INTRODUCTION TO GPU ARCHITECTURE & PROGRAMMING

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

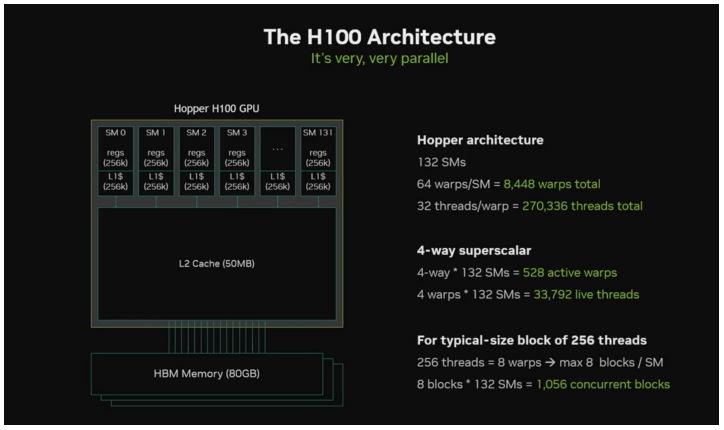
APRIL 2024

Australian National University

Logistics

- Attendance to the Lab sessions is highly encouraged. Most of the practical aspects of the programming models are covered in the Labs.

Introduction to the key concepts of the CUDA Programming Model



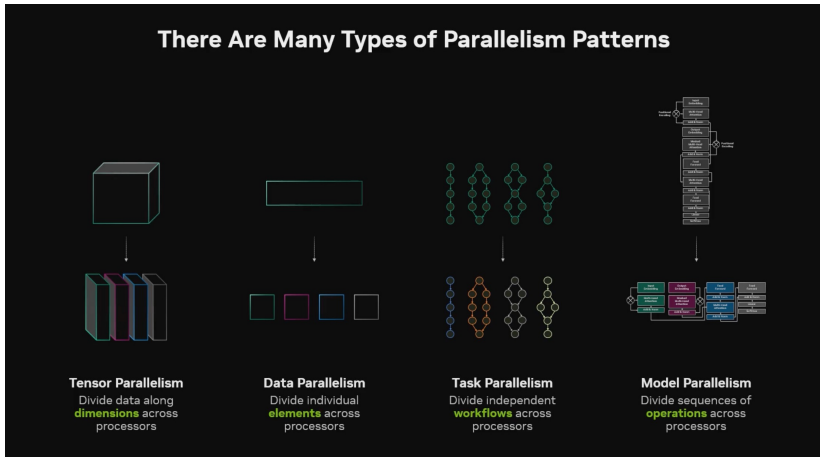
- Programming for the GPU is not an extension of CPU programming
- GPU hardware is changing rapidly, ever more *massive* parallelism
- You need to understand the scale of a problem that a GPU can address

Nvidia slides: Stephen Jones, How To Write A CUDA Program: The Ninja Edition [S62401]

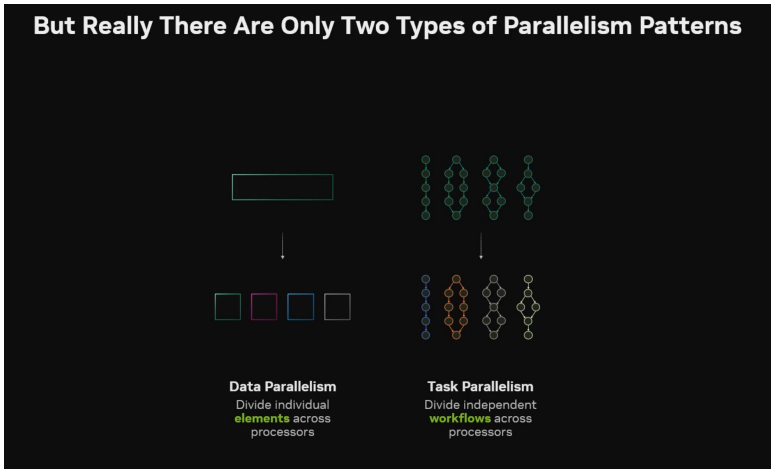
GPU and the CUDA Programming Model

Software	GPU	
Thread	CUDA Core	***Threads*** **are executed by cuda core.** **A** ***warp is 32 threads*** **that are executed simultaneously.**
Thread Block	SM	***Multiple warps can make up a thread block.*** ***Thread blocks*** **are executed on a** ***Streaming Multiprocessor*****.** **Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources**
Grid	Device	***A kernel*** **is launched as a grid of thread blocks**

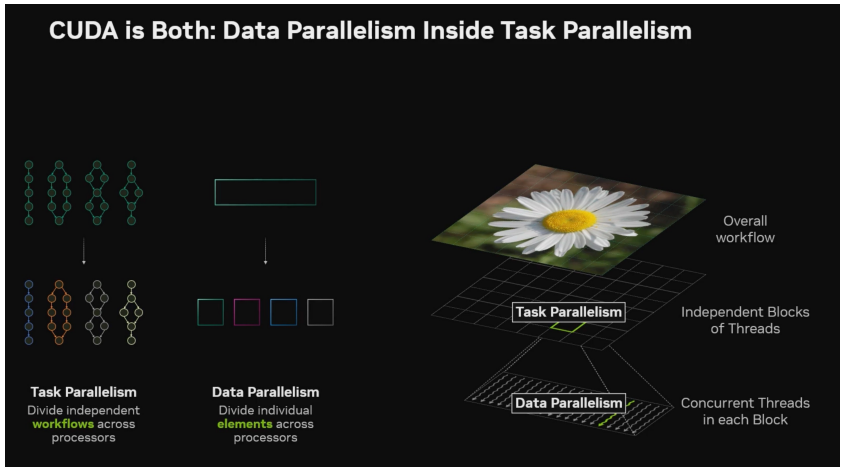
Modified from original source: Maggie Zhang, Nvidia



