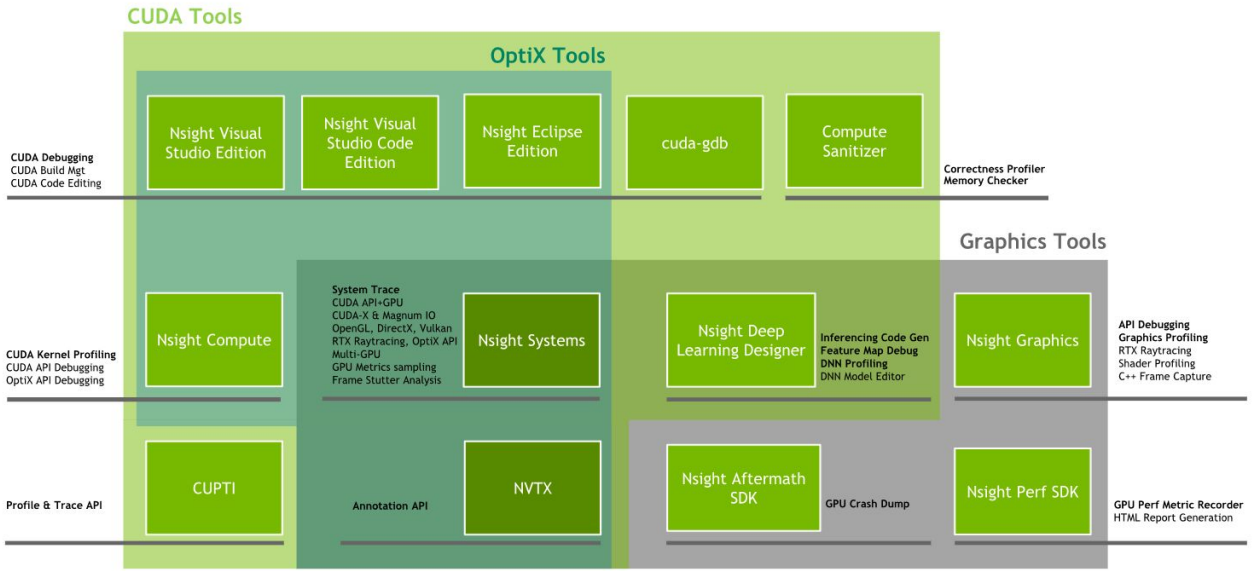
CUDA Toolkit

CUDA profilers and compilers

NSIGHT DEVELOPER TOOLS



Reference here

Sidenotes

- Use the programming guide as much as possible
- For CUDA API, use the Runtime API Documentation
- Nvidia's documentation is very good and well-supported
  - If you have an issue, someone else probably has already solved it: Google it!
- A full CUDA programming guide can be found here

KEEP IN MIND: GPUs are not multi-core processors!

Runtime vs Driver API

- CUDA API exposes the **Runtime** and the **Device** API
- We will refer to the CUDA Runtime API
  - The **Runtime** API eases device code management by providing implicit initialization, context management, and module management. This leads to simpler code, but it also lacks the level of control that the driver API has
  - The **Driver** API offers more fine-grained control, especially over contexts and module loading. Thus, kernel launches are much more complex to implement

Best Practices - 1

- Use the `__restrict__` keyword
- If something is constant use `const`
- **Never** use global variables if you can
  - They mess up with multiple compile units
- If something can be done at **compile time**, do it
  - Macros and `constexpr` are your friends
- Use `CHECK` functions (like in the examples) to verify execution status
  - CUDA delivers performance, not safety
  - Differences between sync vs async errors
  - **TIP:** You can enable checks only in debug mode

Best Practices - 2

- Use `inline` carefully
  - Compilers use it as a suggestion: it is not a contract
  - If needed refers to `__attribute__((always_inline))`
- Grid's dimensions can be estimated at runtime
  - use API `cudaOccupancyMaxActiveBlocksPerMultiprocessor`
  - We will see an example later on
- Pay attention to `#pragma unroll`
  - It can increase performance
  - But it also increases the register pressure
- Pay attention to synchronizations functions
  - `__syncthreads` or `__syncwarp`
  - Fences can also be an options

