

Semester 1 SELT - survey journey

The Student Experience of Learning & Teaching survey allows students to give feedback on their courses and teachers. It is **voluntary** and **confidential**, and run by the Institutional Research (IR) team.

19 May – Survey opens

Check your email or
Wattle page for available
surveys



15 June - Survey closes

IR team perform screening of
comments for welfare concerns



Survey runs for 4 weeks

Please provide constructive and
respectful feedback (*your teacher
can't identify you*)

Find out more on the *Info for Students* webpage:

<https://services.anu.edu.au/learning-teaching/education-data/student-experience-of-learning-teaching-selt/information-for>

7 July

SELT feedback is made available
to teachers and course
convenors to improve future
course delivery



~27 June

Grades are released to
students



Australian
National
University



SELT - Frequently asked questions

What kind of feedback is helpful?

Think about your experience of the course and teaching, and what worked or didn't work for you.

When writing feedback, focus on respectful and constructive language – if you were a teacher, what type of feedback would help you improve the class?

Can teachers see who left specific feedback?

SELT is confidential, and teachers cannot see, or ask to see, the identity of a respondent. Unless you self-identify, for example by using names or describing specific events, teachers cannot identify you.

If your class has very few enrolments, it may be difficult to remain completely anonymous.



Australian
National
University

Find out more on the *Info for Students* webpage:
<https://services.anu.edu.au/learning-teaching/education-data/student-experience-of-learning-teaching-selt/information-for>



COMP4300 - Course Update

➤ Final Exam (40%)

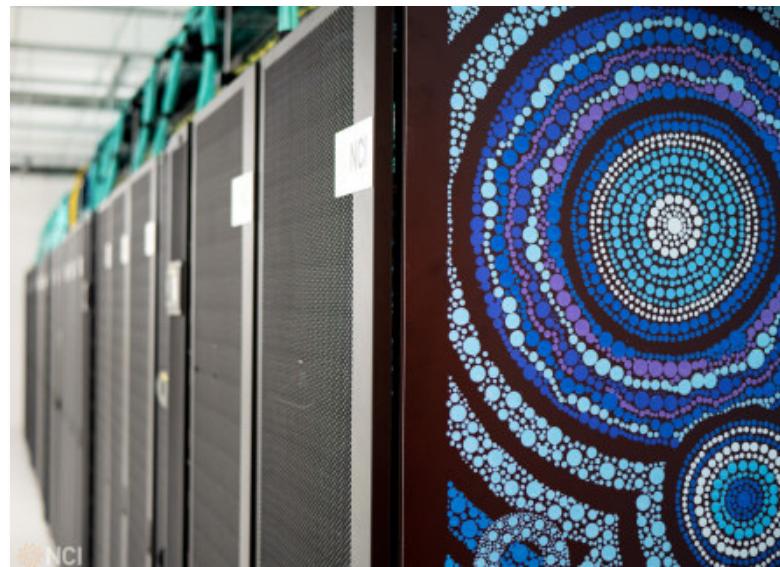
- Wednesday 11/06/2025 at 2:00pm at Copland G31 (Building 24)
- The exam will cover all materials presented in the course e.g. in labs, lectures and assignments etc
- Course/lecture notes permitted.

➤ Assignment 2

- Released on 24 April
- **Due next Monday 26/05/2025, 11:55PM**



PERFORMANCE ANALYSIS WITH THE ROOFLINE MODEL (CPU & GPU)



Australian
National
University

References

Accelerating HPC Applications with NVIDIA Nsight Compute Roofline

Analysis <https://developer.nvidia.com/blog/accelerating-hpc-applications-with-nsight-compute-roofline-analysis/>

C. Yang, T. Kurth, and S. Williams, **Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system**, Concurrency and Computation: Practice and Experience, e5547, 2019. <https://doi.org/10.1002/cpe.5547>

[Analyze CPU Roofline](#)

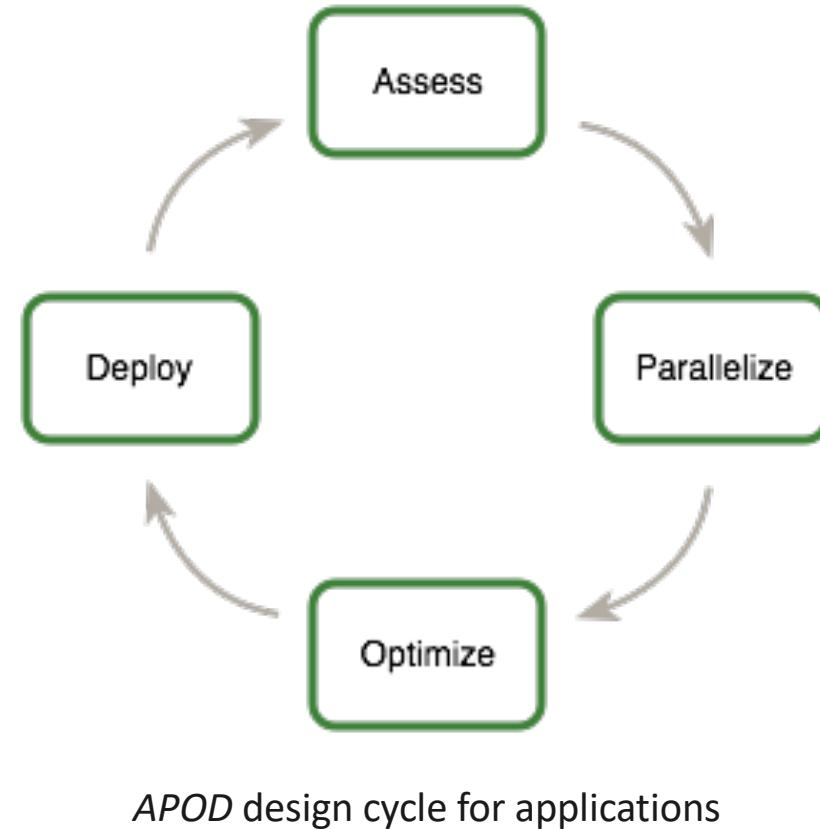
<https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2024-1/analyze-cpu-roofline.html>



Code Optimisation

Assess, Parallelize, Optimize, Deploy (APOD)

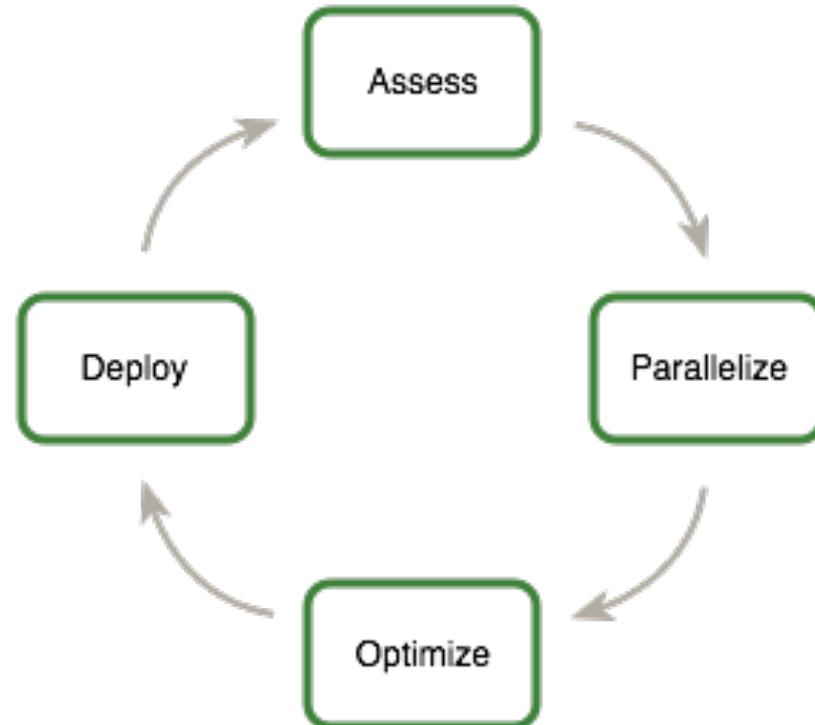
- APOD is a cyclical process
- Initial speedups can be achieved, tested, and deployed with only minimal initial investment of time
- The cycle can begin again by identifying further optimization opportunities, seeing additional speedups
- Allows for incremental deployment of the even faster versions of the application into production



Code Optimisation

Assess, Parallelize, Optimize, Deploy(APOD)

- By understanding the end-user's requirements and constraints and by applying Amdahl's and Gustafson's laws, the developer can determine the likely upper bound of performance improvement from acceleration of the identified portions of the application.

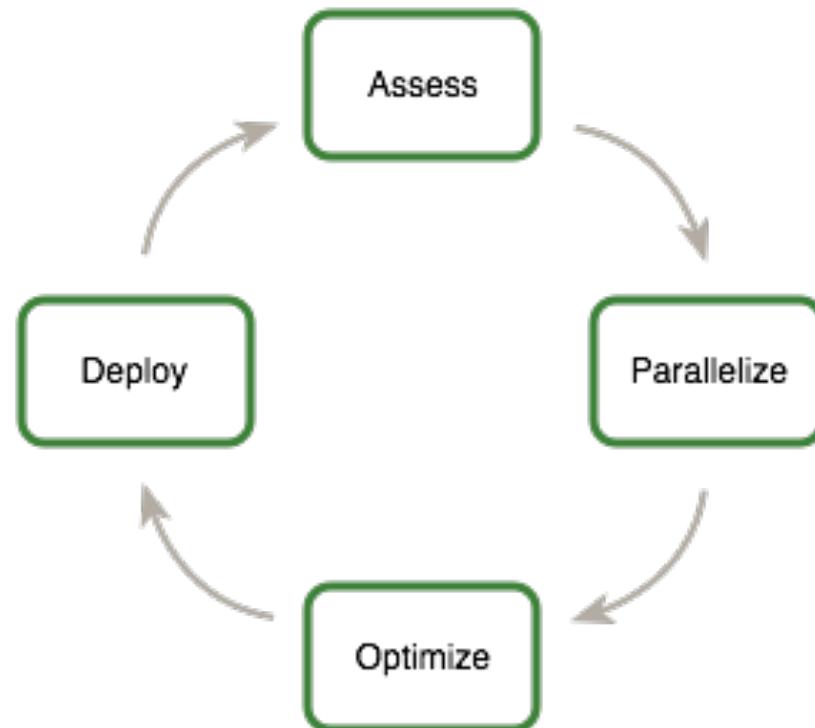


APOD design cycle for applications

Code Optimisation

Assess, Parallelize, Optimize, Deploy (APOD)

- Having identified the hotspots and having done the basic exercises to set goals and expectations, the developer needs to parallelize the code.
- Depending on the original code, this can be as simple as calling into an existing GPU-optimized library such as cuBLAS or adding a few preprocessor directives as hints to a parallelizing compiler.
- Some applications' designs will require some amount of refactoring to expose their inherent parallelism