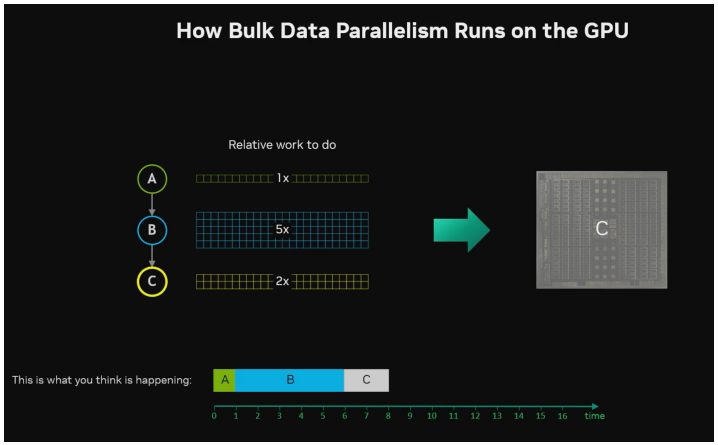
- There are lots of different types of parallelism that are referred to in the literature



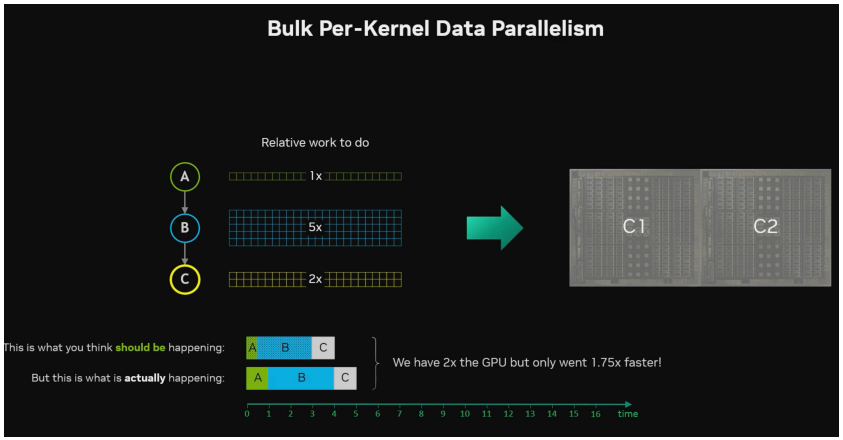
- The reality for the GPU is that there are two fundamental types of parallelism
- Also referred too as fine- and coarse-grained parallelism



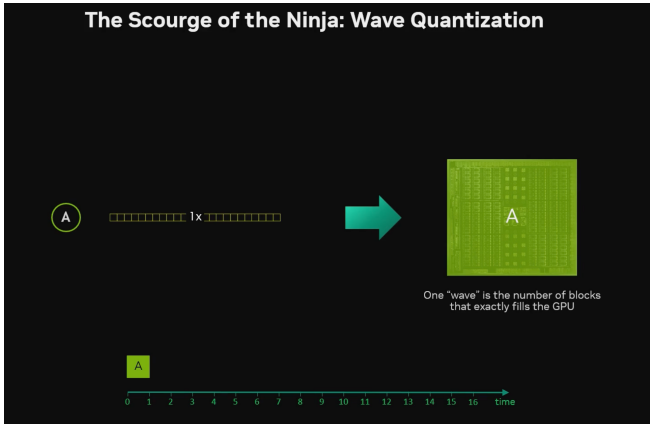
- To achieve high performance on the GPU you need to address both types of parallelism
- If you address only one you will see only a fraction of the possible performance



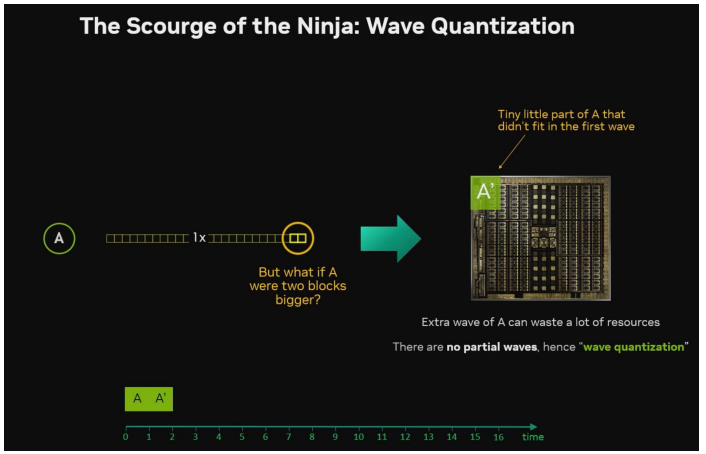
- Task A → Task B → Task C (Task B can only start after Task A completes)
- We must understand how the hardware implements parallelism
- Future lectures will delve in to this in more detail



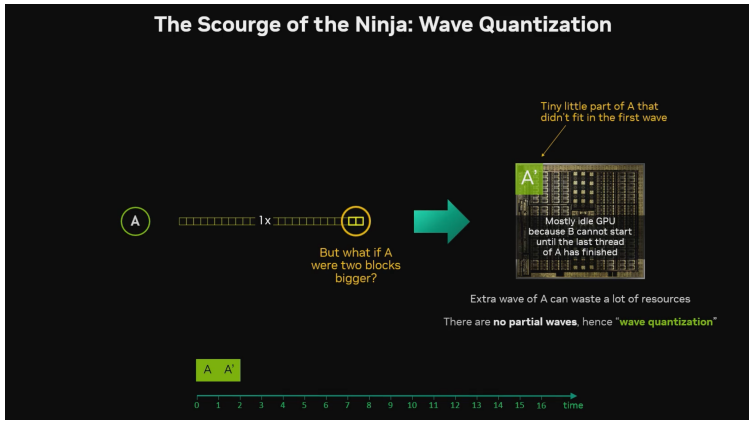
- What happens when we double the size or number of GPUs?
- Task A and the last step of B fit in half the expanded GPU
- For a fixed problem size that fits in one GPU we do not get a 2x gain



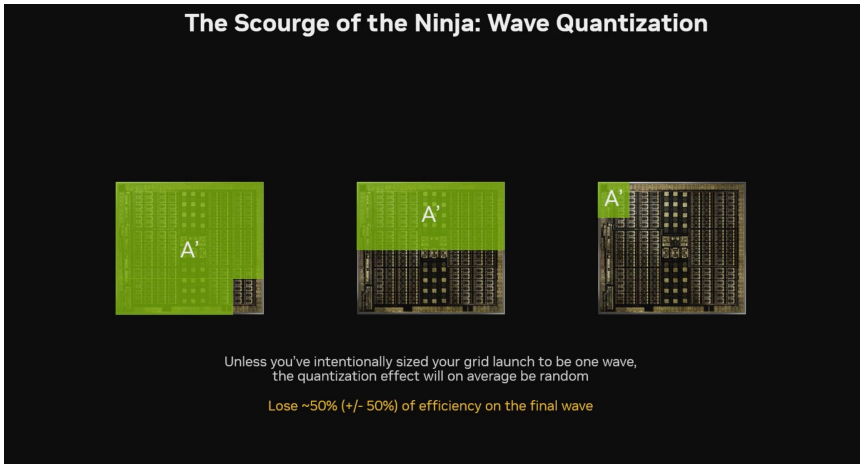
- A Wave is the ideal number of blocks that fills a GPU
- Wave quantization is a key challenge