List of Optimizations

- Recap of possible optimizations we will apply in the exercise lesson

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

Lesson Outline

- Compiling
- Debugging
- Binary Utilities
- Profiling & Metrics
- Roofline

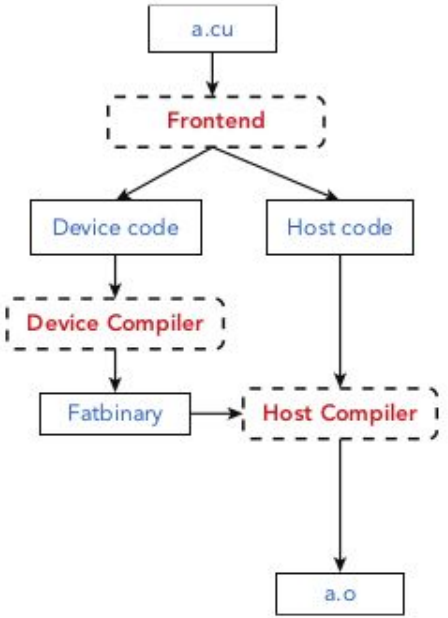
Compiling

NVCC

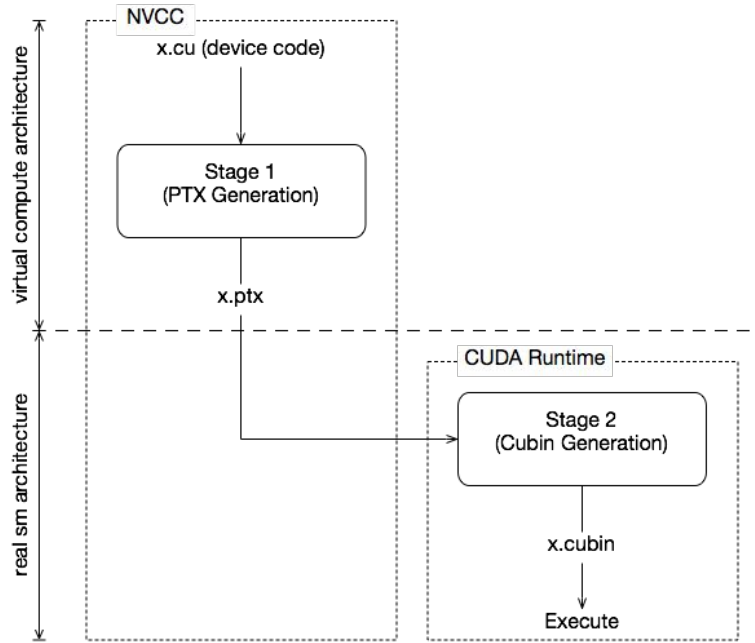
- The NVidia C Compiler
  - involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file
- NVCC supports most of the C++11 constructs onward
  - You can find a detailed list here
- Reference at this link
- More on that here

Compilation Output

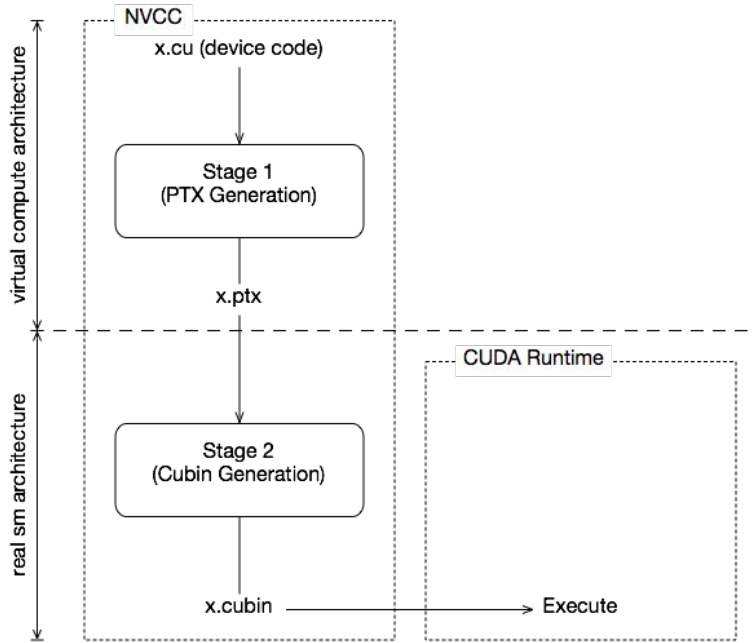
- CUDA compilation produces different files at different stages
  - CUBIN file contains CUDA executable binary code to target a real architecture
  - PTX (Parallel Thread eXecution) is an ISA, and PTX files are used to target virtual architecture
  - FATBIN contains both of them, and they embed PTX and CUBIN in the executable
- PTX differs from SASS
  - SASS is a low-level assembly language that compiles to binary microcode and executes natively on NVIDIA GPU hardware.