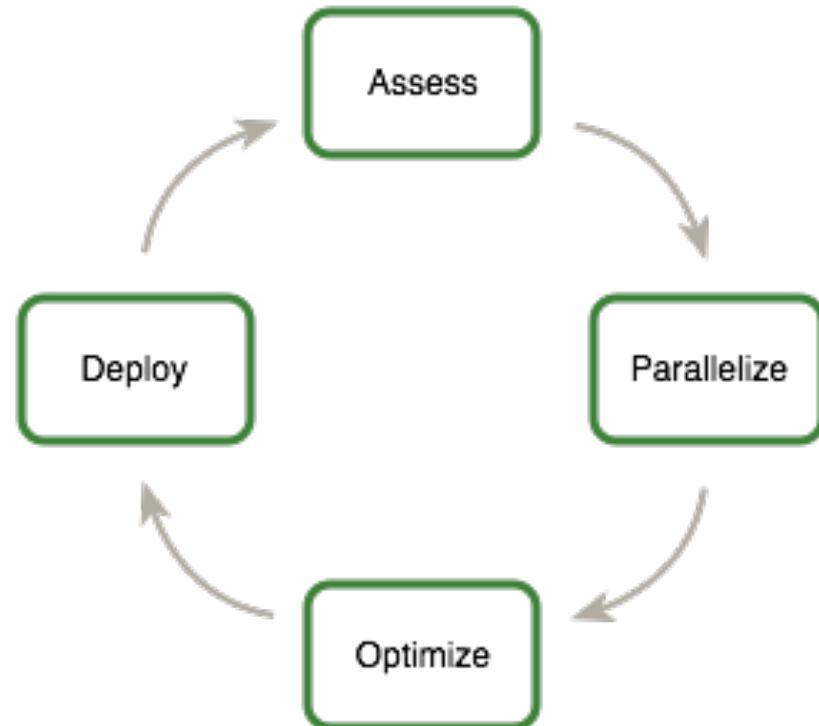


APOD design cycle for applications

Code Optimisation

*Assess, Parallelize, **Optimize**, Deploy(APOD)*

- After each round of application parallelization is complete, the developer can move to optimizing the implementation to improve performance
- Optimizations can be applied at various levels e.g. from overlapping data transfers with computation, all the way down to fine-tuning floating-point operation sequences
- The available profiling tools are invaluable for guiding this process

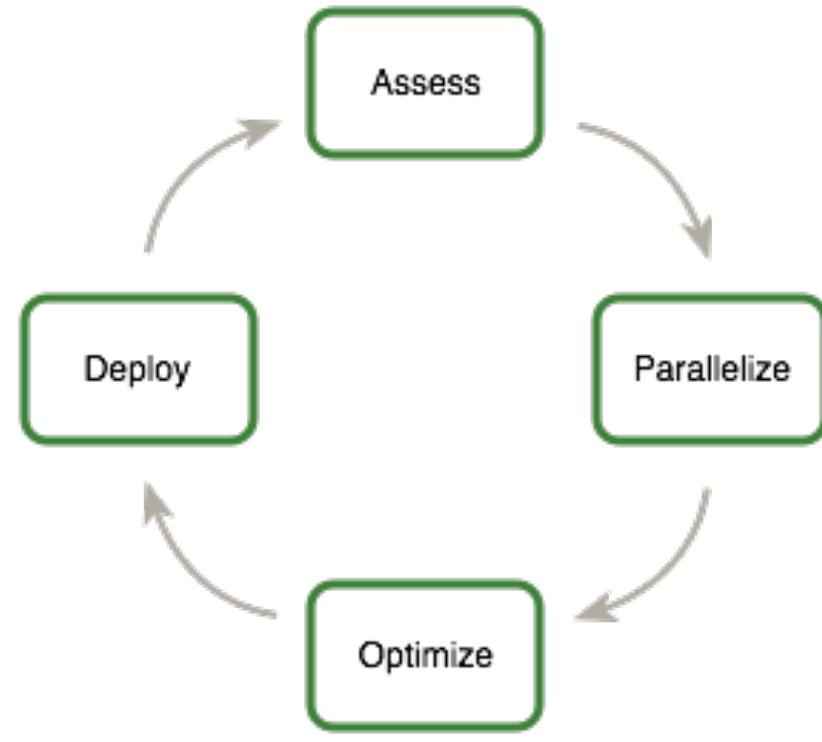


APOD design cycle for applications

CPU and GPU Code Optimisation

Assess, Parallelize, Optimize, Deploy (APOD)

- Having completed the GPU acceleration of one or more components of the application it is possible to compare the outcome with the original expectations developed in the *assess* step
- The partially parallelized implementation can be carried through to production as it allows the user to profit from their investment as early as possible (the speedup may be partial but is still valuable)



APOD design cycle for applications



Performance Models

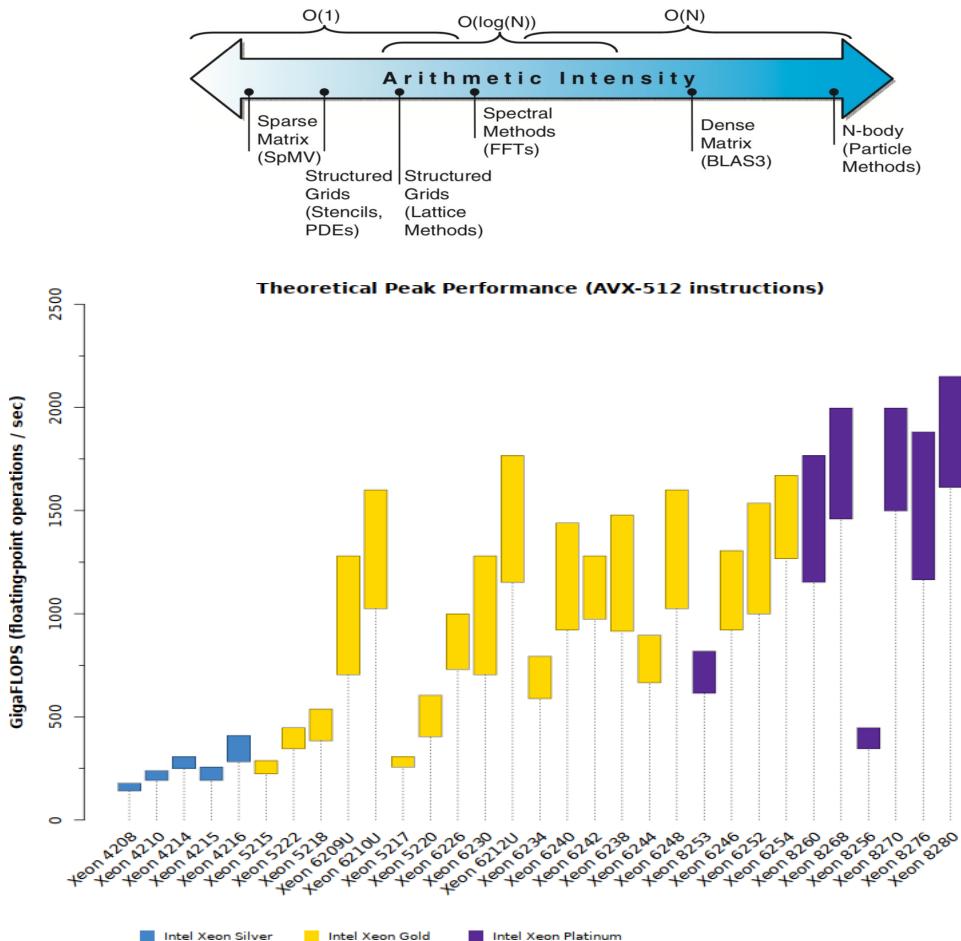
We need a quantitative model that defines good performance with reference to the specific hardware available. Good performance is defined by two fundamental requirements:

Scalability requirement: Must not be significantly penalized by serialization (Amdahl's Law), load imbalance or communication (parallel overhead, synchronization)

Efficiency requirement: Must attain high utilization of the CPU's compute and/or bandwidth capabilities

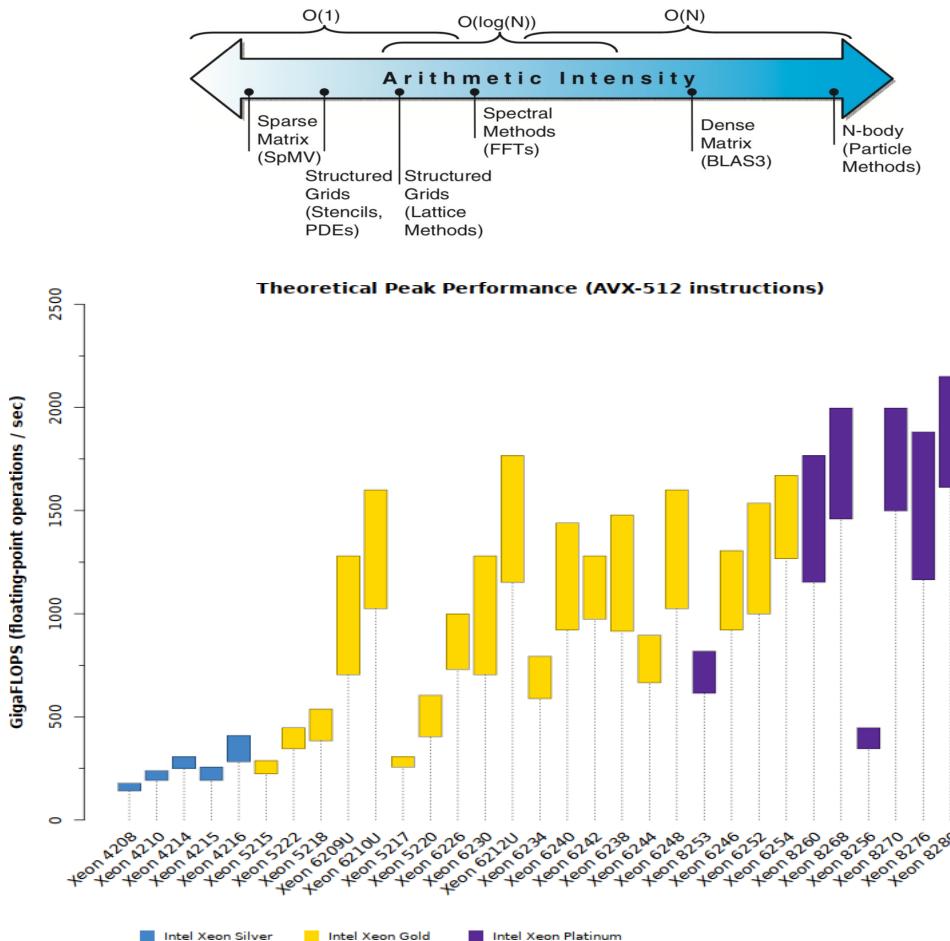


Performance Metrics



- In order to find such a model let's start by looking at some simple algorithmic motifs which are common in scientific computing.
- Clearly, a first limit for their execution speed is the peak *floating point operations* (FLOPs) performance of the hardware, measured in FLOPs/s (Figure: Cascade Lake <https://www.microway.com>)
- Another limit is *main memory bandwidth*, which restricts the speed of data transfers from and to the CPU.
- The more FLOPs/s an application performs *per unit of data transferred*, the more likely the code is to reach peak performance.

Performance Metrics



The key metric is the **arithmetic intensity**:

$$AI = \frac{FLOP/s}{Bytes/s} \quad (1)$$

where the denominator is the number of Bytes read from and written to main memory per second.