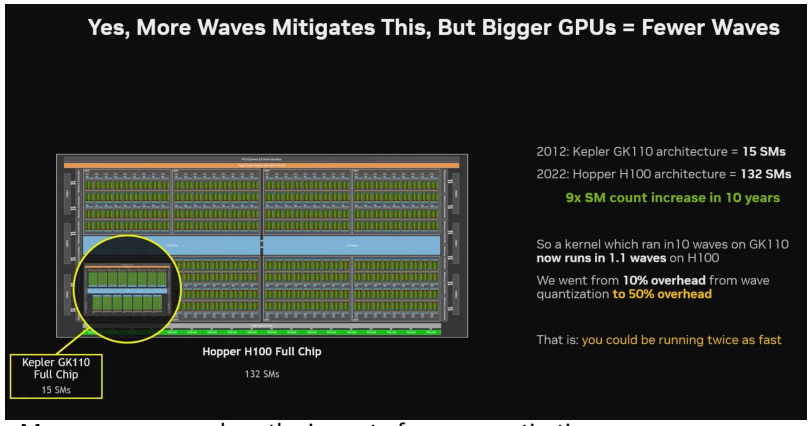
- Here is an extreme example of the problem of wave quantization
- The problem size is just a little larger than a wave
- Lots of resources will be wasted



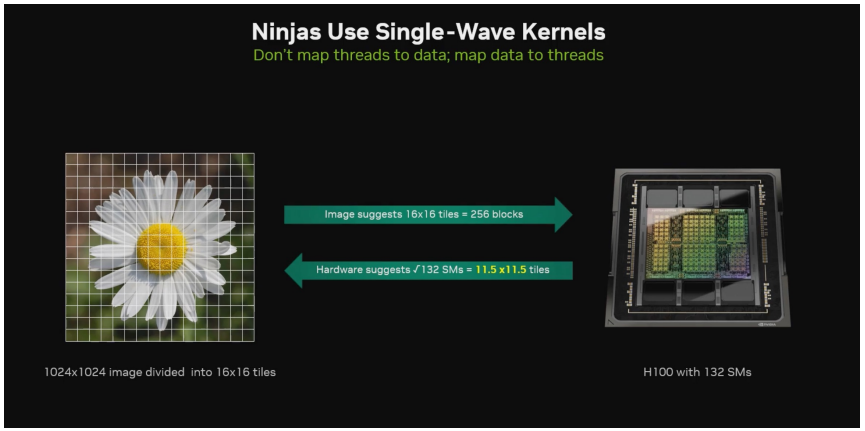
- Here is an extreme example of the problem of wave quantization
- The problem is just a little larger than a wave (2 blocks)
- Most of the GPU will be idle when running A' , Task B cannot start



- Wave quantization Statistics: on average you will lose 50% of the performance
- Without planning, you may lose much more performance



- More waves can reduce the impact of wave quantization –
- original design assumption was for 100 waves
- The dramatic increase in the size of GPUs (the number of SMs) has reduced the number of waves for a fixed workload and increased the overhead



- Natural assumption is to map threads to data
- Correct mapping is the reverse - data to threads
- Divide your tasks across 132 SMs

Ninjas Use Single-Wave Kernels

Don't map threads to data; map data to threads

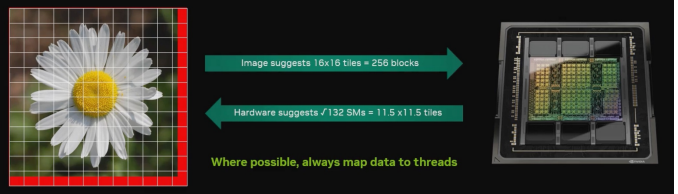


Image suggests 16x16 tiles = 256 blocks

Hardware suggests √132 SMs = 11.5 x11.5 tiles

Where possible, always map data to threads

1024x1024 image divided into **12x12 tiles**

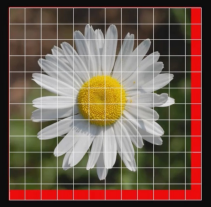
Red edge indicates where naïve rounding leaves imbalanced workload(*)

H100 with 132 SMs

(*) For better load balancing alternate tile sizes of 11 and 12, although this increases complexity and may be impossible depending on application

- The consequence of poor mapping is that we have an imbalanced workload

Not a Silver Bullet



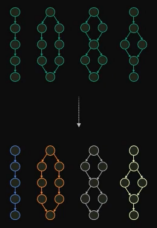
Single-wave kernels are better in almost all cases than non-integer-wave

Similar to "grid-stride loop" pattern frquently taught for CUDA

But there are a number of problems which may prevent use:

1. Some algorithms require specific size of tiling
2. Must account for GPUs of different sizes (e.g. RTX-3090/80/70/60)
3. Increase in code complexity by having non-constant tile size
4. Load imbalance remains; may be no better than an extra partial wave

- The optimal programming approach is to produce single-wave kernels
- This will not always be possible, watch out for load imbalance
- Bulk data parallelism will not typically achieve 100% efficiency



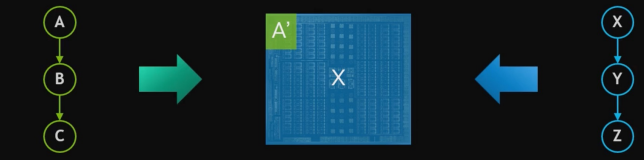
Task Parallelism

Divide independent **workflows** across processors

How about the other kind of parallelism?

- Data parallelism alone will rarely be sufficient to achieve top performance
- Task parallelism will help, but it is harder to implement

So Why Does Task Parallelism Help?