Compilation Phases



Ahead Of Time



Just In Time

Examples

- Compile for a virtual architecture with compute capabilities 5.0, Stage 2 JIT at runtime

  ```
  $ nvcc <your_app> -arch=compute_50
  ```

- Compile for a specific architecture sm_50, Stage 1 AOT at compile-time

  ```
  $ nvcc <your_app> -arch=sm_50
  ```

- You can also combine the two to have support for both virtual arch and real ones

  ```
  $ nvcc <your_app> -arch=compute_50,sm_50,sm_52
  ```

- If you use the -v options you can see how the compilation trajectory change
- **KEEP IN MIND**: if you use -Xptxas -v you can also get some stats about specific kernels compilation

Other SDK & Compilers

- NVIDIA HPC SDK
  - NVIDIA software developer kit for high-performance environment
- NVVM
  - CUDA C compiler front-end (NVCC) can generate NVVM IR.
  - The NVVM compiler (which is based on LLVM) generates PTX code from NVVM IR
- CUCLANG
  - You can also compiler CUDA code using the CLANG compiler
  - Part of the LLVM project

Debugging

CUDA-GDB

- Is the CUDA version of standard GDB
- Available features are
  - Breakpoints
  - Memory Inspection
  - Multiple GPUs and Contexts support
  - Error detection
- You can enable debug mode upon compiling
  - You need to use -g for debug host code and -G for debug device code
  - `$ nvcc -g -G <your_app>`
- Extensive reference from NVIDIA and presentation

Usage

- Simple as executing `cuda-gdb <your_application>`
- You can shift focus using the Thread Focus
  - You can switch between threads to get local variable value

    ```
    (cuda-gdb) cuda kernel 2 block 1,0,0 thread 3,0,0
    ```
  - Or you can get your current focus

    ```
    (cuda-gdb) cuda kernel block thread
    ```
- You can also print variables or registers

  ```
  (cuda-gdb) print my_variable
  (cuda-gdb) info registers R0 R1 R4
  ```

Binary Utilities

CUDA Binary Utilities

- A CUDA binary (also referred to as cubin) file is a file that consists of CUDA executable code sections as well as other sections
- It is an ELF-formatted file, a common standard file format for executable files
- NVCC embeds cubin files into the host executable file.
- Reference here

CUOBJDUMP & NVDISASM

- CUOBJDUMP extracts information from CUDA binary files (both standalone and those embedded in host binaries) and presents them in human-readable format
- NVIDIASM extracts information from standalone cubin files and presents them in human-readable format.

Table 1. Comparison of `cuobjdump` and `nvdisasm`

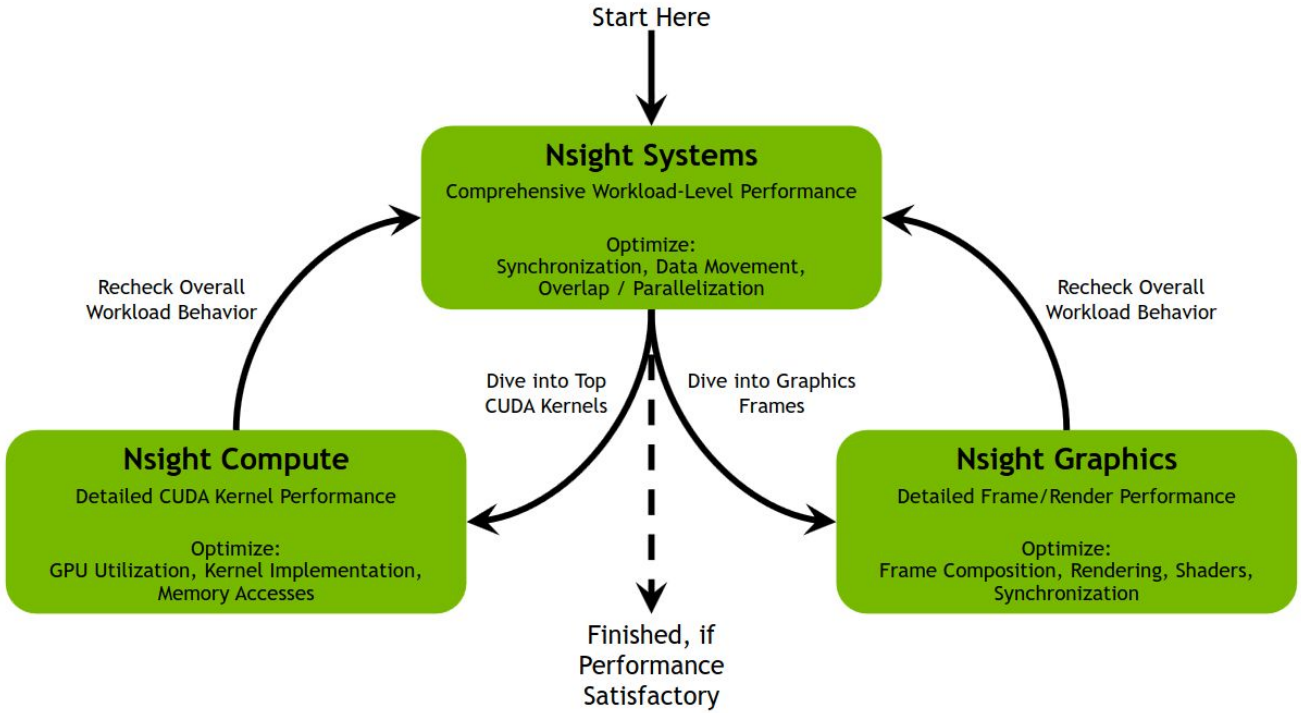
	`cuobjdump`	`nvdisasm`
Disassemble cubin	Yes	Yes
Extract ptx and extract and disassemble cubin from the following input files: › Host binaries › Executables › Object files › Static libraries › External fatbinary files	Yes	No
Control flow analysis and output	No	Yes
Advanced display options	No	Yes