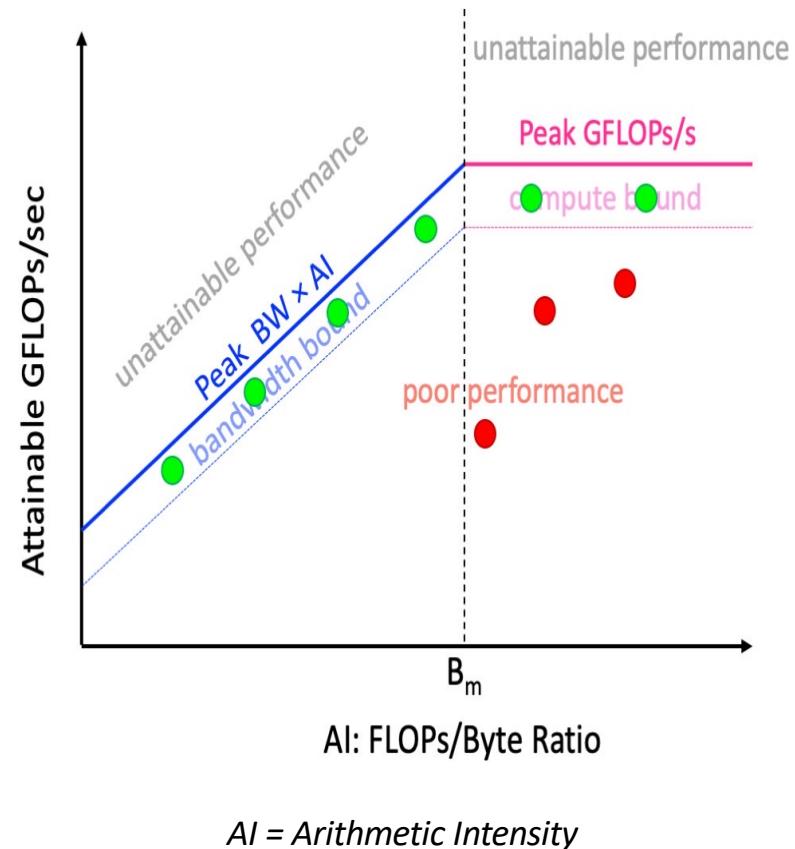
- The AI can be calculated taking the total number of FLOPs divided by the Bytes transferred to and from main memory during program execution.
- An especially important value is the *machine balance* B_m : the ratio between the peak FLOP performance and the memory bandwidth

$$B_m = \frac{\text{Peak Performance (FLOP/s)}}{\text{Memory bandwidth (Bytes/s)}} \quad (2)$$



The Roofline Model

- This is a simple but very powerful model that ties FLOP performance, AI and memory performance in a 2D graph. [William, Waterman, and Patterson, Communications of the ACM Volume 52, Number 4 (2009), Pages 65-76]
- One good way to find peak memory performance is to use the STREAM benchmarks.
- For a given code snippet (kernel) one can find a point on the horizontal axis by measuring its AI.
- The performance of that kernel lies on the vertical line through that point.

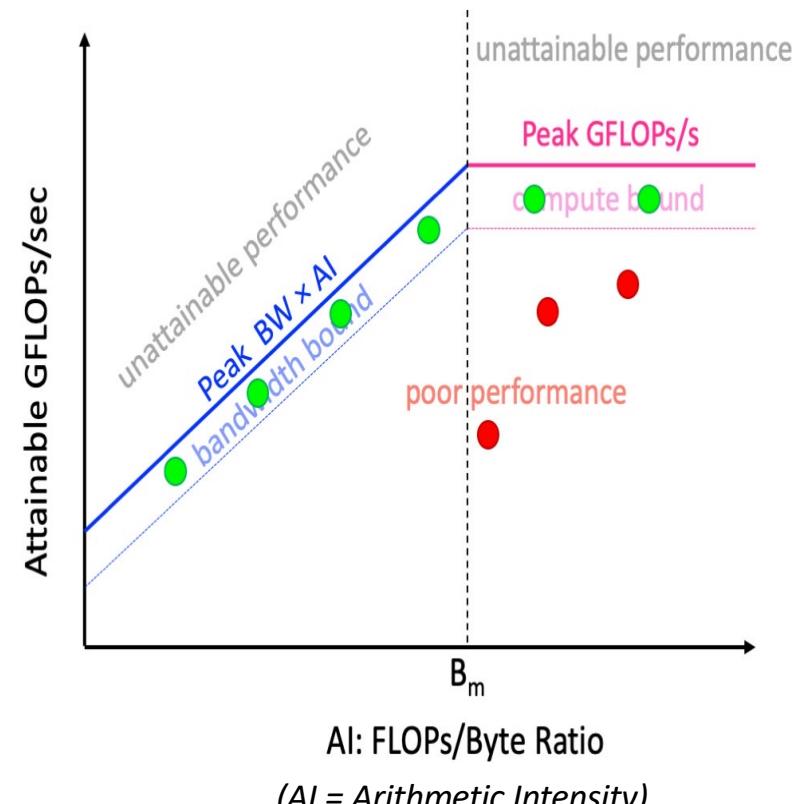


The Roofline Model

- For each given AI, the “roofline” is the maximum FLOP performance achievable by the code on the specific hardware architecture used.

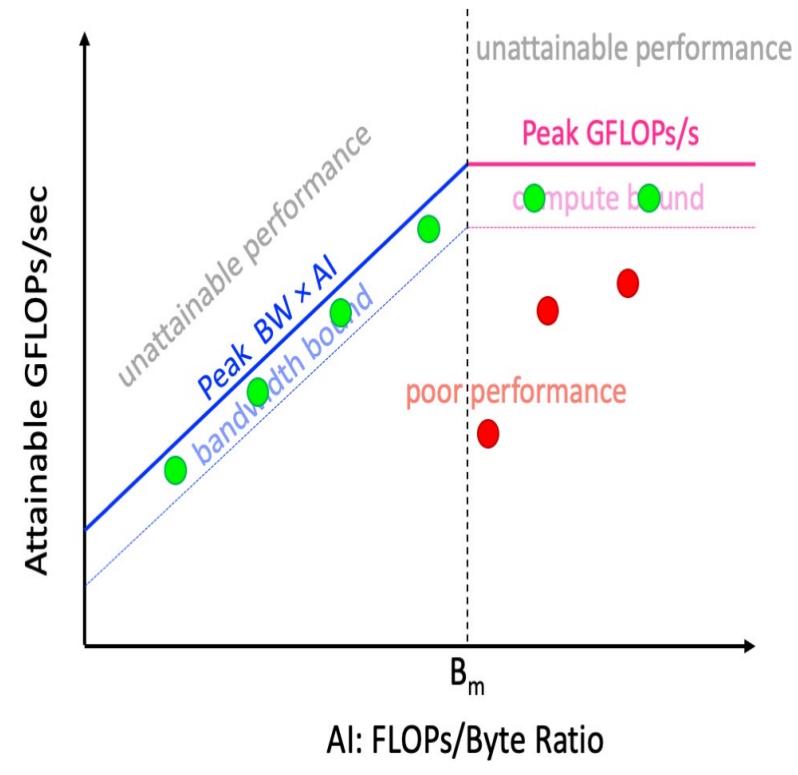
$$\text{Max Attainable GFLOP/s} = \min \begin{cases} \text{Peak Memory BW} * \text{AI} \\ \text{Peak FLOP Performance} \end{cases}$$

- There are four performance regions: -
 - unattainable performance
 - bandwidth bound performance
 - compute bound performance
 - poor performance
- The ridge point has abscissa equal to the machine balance.



The Roofline Model

- Kernels that lie close to the roofline are making good use of the hardware resources.
- Kernels can have low performance (GFLOPs/s), but make good use (% STREAM) of a machine.
- Kernels can have a relatively high performance (GFLOPs/s), but still make poor use (% Peak GFLOPs/s) of a machine.

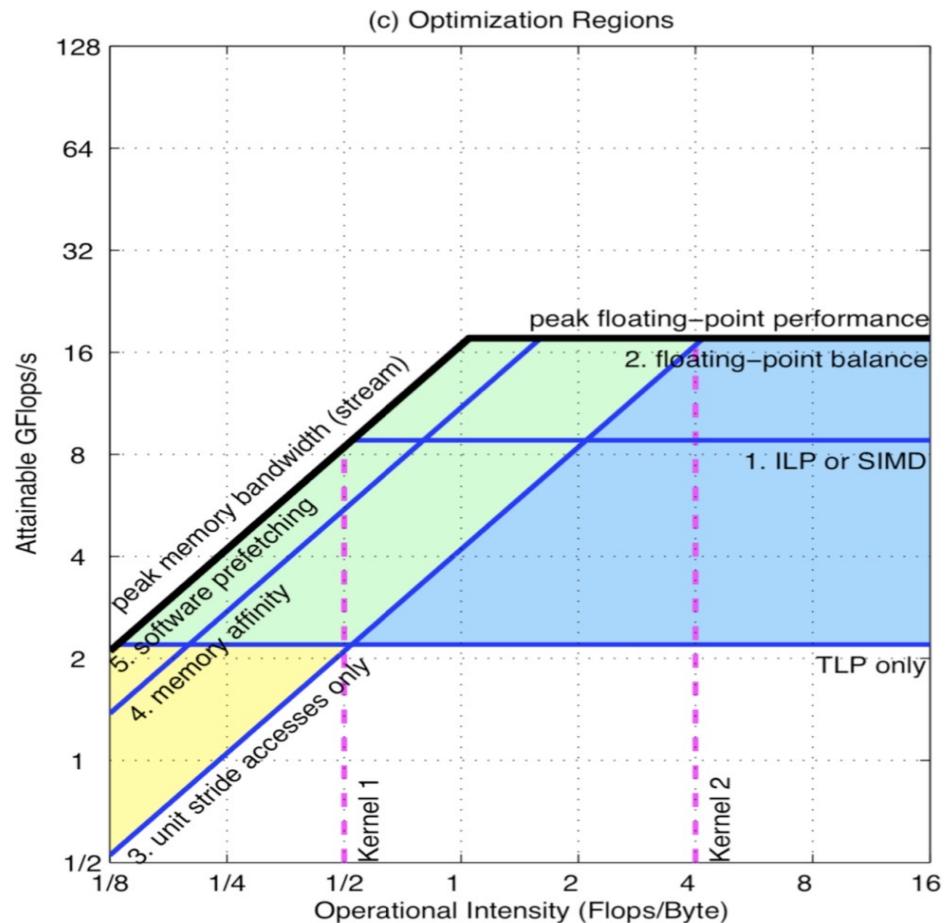


AI = Arithmetic Intensity



CPU Code under the Roofline

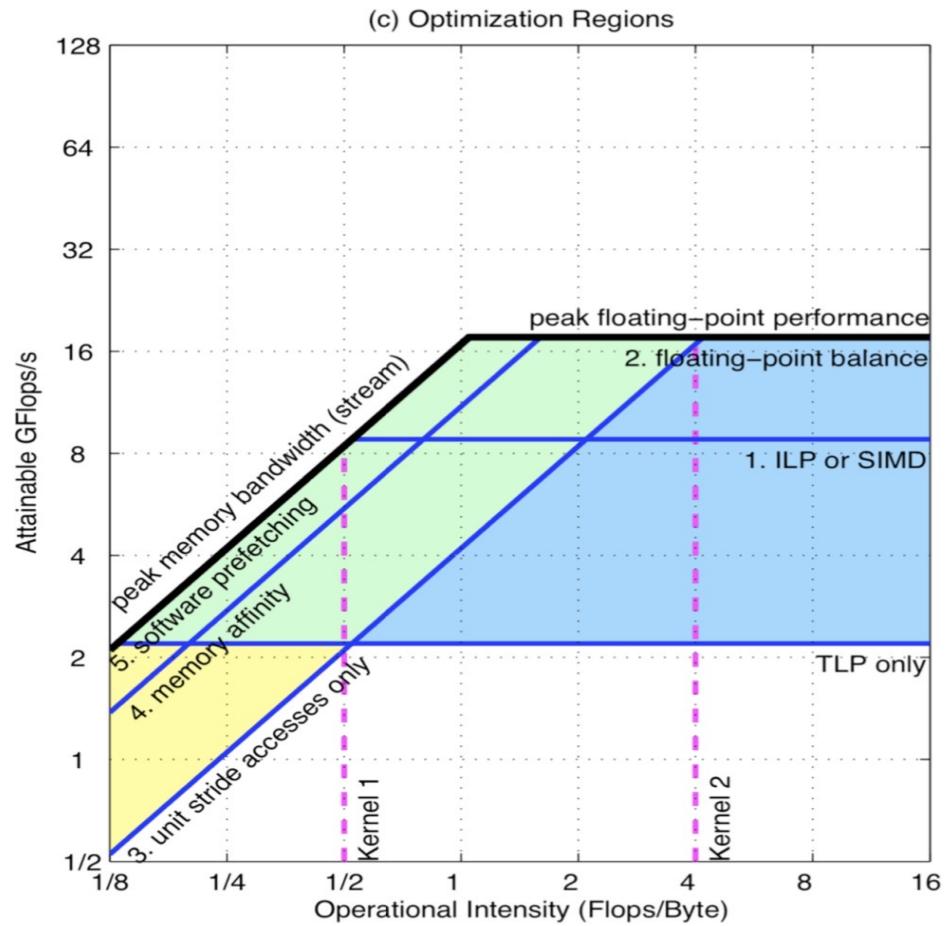
How can your CPU kernel lie in the poor performance region?