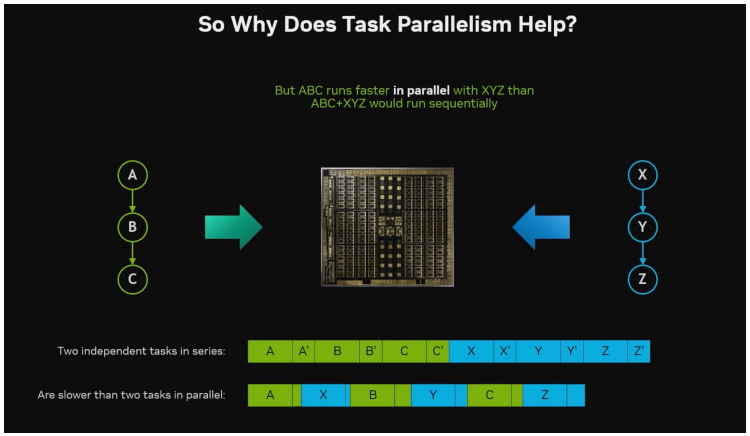


Task 1: A → B → C

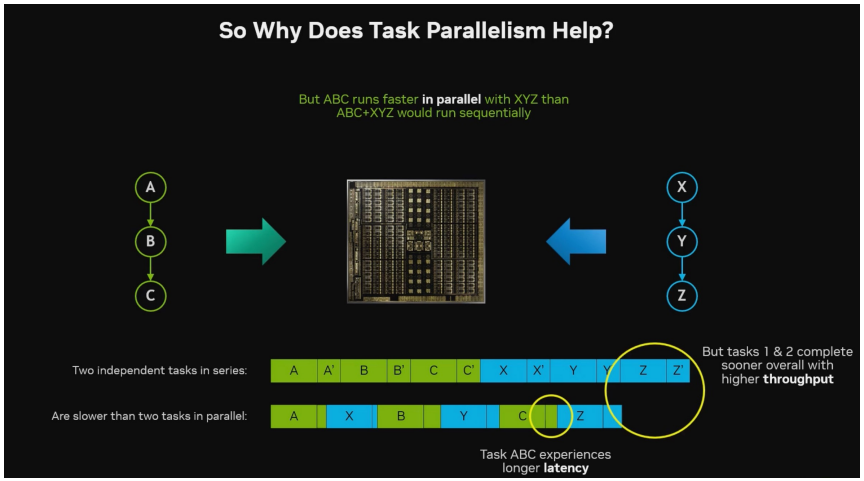
I can't run B, but X is independent of A so I can run X immediately

Task 2: X → Y → Z

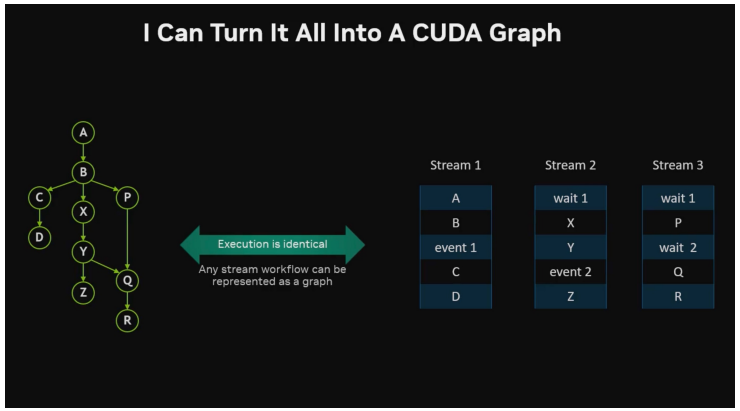
- Task A does not fill the GPU and Task B cannot run until A finishes
- Task X is independent of Task A, so Task X can now fill the GPU



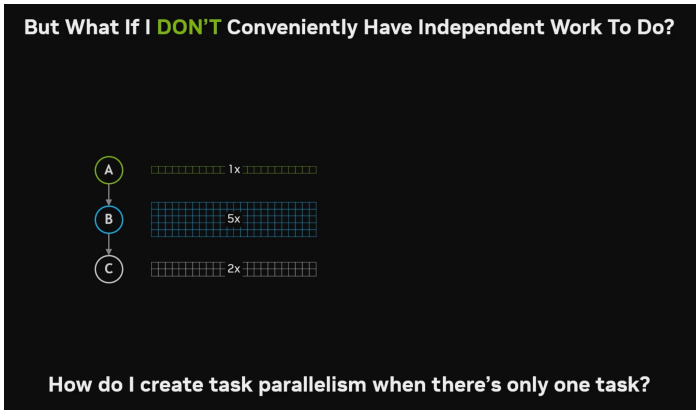
- CUDA streams – concurrent execution
- Stream = A sequence of operations that execute in issue-order on the GPU
- CUDA operations from different streams may be interleaved



- Throughput is faster with task parallelism



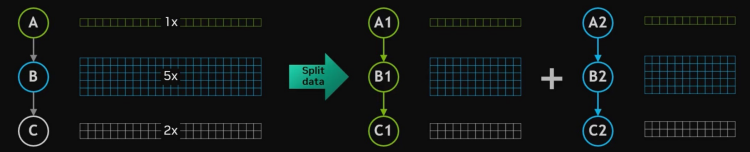
- Complex task parallelism can be represented in a CUDA graph
- A CUDA graph enables multiple GPU operations to be launched through a single CPU operation
- Build and launch CUDA graphs



- Not all problems you may encounter can be divided into multiple independent tasks …

The Obvious Approach: Split The Data In Two

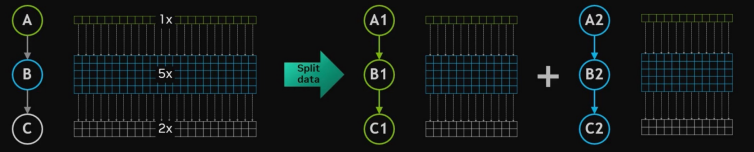
Also known as "Pipeline Parallelism" – yet another type of parallelism which is really just task parallelism



- Pipeline parallelism allows you to create and take advantage of parallel tasks

Easy For Elementwise Programs

Programs never are entirely elementwise, but splitting the kernels which are will always win a little