Profiling & Metrics

The old way: Visual Profiler and NVPROF

- The NVPROF profiling tool enables you to collect and view profiling data from the command line
- The NVIDIA Visual Profiler (NVVP) allows you to visualize and optimize the performance of your application
- The NVIDIA Volta platform is the last architecture on which these tools are fully supported
- This post from NVIDIA talks about the transition to the new profiling system

NSIGHT Profiling



NSIGHT SYSTEMS

- System-wide profiler applications
- You should start profiling from this tool
- You can use both the CLI and GUI version
- Already included in CUDA toolkit
- All reference here

NSIGHT COMPUTE

- It is an interactive kernel profiler for CUDA applications
- You can use both the CLI and GUI version
- Already included in CUDA toolkit
- All reference here

CUPTI - Metrics

- The CUDA Profiling Tools Interface (CUPTI) enables the creation of profiling and tracing tools that target CUDA applications
- NVIDIA NSIGHT tools are based on CUPTI API to collect data and metrics
- All the references here

NVTX

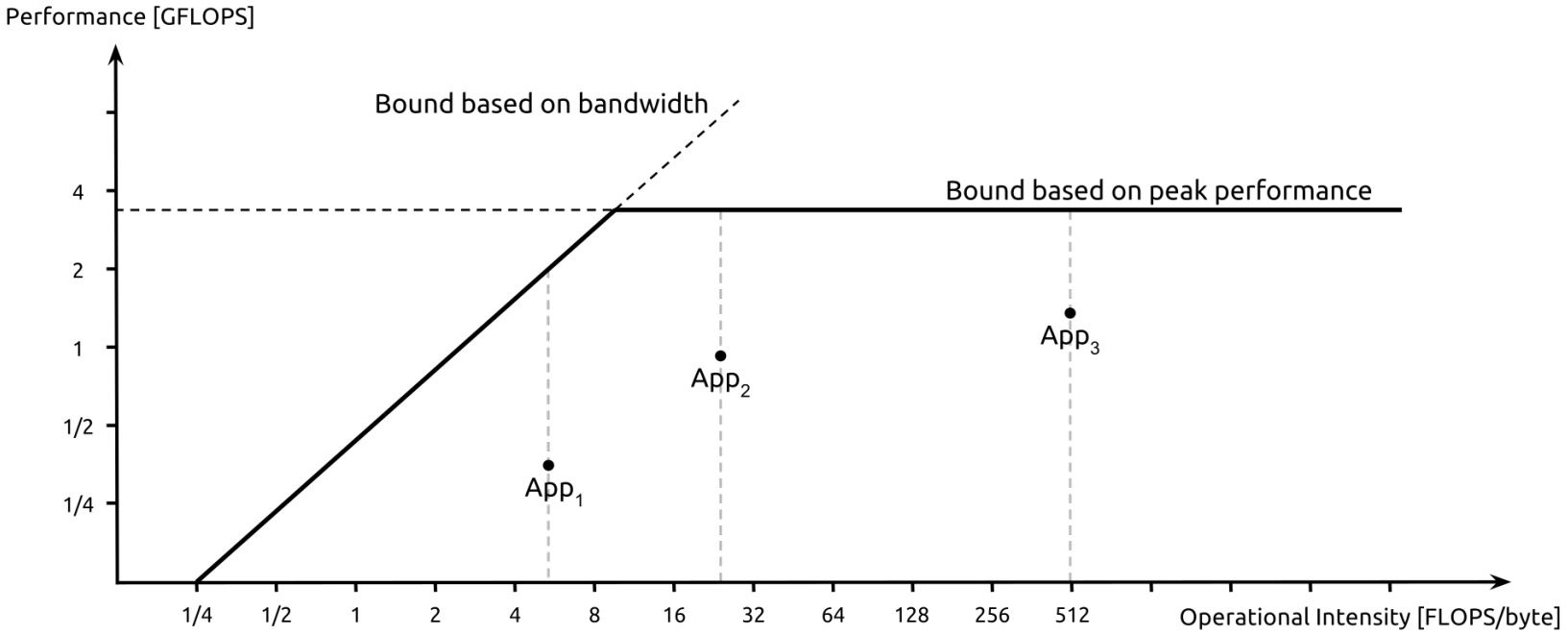
- The NVIDIA Tools Extension (NVTX) library is a set of functions that a developer can use to provide additional information to tools
- It can help when profiling using CUPTI, since it allows to define ranges of time upon execution to focus attention
- All the references here