CPU Code under the Roofline

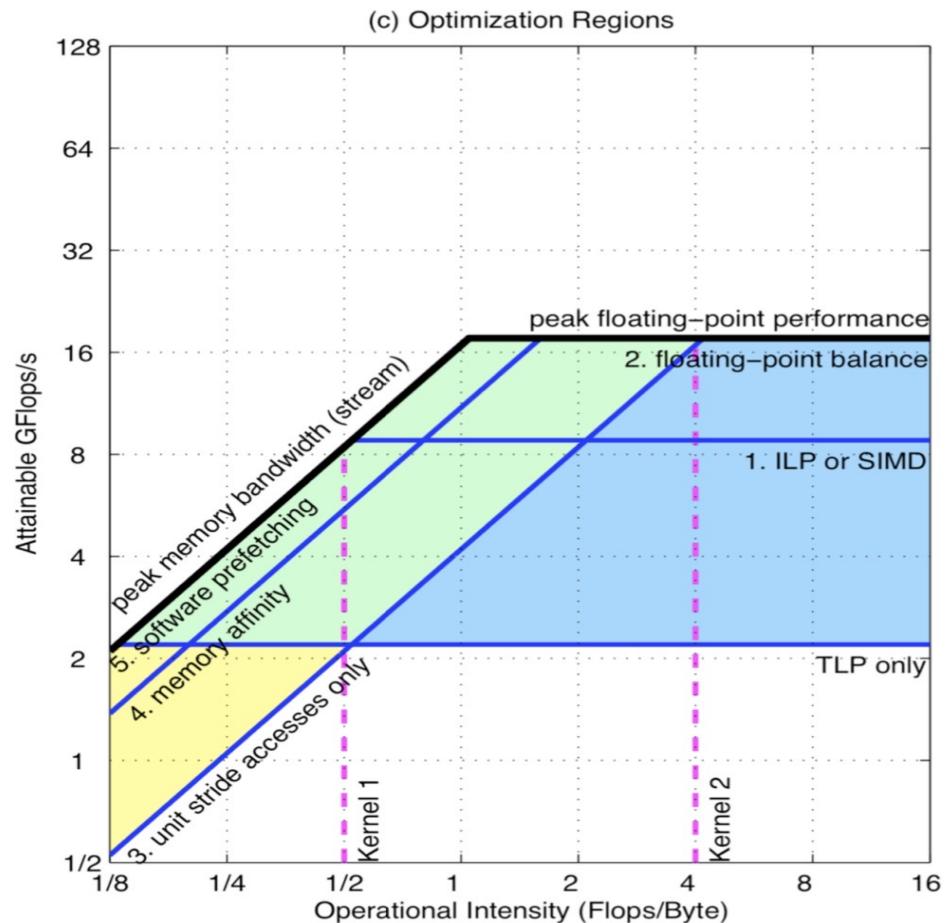
How can you CPU kernel lie in the poor performance region?

- A likely reason is the lack of some specific optimizations that enable the program to use the underlying hardware efficiently.



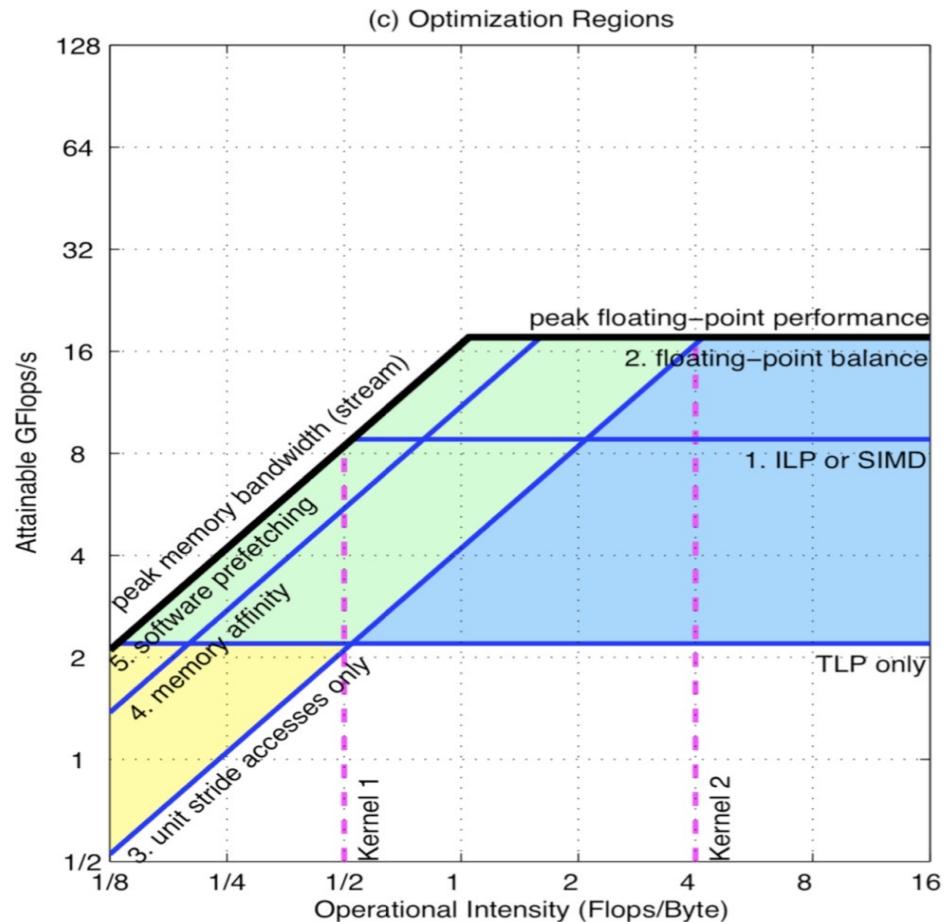
CPU Code under the Roofline

- Each optimization constitutes a “performance ceiling” below the appropriate roofline.
- The horizontal lines are performance rooflines associated with FLOP performance.
- The tilted lines are associated with memory performance.



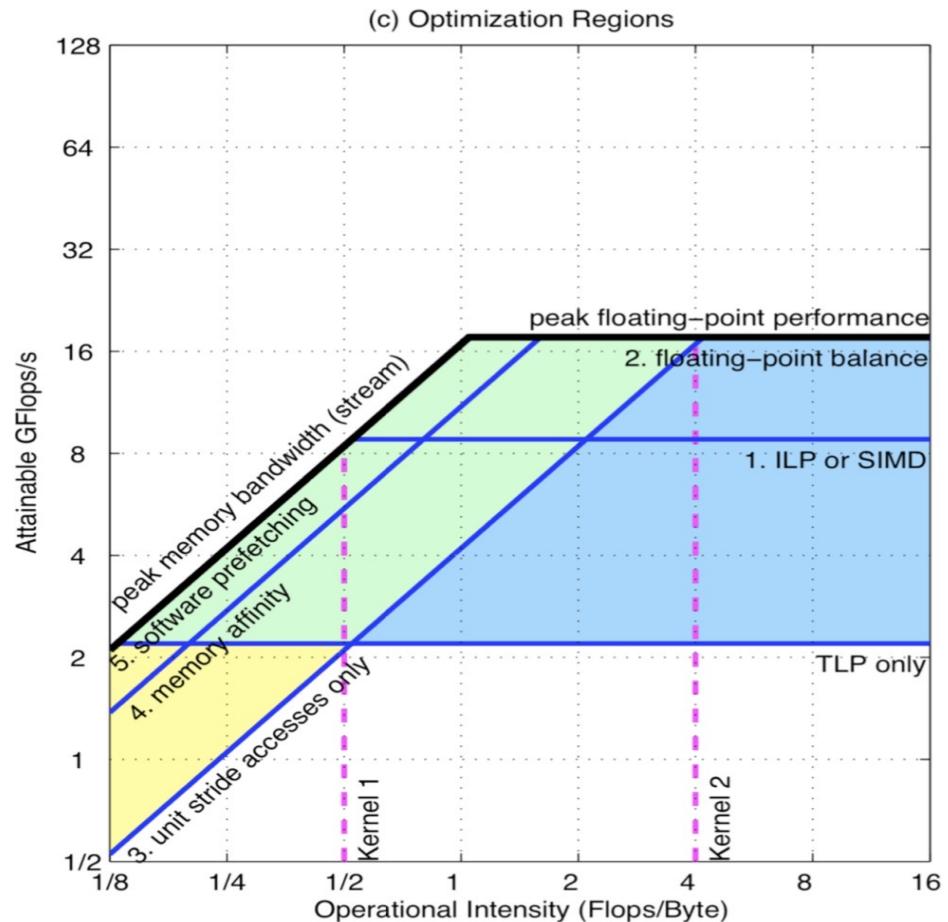
CPU Code under the Roofline

- The first FLOP performance ceiling is associated with *Thread Level Parallelism (TLP)*.
 - In order to achieve reasonable performance on modern processors one must use all cores in parallel.
- The second FLOP performance ceiling is associated with *Instruction Level Parallelism (ILP)* and *SIMD parallelism*.
 - Maximize ILP is about hiding completely functional unit latency: loop unrolling, loop fusion, avoid branching in inner loops, etc.
 - Need to use SIMD instructions in order to use all processors floating point arithmetic units.