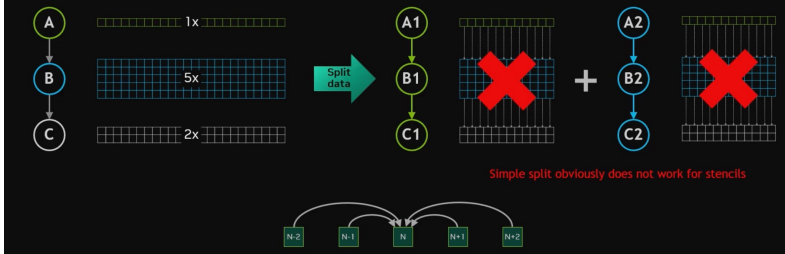

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

Elementwise "ax plus y" vector scale-and-addition

- Always take advantage of elementwise kernels if they are greater than one wave

But The Real World Is Almost Never Elementwise

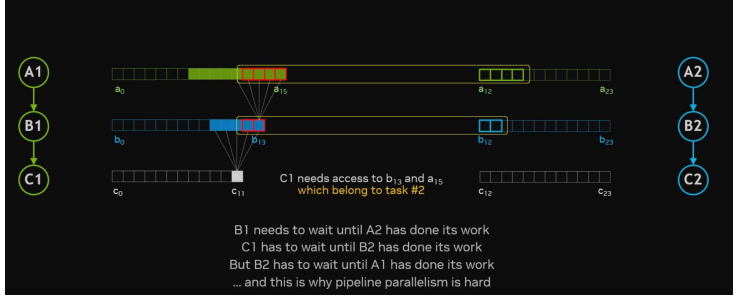
However, it is very often localised, like a stencil, instead of needing random access to all data



- Elementwise operations are a rare opportunity
- Convolutions are an example where surrounding data is required

... Create Dependencies Between Tasks 1 & 2

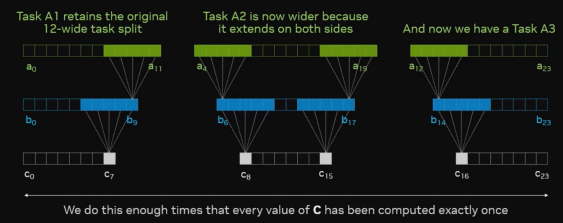
This makes it very hard to co-schedule sub-portions of A, B and C



- Pipeline parallelism by splitting tasks can lead to chained dependencies and undermine any performance gains

So Instead, Reduce the Sizes of B & C

Of course, now you need more than two tasks

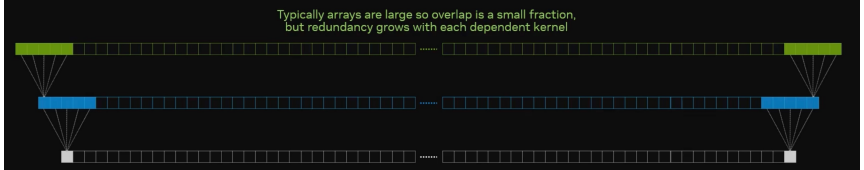


- Avoid the dependencies by reducing the size of dependent tasks (B&C)
- Reducing the size of tasks increases the number of tasks

Cost Of Redundant Computation Depends On Data Size

because it's common that each sub-task A1, A2, A3 operates over many data elements

Often, recomputing something is faster than writing it out to memory and then reading it back in



- This approach also introduces redundant computation at the edges
- The impact of redundant computation can be small as a few % over large arrays

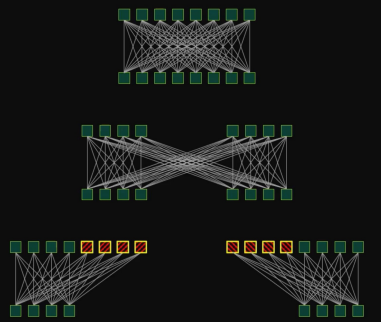
The Real Problem: All-to-All Algorithms

For example: sorting, fourier transforms, and unfortunately many other useful things

- All to all algorithms require extensive communication and synchronization
- Memory usage and bandwidth can limit performance

All-to-All Algorithms Break Pipelining

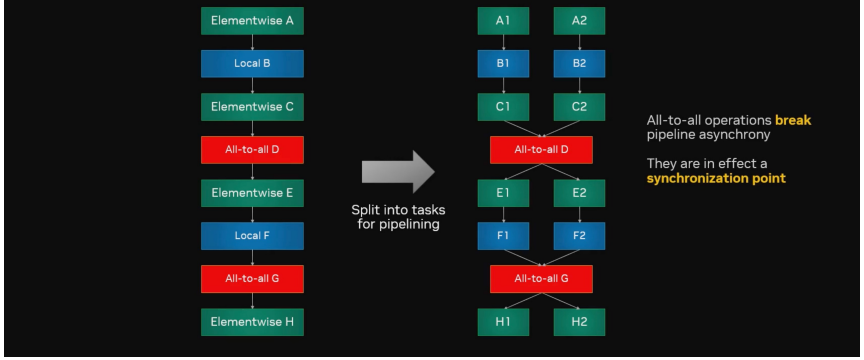
I always end up with 100% redundant computation, so there's no point splitting the operation



- The pipelining solution delivers no benefits in this case
- You will often be working with all-to-all algorithms

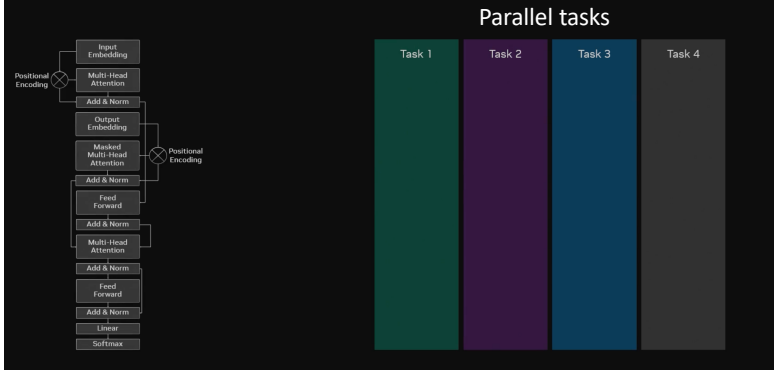
All-to-All Algorithms Break Pipelining

I can run until I hit an all-to-all operation, but then I need to sync across the whole workload