Roofline

The Roofline Performance Model - 1

- The roofline model is a visual model used to provide performance estimates of a given compute kernel or application on a specific hardware
- A device can be characterized by
  - Computational peak performance which is the maximum number of operations per second (i.e., floating point FLOPS/s)
  - Bandwidth peak performance is measured in bytes that can be fetched and stored (bytes/s)
- Kernels or applications are evaluated in terms of Arithmetic Intensity (AI)
  - It is defined as defined as the number of floating-point operations per unit of data FLOPS/bytes
- Great talk from NERSC at GTC 2020

The Roofline Performance Model - 2



The Roofline Performance Model - Exercise

- Consider this simple kernel about matrix multiplication
- It does 3 access to global memory access(single precision)
  - 2 read and 1 write, 32 bit, thus 8 bytes in total
- It does 2 operations
  - 2 FLOP / 8 bytes = 0.25 FLOP/B
- Considering an A100 with HBM's bandwidth of *1555 GB/second*, you end up with *389 GFLOPS*
  - *1555 GB/second * 0.25 FLOP/B = 389 GFLOPS*

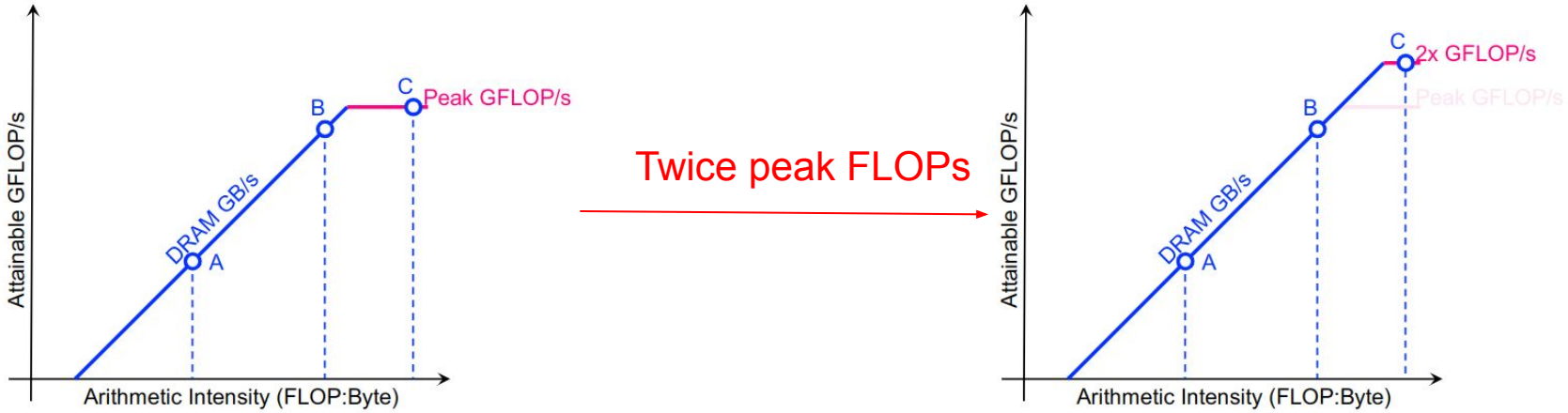
```
for (int k = 0; k < Width; ++k) {
    Pvalue += M[row*Width+k] * N[k*Width+col];
}
```

The Roofline Performance Model - Exercise

- The single precision computational power of an A100 is 19,5 GFLOPS
  - Only 2% of the peak computational power
- Thus, the kernel is memory-bound
- Instead IF the single precision computational power would have been 200 GFLOPS, the kernel would have been compute bound

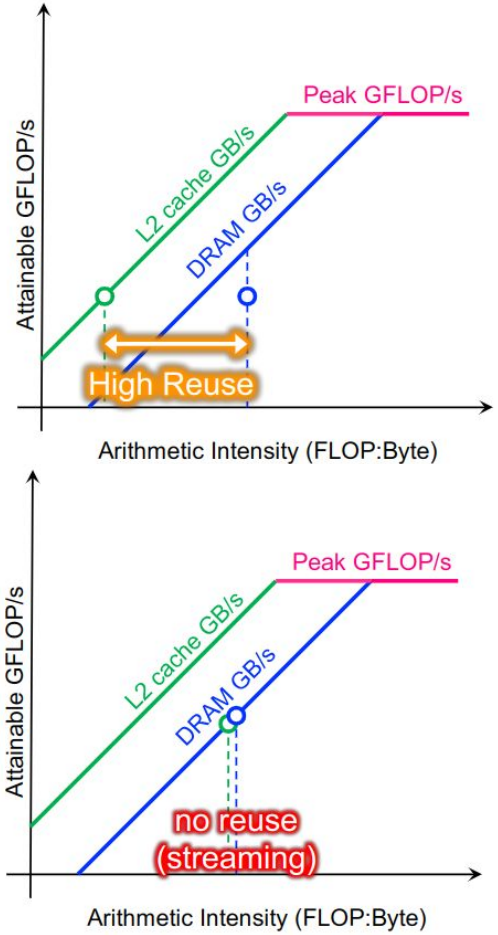
```
for (int k = 0; k < Width; ++k) {
    Pvalue += M[row*Width+k] * N[k*Width+col];
}
```