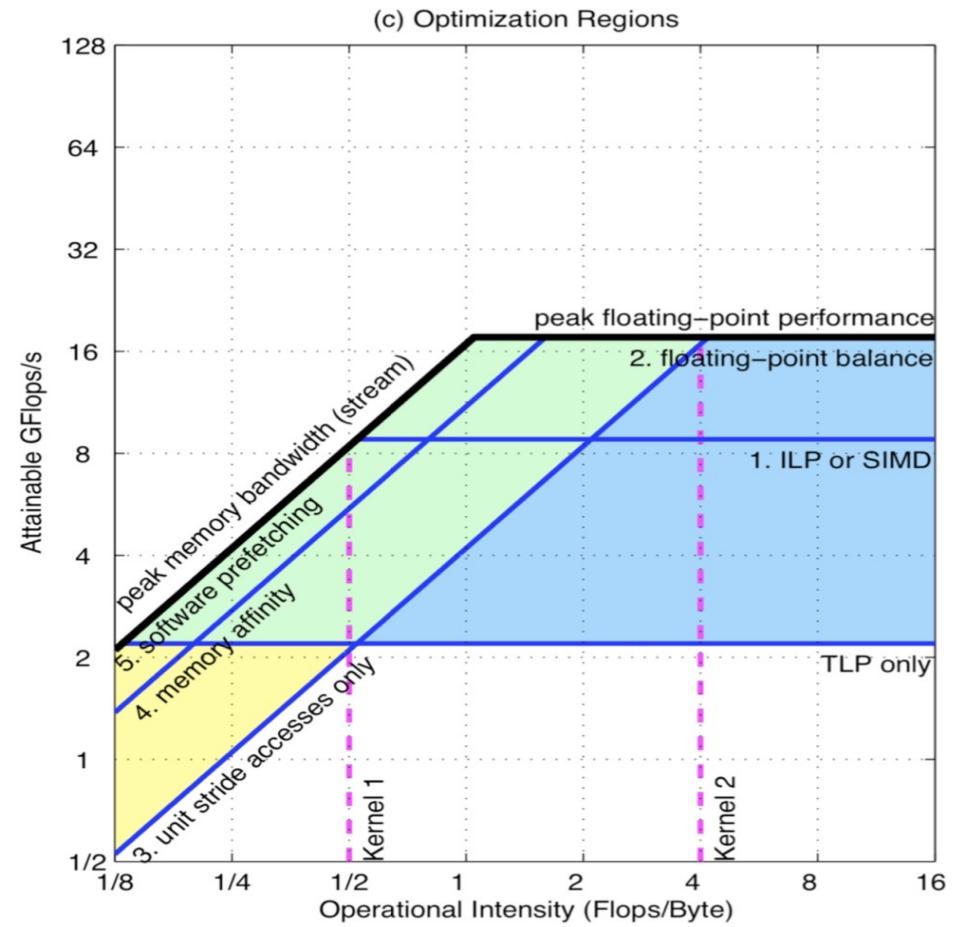
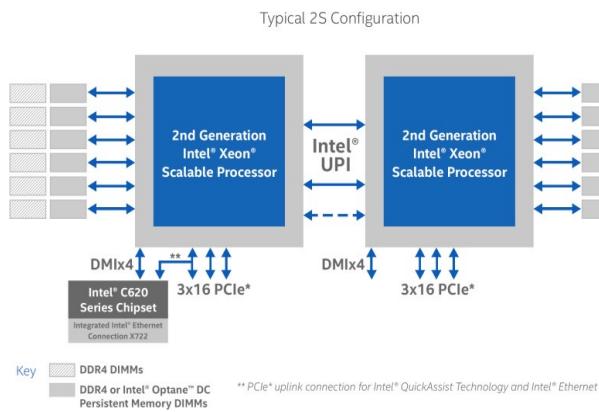
CPU Code under the Roofline

- The final FLOP performance ceiling is associated with *floating point instruction mix*.
- Peak FLOP performance requires an equal number of nearly simultaneous additions and multiplications e.g. $(a*x) + b$
- This balance is necessary typically because the computer supports a fused multiply-add (FMA) instruction.
- Another reason may be also that FP units have an equal number of FP adders and FP multipliers.



CPU Code under the Roofline

- The first memory performance ceiling is associated with *strided access*.
- In order to reduce main memory traffic, inner loops must have a *unit stride* access pattern.
- The second is about *memory affinity*, that is NUMA effects.
- Allocate data and the threads tasked to operate on that data to the same memory/numa domain pair



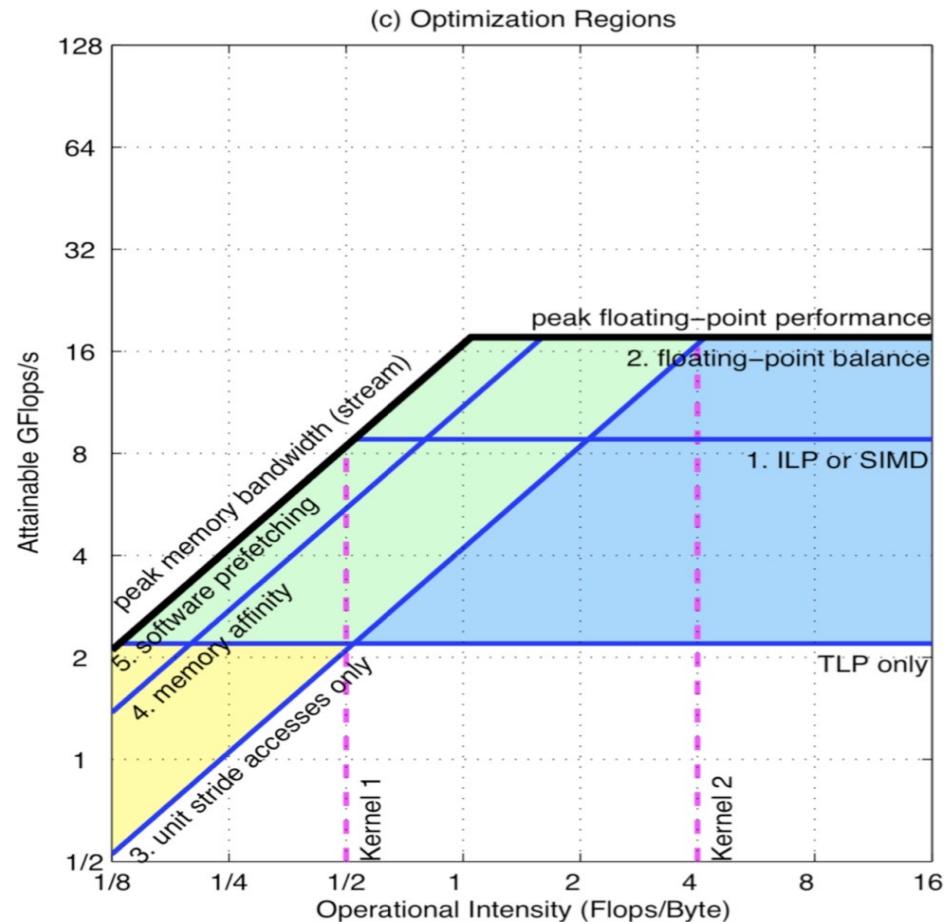
CPU Code under the Roofline

- The final memory performance ceiling is associated with *software prefetching*.

```
# pragma prefetch
```

```
# include <immintrin.h>
void _mm_prefetch (char const* p,
    int i)
```

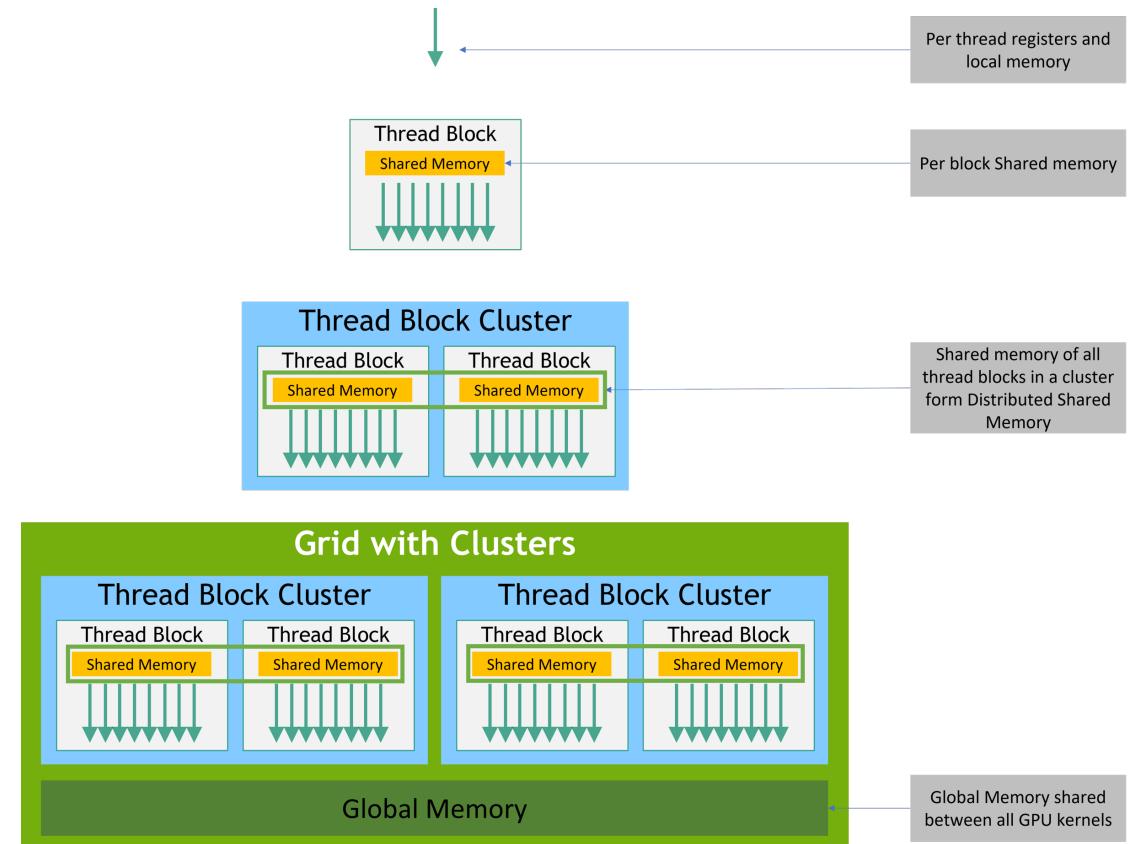
- Avoid branching in bottleneck-determining code, unroll loops, etc.