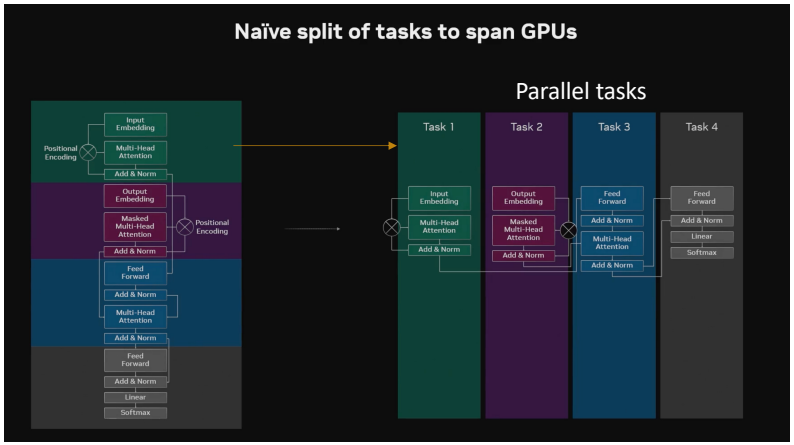
- You may be able to break chunks of your problem into pipelines
- All-to-all will act as a synchronization point

Model Parallelism: Task Parallelism For Complex Workflows



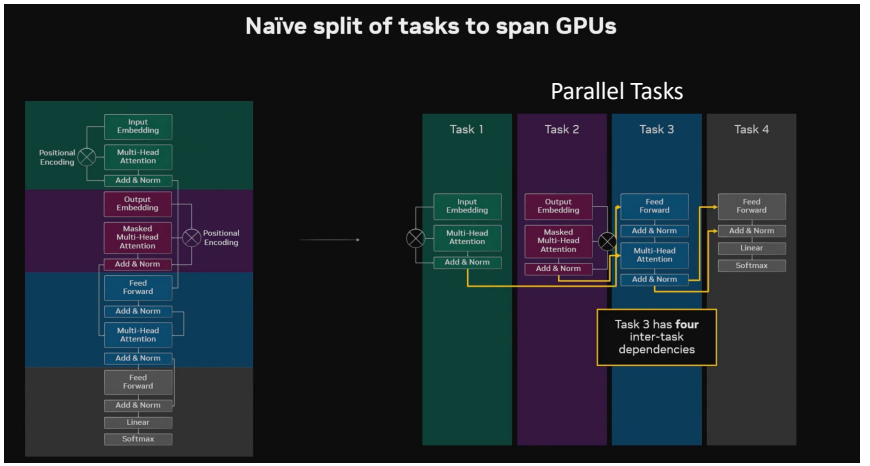
- Model parallelism divides a model into separate tasks
- The example is a multi-layer deep learning model

Naïve split of tasks to span GPUs



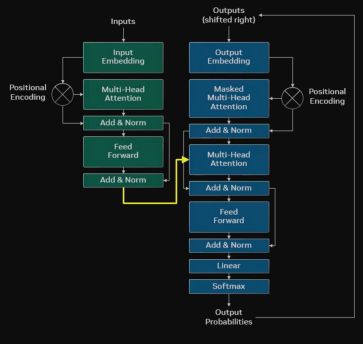
- We can divide the model into seperate parts
- This is a form of task parallelism for complex workflows

Naïve split of tasks to span GPUs



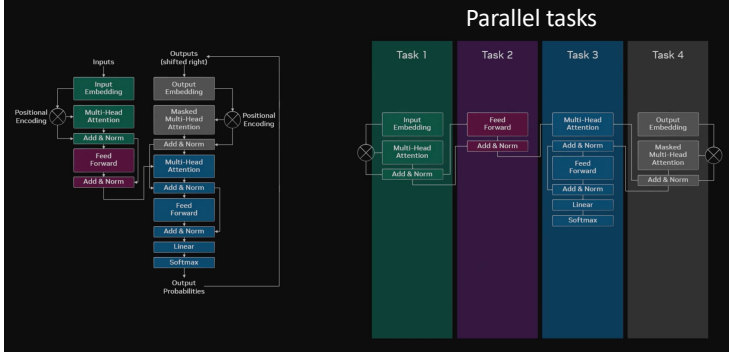
- A simple split may not work well if you ignore dependencies between tasks
- In this example Task 3 will act as a bottleneck

Unbalanced Tasks Are Often Much Cheaper Than Over-Synchronization



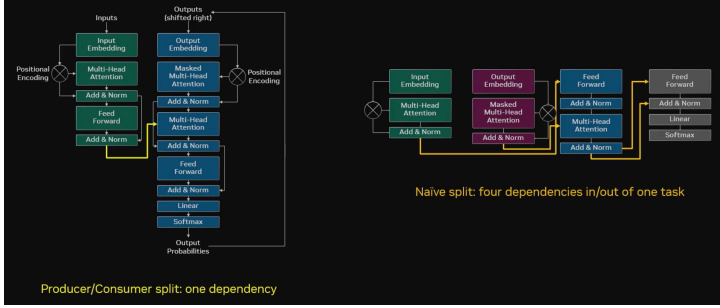
- Reducing synchronization can be more efficient that attempting to balance the task workload

And If You Need To Split More, Do It Hierarchically



- Once you have identified a split that minimizes synchronization you can then further split based on that hierarchy

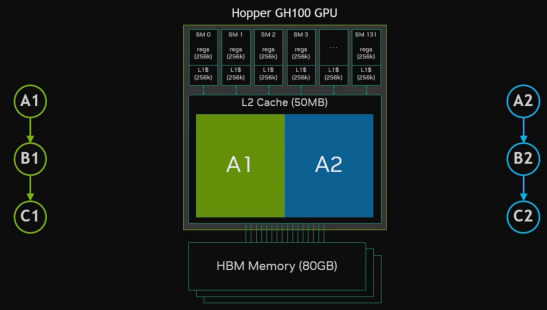
In Other Words, Split The Model To Minimise Inter-Task Sync



- A key goal when implementing model parallelism is to minimize inter-task synchronization ie Reduce waiting time and keep the GPU busy
- This applies to task parallelism in general

