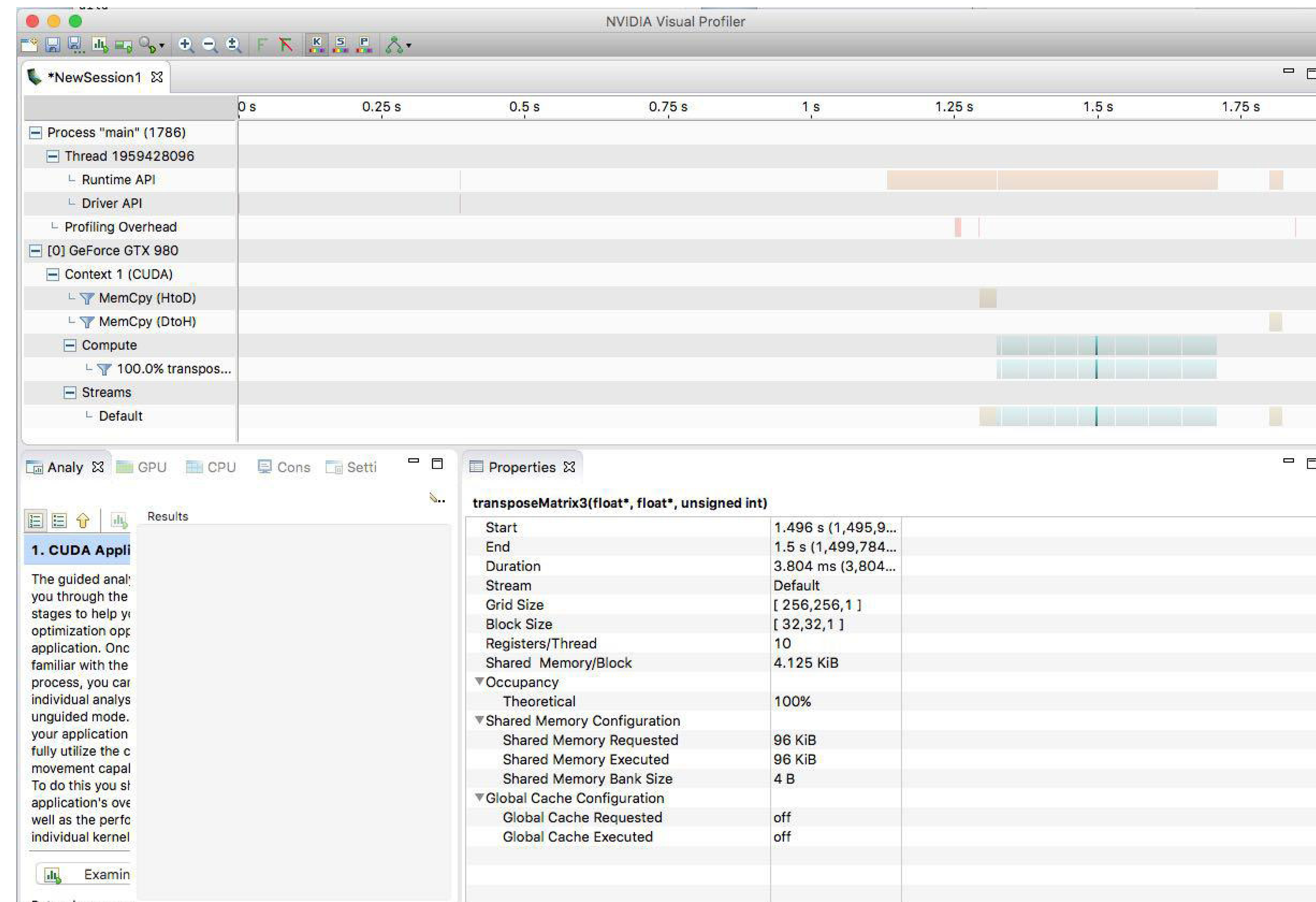
nvprof / nvvp

- nvvp: visual profiler



nvprof / nvvp

- nvvp: visual profiler



© Christian Lessig, 2017