GPU Code Optimisation

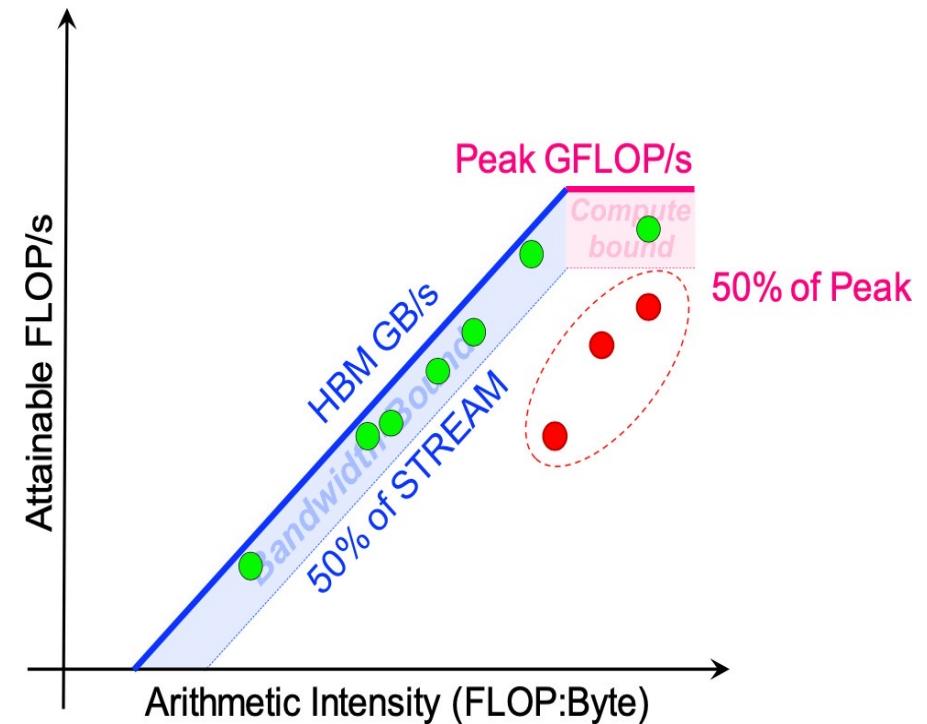
GPU Performance optimization revolves around four basic strategies:

- *Maximize parallel execution* to achieve maximum utilization
- *Optimize memory usage* to achieve maximum memory throughput
- *Optimize instruction usage* to achieve maximum instruction throughput
- *Minimize memory thrashing* by not constantly allocating and freeing memory



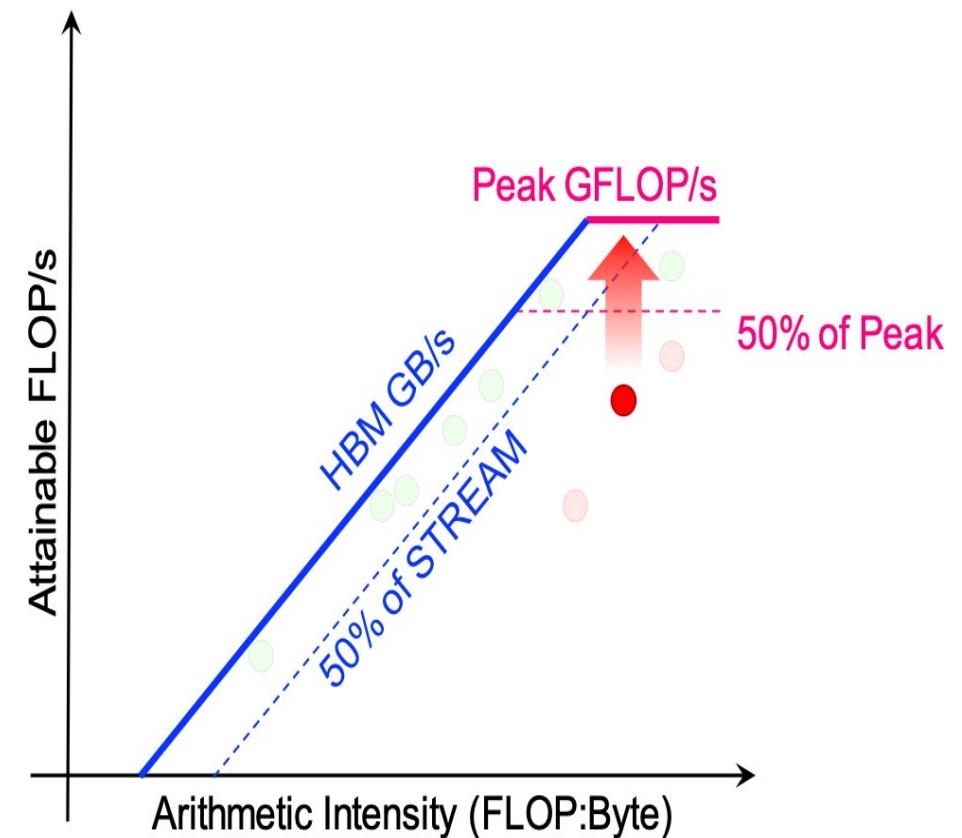
GPU Code under the Roofline

- Typically reasons for low performance are architecture (and kernel) dependent
- Why could the kernel corresponding to red dots have low performance?
- How can we increase performance?



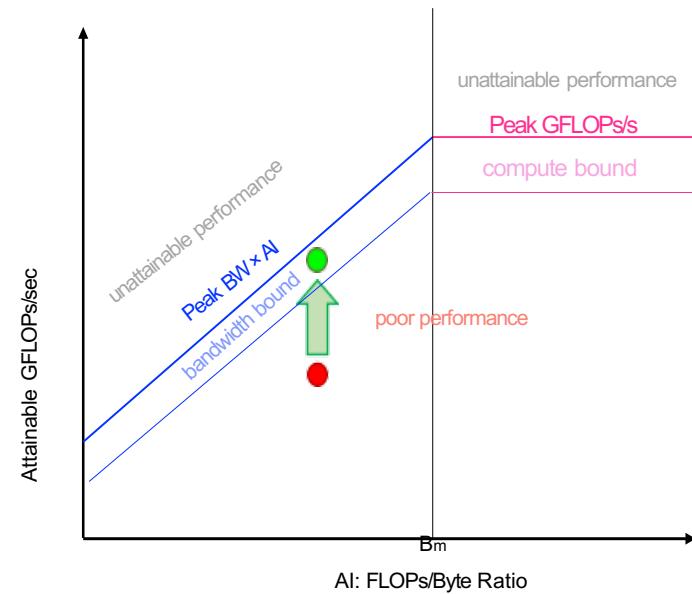
GPU Code under the Roofline

- Expose more parallelism to maximise SM usage and hide arithmetic latency
 - Change grid and block sizes (kernel configuration) to increase block-level parallelism
 - Unroll loops
 - Tuning number of registers per thread
- Avoid thread divergence



GPU Code under the Roofline

- What if we want better bandwidth utilization?



GPU Code under the Roofline

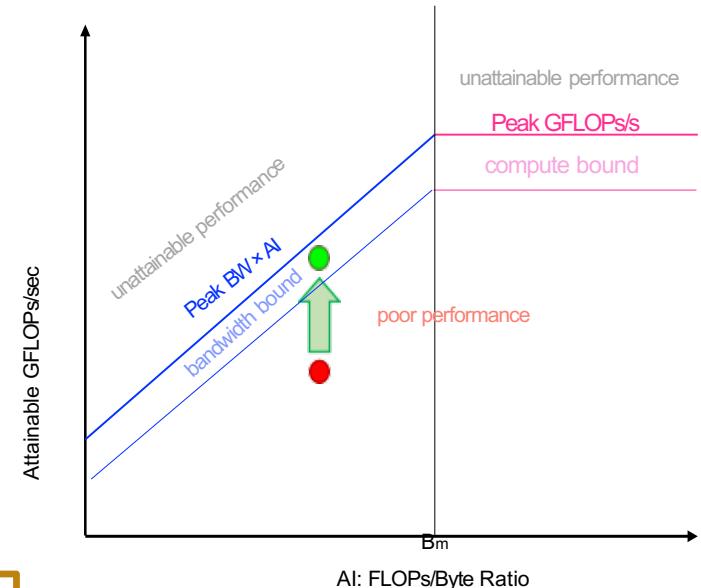
- What if we want better bandwidth utilization?
 - Optimize for obtaining aligned and coalesced memory transactions
 - This may require a redesign of data structures (e.g. AoS to SoA) and better indexing
 - Expose more parallelism to maximise SM usage and hide memory latency

Array of Structures (AoS)

```
struct point3D {  
    float x;  
    float y;  
    float z;  
};  
struct point3D points[N];
```

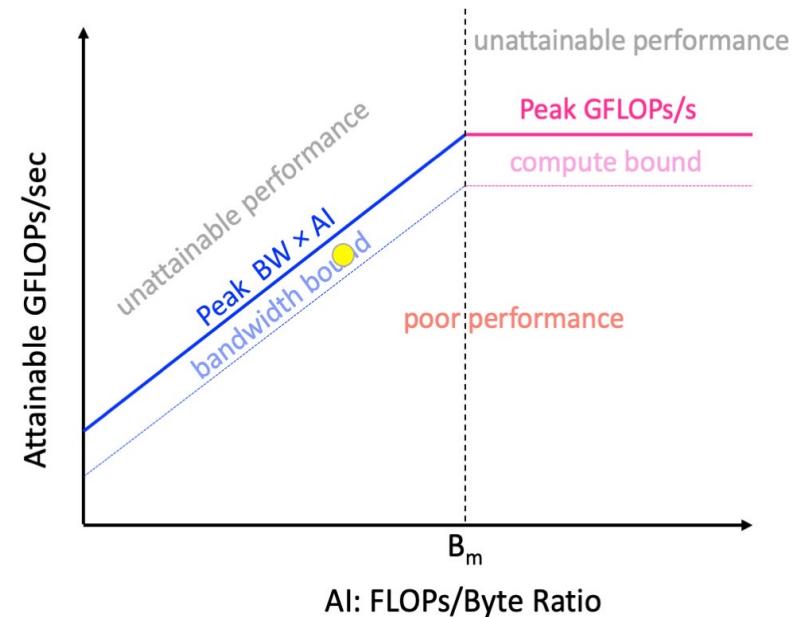
Structure of Arrays (SoA)

```
struct pointlist3D {  
    float x[N];  
    float y[N];  
    float z[N];  
};  
struct pointlist3D points;
```



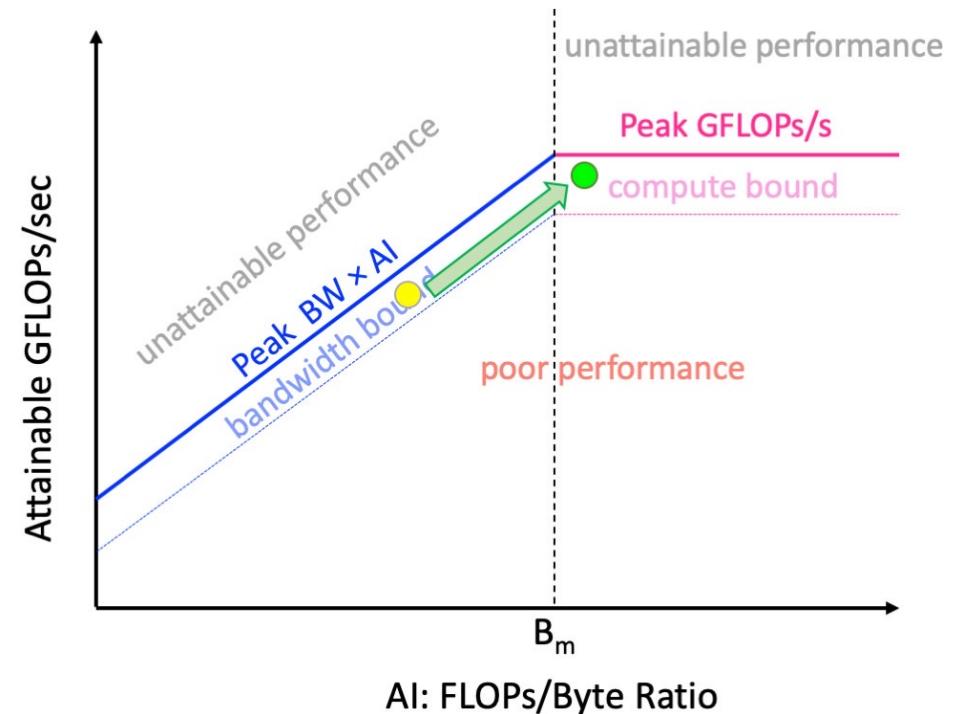
High Performance GPU Code

- Assume your kernels yields the yellow dot performance.
 - Are you happy with this level of performance...?