A Big Problem With Task Parallelism: Thrashing The Cache

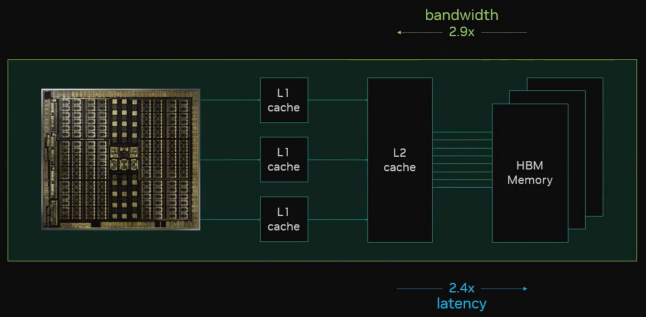
Each task sees a proportionately smaller cache because by definition they are working on different data



- As with many compute architectures, GPU memory is a critical resource
- The more tasks the less cache available, the more cache misses that undermine efficiency

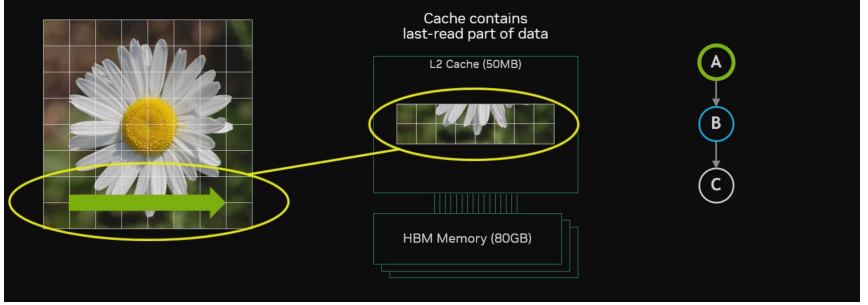
You really really want cached data

A high L2 hit rate can give as much as a 10x boost in performance



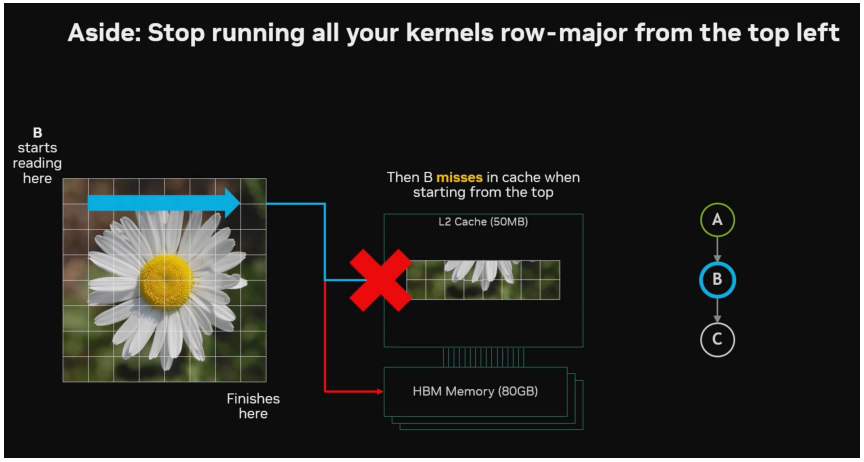
- A high cache hit rate produces the highest performing code
- L2 cache has higher bandwidth and lower latency than HBM memory

Aside: Stop running all your kernels row-major from the top left



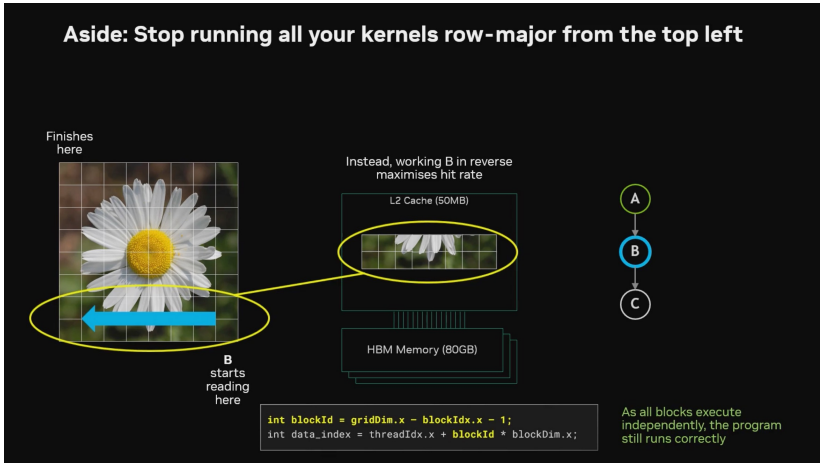
- Row-major finishes at the bottom
- Task B will start again at the top left ….

Aside: Stop running all your kernels row-major from the top left

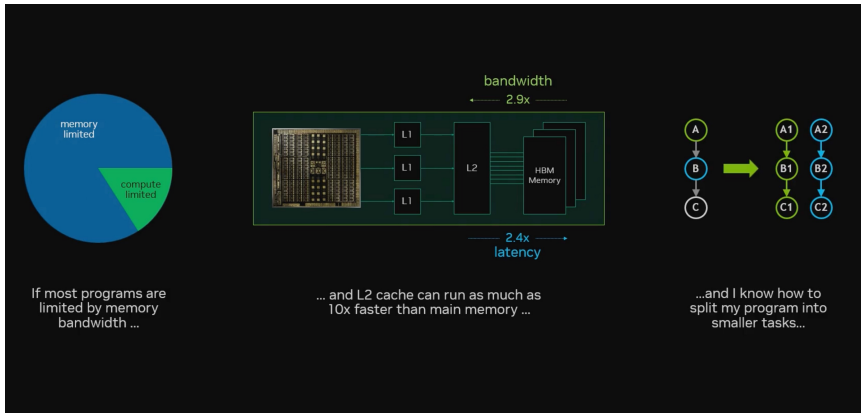


- When switching to Task B you will always generate a cache-miss with row-major kernels

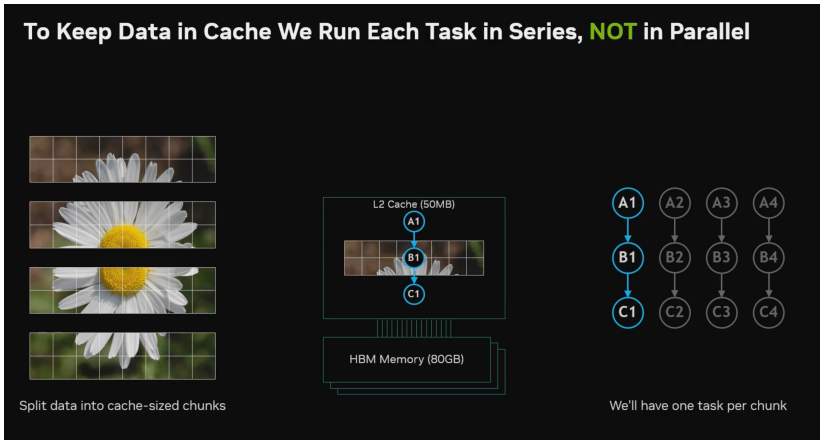
Aside: Stop running all your kernels row-major from the top left