The Roofline Performance Model - 3

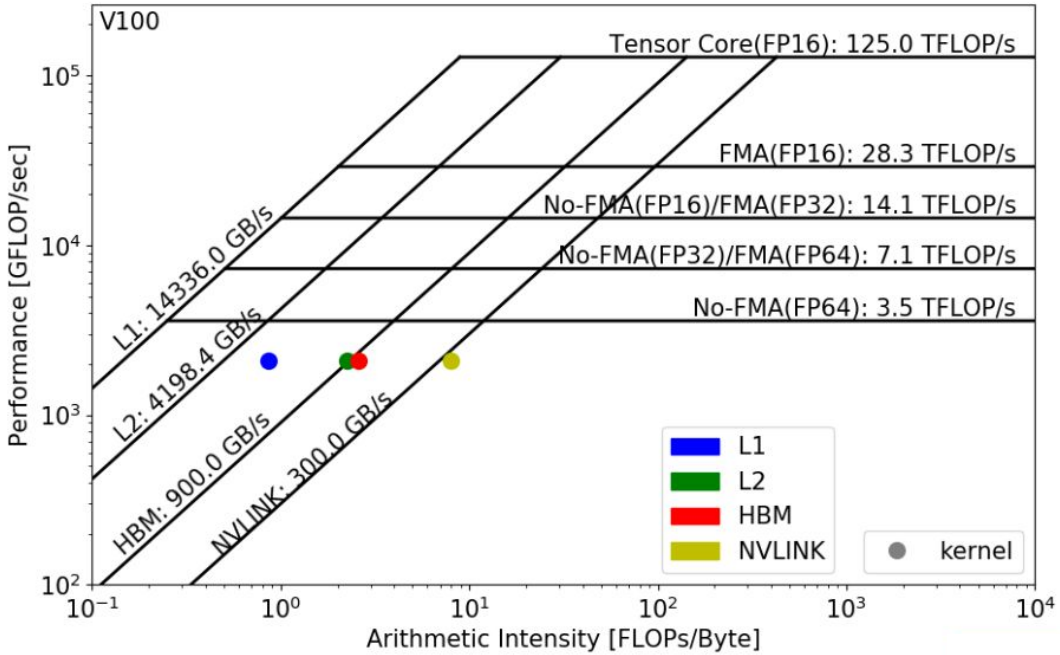


Hierarchical Rooflines - 1

- Construct superposition of Rooflines, measure AI and bandwidth for each level of memory/cache
  - Widely separated Arithmetic Intensities indicate high reuse in the cache (upper image)
  - Similar Arithmetic Intensities indicate effectively no cache reuse (lower image)
- **But** performance is ultimately the minimum of these bounds