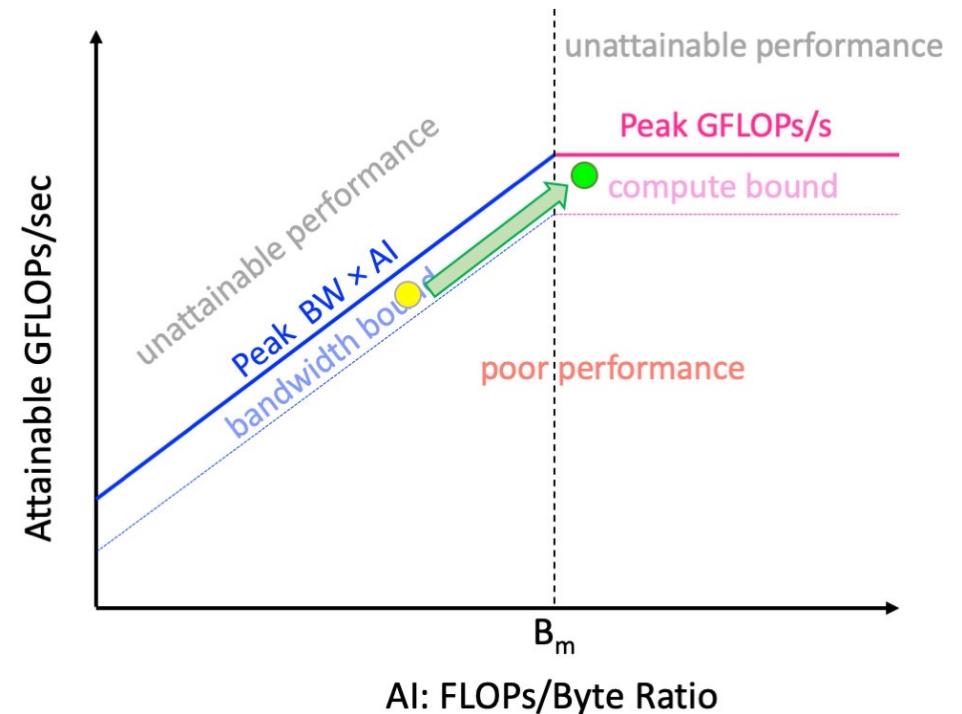
Increasing AI for GPU Code

- Assume your kernels yields the yellow dot performance.
 - Are you happy with this level of performance...?
 - *How can we increase AI?*



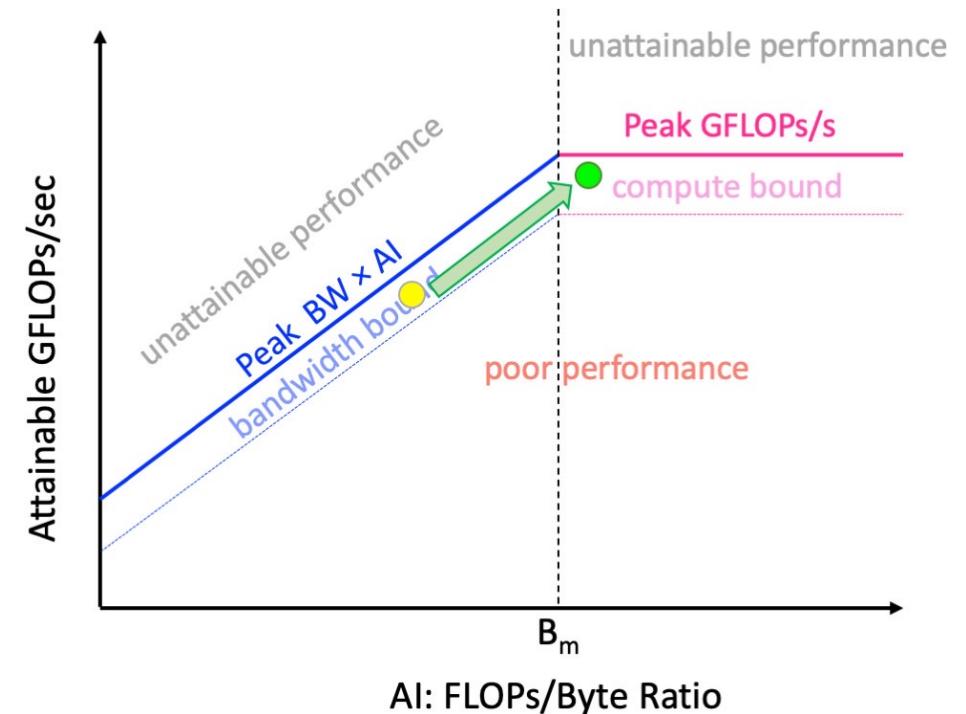
Increasing AI for GPU Code

- Assume your kernels yields the yellow dot performance. Are you happy with it?
- How can we increase AI?
 - *Increasing the data reuse and reducing the data movement increases AI i.e. reducing data movement*



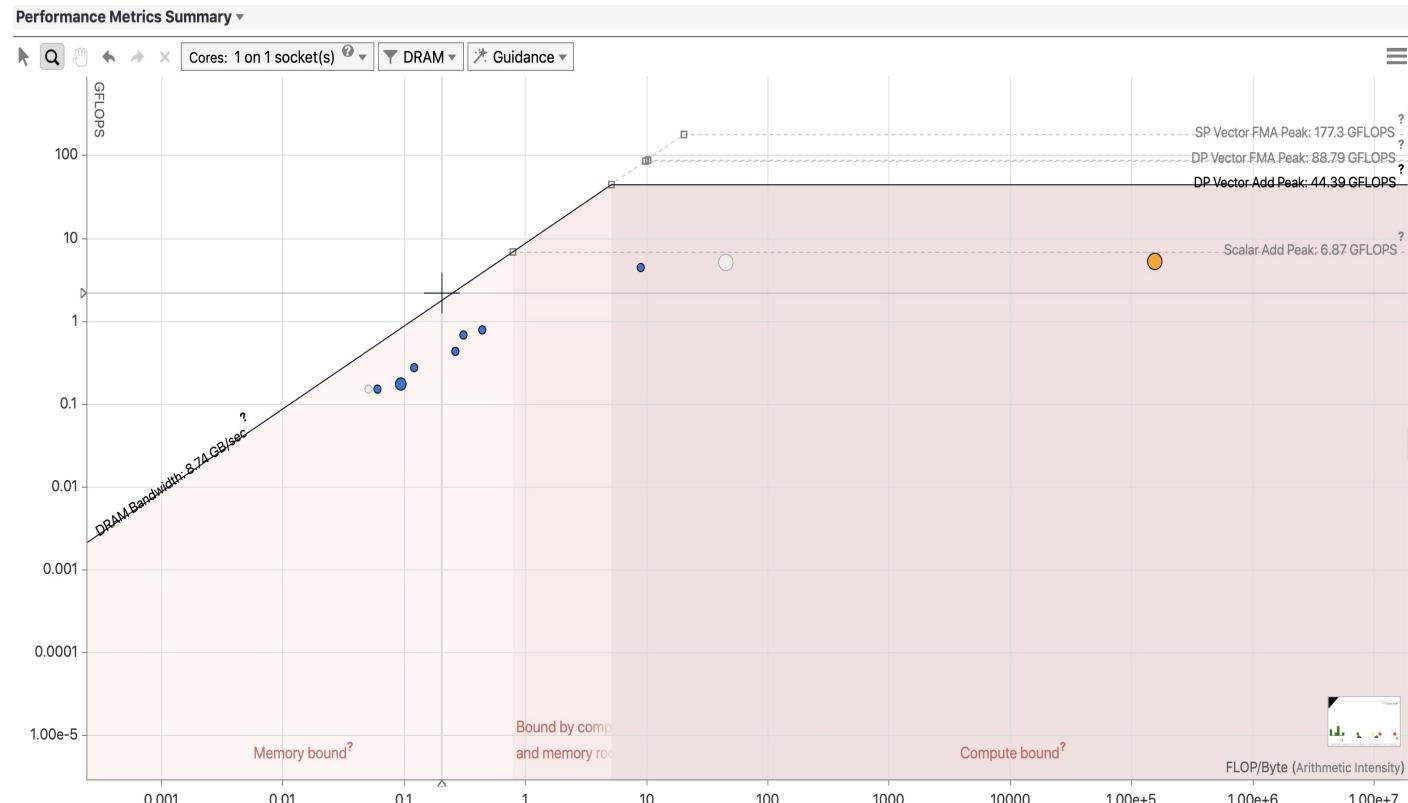
Increasing AI for GPU Code

- Assume your kernels yields the yellow dot performance. Are you happy with it?
- How can we increase AI?
 - Increasing the data reuse and reducing the data movement increases AI
 - Better caching → Optimize code to be cache friendly (avoid false sharing on CPU)
 - Usage of shared memory on GPU to reduce global memory transactions
 - Also, you may be able to achieve better performance with a fundamental algorithm redesign!