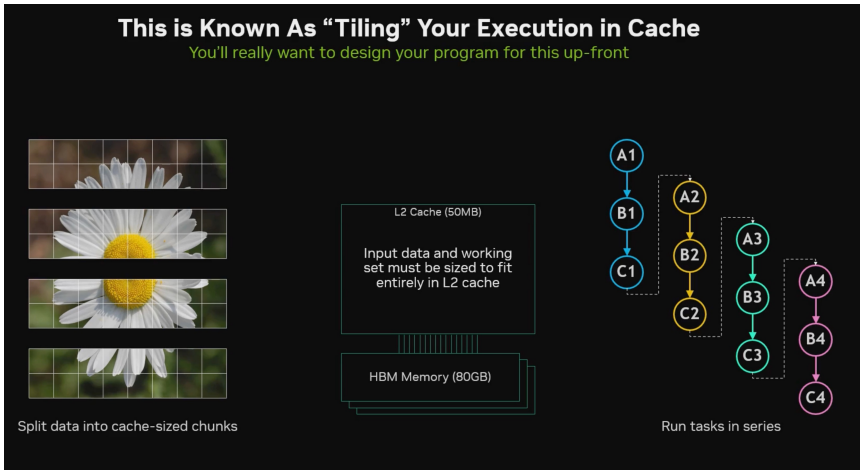
- Running with B in reverse order will produce a cache hit (~10x faster)
- Managing cache effectively can deliver significant benefits



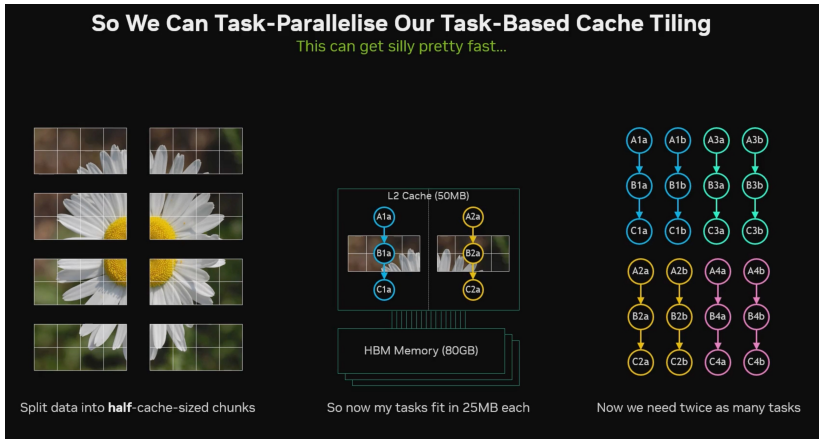
- Identifying whether your program is bandwidth limited is essential to producing high performing code
- For most problems that you encounter this will be the case
- Can we run our problem in L2 cache?



- Previous examples demonstrated how to split problems into smaller tasks
- Split the tasks into L2 cache-sized chunks
- Run each Task in series on the cache size chunk!



- Running tasks in series is known as tiling e.g tile-based graphics rendering
- Choosing the optimal tiling size is crucial for achieving good performance



- Can take advantage of both task and data parallelism, all running in cache
- Programming complexity increases
- Design from the start, refactoring to achieve this most likely will be hard

The grid stride loop pattern in CUDA

- The **grid stride loop pattern** is a technique used in CUDA programming to ensure that a kernel can efficiently process data arrays of any size

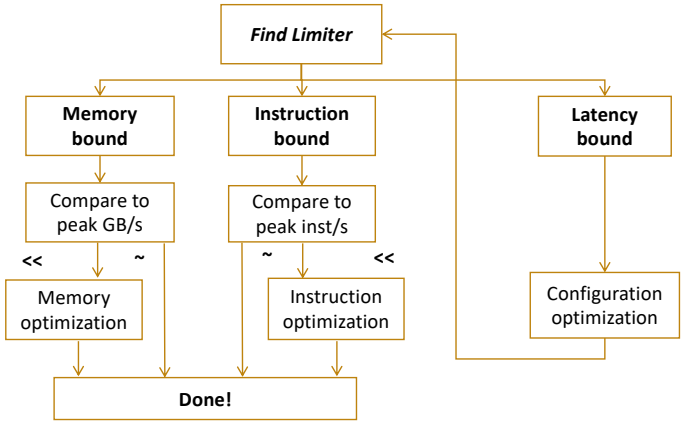
```
__global__ void  saxpy(int  n, float  a, float *x, float *y) {
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; i += blockDim.x * gridDim.x) {
        y[i] =  a * x[i] + y[i];
    }
}
```

- In this example, each thread calculates its unique index in the array (`i = blockIdx.x * blockDim.x + threadIdx.x`), and then processes the element at that index.
- The thread then increments its index by the total number of threads in the grid (`blockDim.x * gridDim.x`), and processes the next element, repeating this process until all elements have been processed
- This pattern allows the kernel to handle data arrays of any size, even when the number of threads launched is less than the number of data elements.
- It also makes your CUDA kernels more flexible and scalable

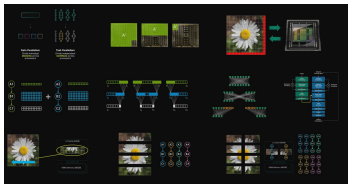
Optimization Workflow in CUDA

Wrong View of Optimization!

- Try all the optimization methods in the book
- …optimization is endless…



Summary



- Programming a *streaming multiprocessor* is not an extension of CPU programming!
- Is a GPU required based on the scale of the task and the ability to expose parallelism?
- Data and task parallelism concepts are the GPU fundamentals that you should master
- Seek to achieve wave quantization on the target GPU
- Task parallelism
- All-to-all algorithms break task parallelism, use higher level model parallelism
- Create a CUDA graph of complex model parallelism tasks, reduce dependencies
- Avoid bandwidth limitations, tile execution in cache