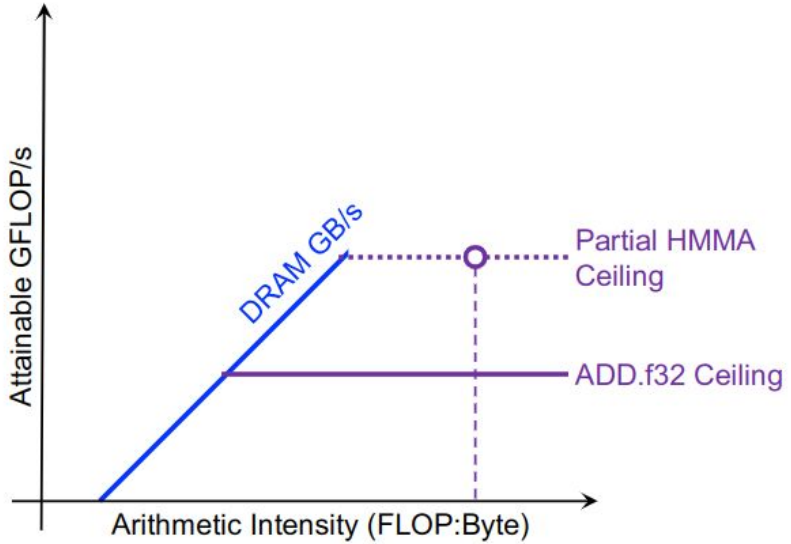


Hierarchical Rooflines - 2



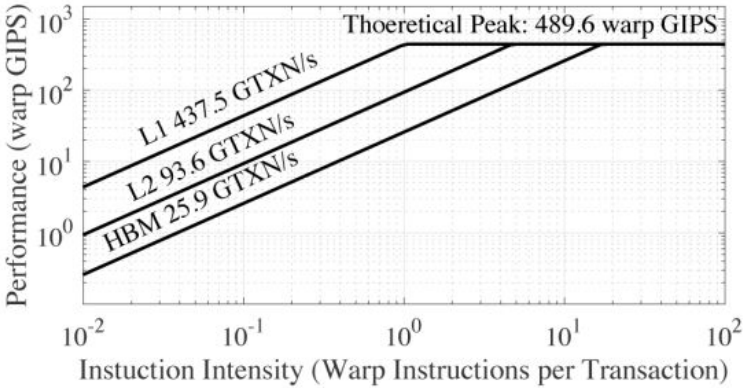
Rooflines and Instructions

- Modern CPUs and GPUs are increasingly reliant on special (fused) instructions that perform multiple operations (like FMA)
- However, kernels/apps will mix HMMA with FMA, MULs, ADDs, …



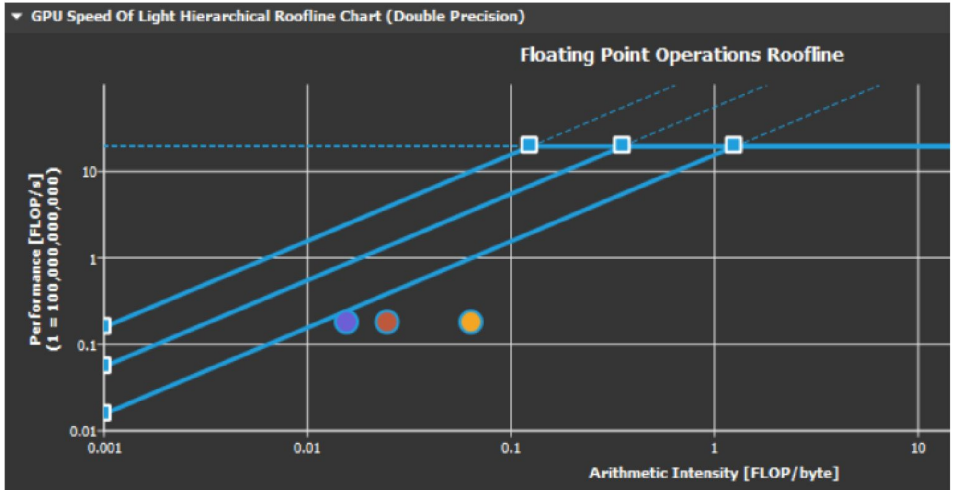
Instructions and Transactions Rooflines

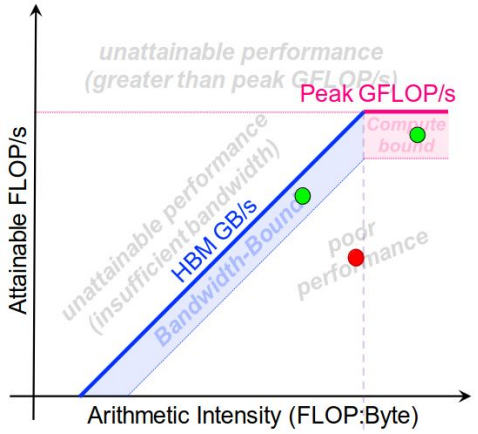
- The classic FLOP-centric approach is inappropriate for emerging applications that perform more integer operations than floating-point operations
- The **Instruction Roofline** incorporates instructions and memory transactions across all memory hierarchies together
- Methodology presented in this IEEE's article



NVIDIA Roofline

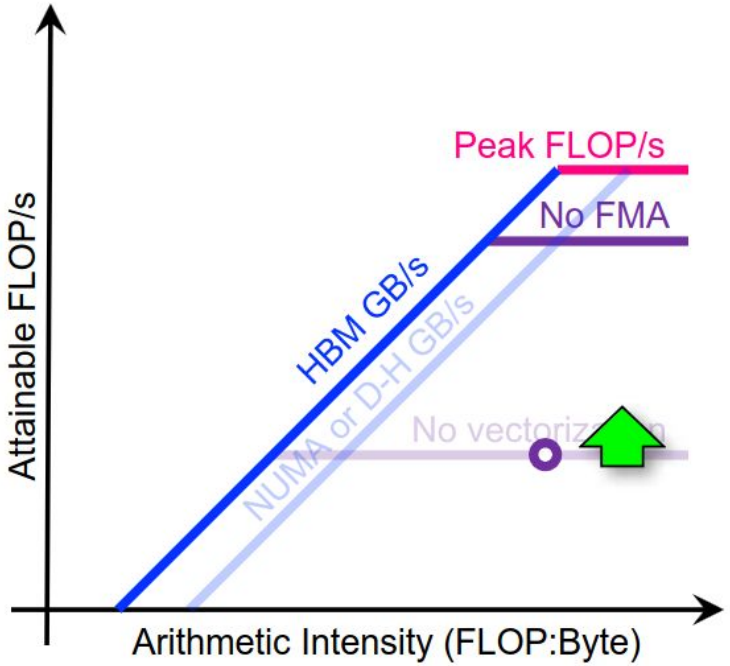
- You can use Nsight Compute to get a roofline of your kernel
- You can find a great lecture at GTC 2021
- You have to change NCU sections to do it, as in this repo