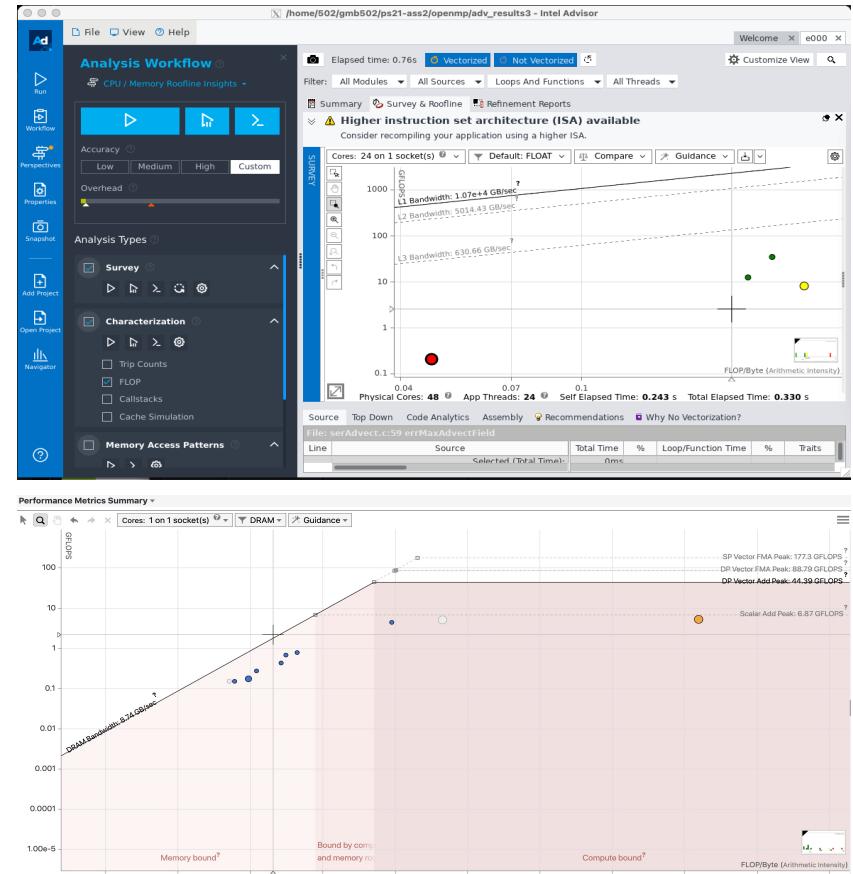
Roofline Analysis with Intel Advisor

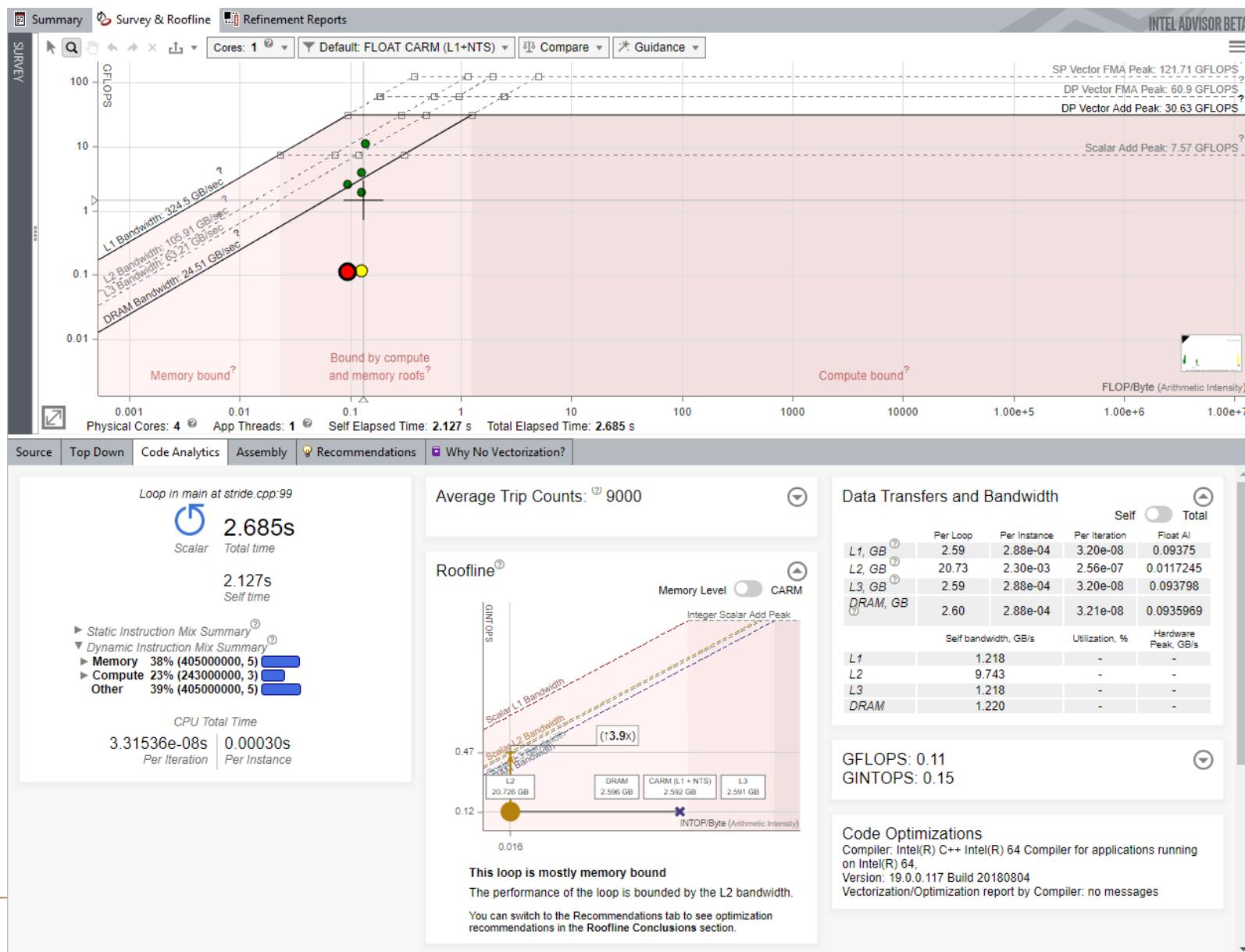
- You can use the *Intel Advisor tool* to produce a roofline plot for CPU



Roofline Analysis with Intel Advisor

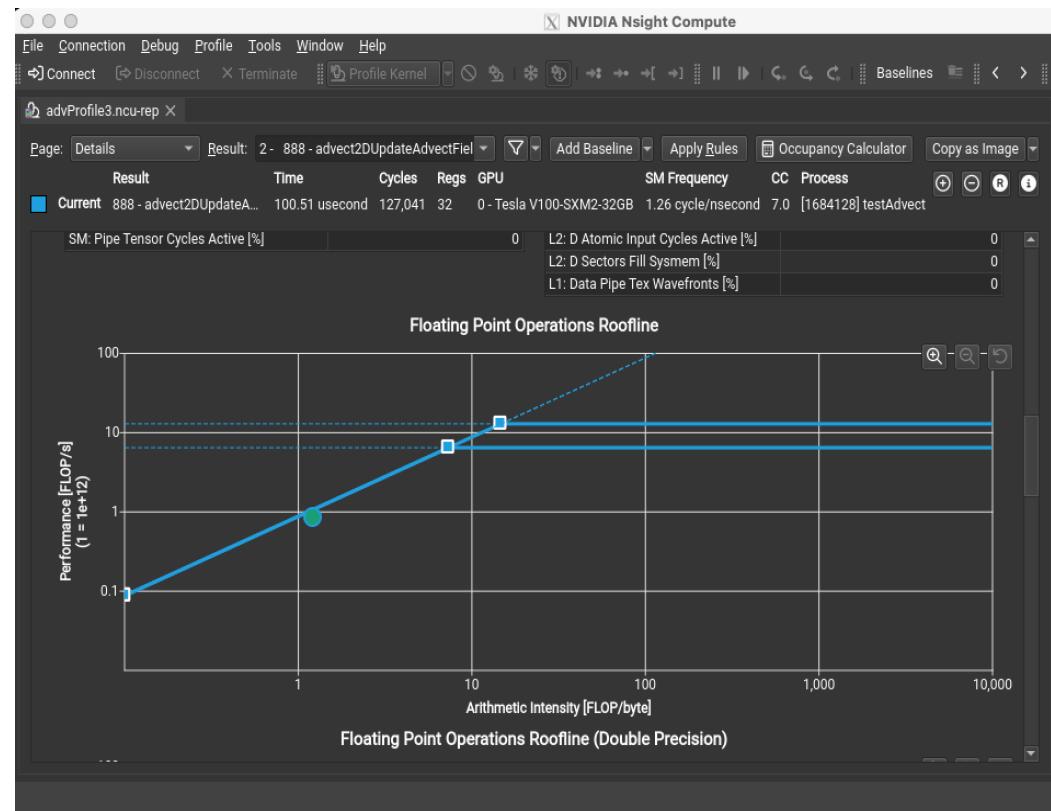
- Ideally, compile your program with icx, not gcc
- You need full isolation from other users when profiling
- module load intel-advisor
- ```
OMP NUM THREADS=NT advixe-cl --collect
 roofline --project-dir ./results dir --
 search-dir src:=./ --search-dir bin:=./ --
 ./testAdvect -P M N
```
- advixe-gui to launch the GUI





# Roofline Analysis with Nvidia Nsight Compute

- You can use NVIDIA Nsight Compute to produce a roofline plot for GPU



# Roofline Analysis with Nvidia Nsight Compute

**Kernel\_A** is an instruction throughput bound kernel: with each thread we do 10000 double precision adds, for an arithmetic intensity of  $10000 / 8$  (bytes per double precision word) = 1250.

```
__global__ void kernel_A(double* A, int N, int M)
{
 double d = 0.0;
 int idx = threadIdx.x + blockIdx.x * blockDim.x;

 if (idx < N) {
#pragma unroll(100)
 for (int j = 0; j < M; ++j) {
 d += A[idx];
 }
 A[idx] = d;
 }
}
```



# Roofline Analysis with Nvidia Nsight Compute

**Kernel\_B** is identical to **kernel\_A**, except that we artificially throttle occupancy on the GPU by allocating 96 kB of shared memory per thread block, which means that only one thread block can be resident on an SM at any one time, for an occupancy of  $1/32 = 3.125\%$ .

```
__global__ void kernel_A(double* A, int N, int M)
{
 double d = 0.0;
 int idx = threadIdx.x + blockIdx.x * blockDim.x;

 if (idx < N) {
#pragma unroll(100)
 for (int j = 0; j < M; ++j) {
 d += A[idx];
 }
 A[idx] = d;
 }
}
```

```
cudaFuncSetAttribute(kernel_B,
cudaFuncAttributeMaxDynamicSharedMemorySize, 96 * 1024);
kernel_B<<<numBlocks, threadsPerBlock, 96 * 1024>>>(A, N, M);
```



# Roofline Analysis with Nvidia Nsight Compute

**Kernel\_C** is primarily memory-bandwidth bound -- we just do a single double precision add, combined with a load and a store (for an arithmetic intensity of  $1 / 16 = .0625$ ). The memory access pattern is strided -- we load every element of B exactly once and store every element of A exactly once, but any given warp is accessing memory locations 32 bytes apart between each thread

```
__global__ void kernel_C(double* A, const double* B, int N)
{
 int idx = threadIdx.x + blockIdx.x * blockDim.x;

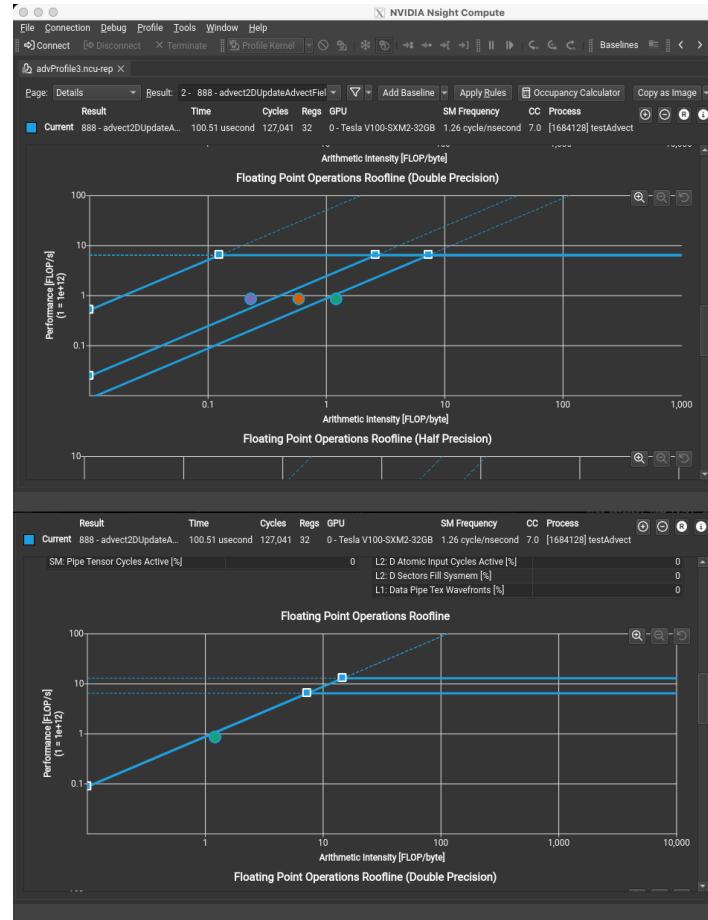
 // Strided memory access: warp 0 accesses (0,stride, 2*stride,
 ...), warp 1 accesses
 // (1, stride + 1, 2*stride + 1, ...).
 const int stride = 16;
 int strided_idx = threadIdx.x * stride + blockIdx.x % stride +
(blockIdx.x / stride) * stride * blockDim.x;

 if (strided_idx < N) {
 A[idx] = B[strided_idx] + B[strided_idx];
 }
}
```



# Roofline Analysis with Nvidia Nsight Compute

- module load cuda
- ncu --set full -o ProfileOutput <application>
- ncu-ui to launch the GUI



# DEMO

## Roofline Analysis with NVIDIA Nsight Compute