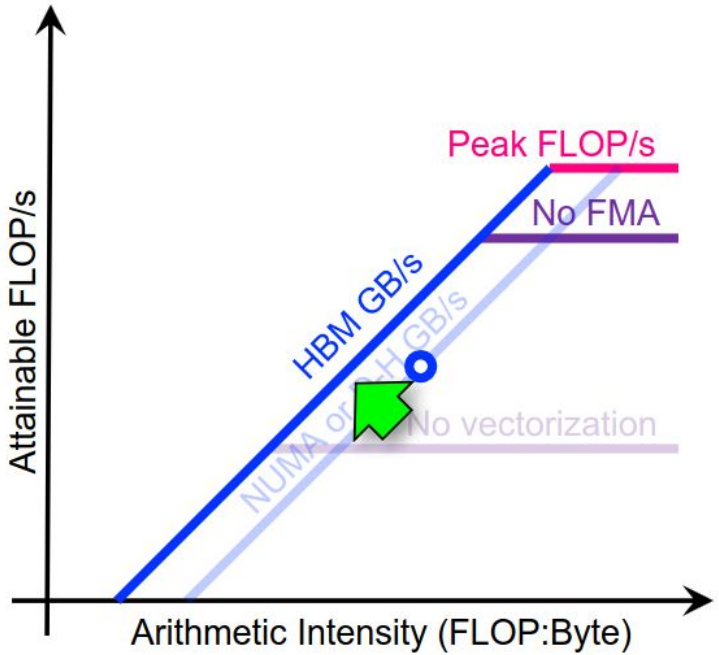
Roofline Guided Optimizations - 1

- Maximize compute performance
  - Multithreading
  - Vectorization
  - Increase SM occupancy
  - Utilize FMA instructions
  - Minimize thread divergence



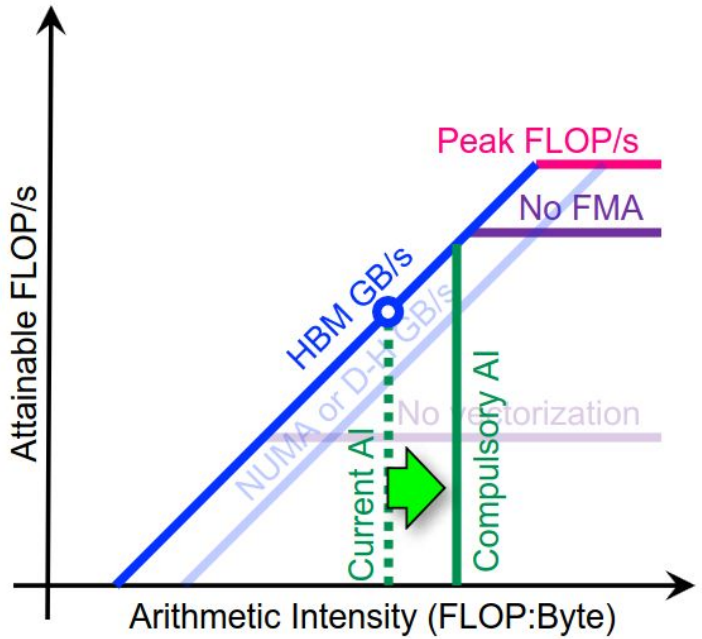
Roofline Guided Optimizations - 2

- Maximize memory bandwidth
  - Utilize higher-level caches
  - Avoid H-D transfers
  - Avoid uncoalesced memory access



Roofline Guided Optimizations - 3

- Improve AI
  - Minimize data movement
  - Exploit cache reuse



Rooflines and Profiling - Exercise

- Now we put all together with an example
  - We profile a matrix multiplication, similar to the one explained in the first lessons
- The objective is
  - Use the tools we have previously saw
  - See the impact of different optimizations on performance and rooflines
- More detailed examples can be found
  - Roofline model more in detail from this repo
  - For code and further explanation refers to this very good article about cuBLAS like matrixMul performance
    - The code is available at the following link

Rooflines and Profiling - cuBLAS

- Matrix multiplication comparison
  - Refers to the following article an repo
  - cuBLAS
    - NVIDIA version of BLAS
      - BLAS functionality is categorized into three sets of routines called "levels"
      - We use the level 3 containing matrix-matrix operations, which performs $O(N^3)$ operations on $O(N^2)$ data
- Matrix transpose analysis

Thank you for your attention!

Gianmarco Accordi
gianmarco.accordi@polimi.it