北京大学
PEKING UNIVERSITY

智能硬件体系结构

第十讲：多核多线程与静态优化（aka.编译）

主讲：陶耀宇、李萌

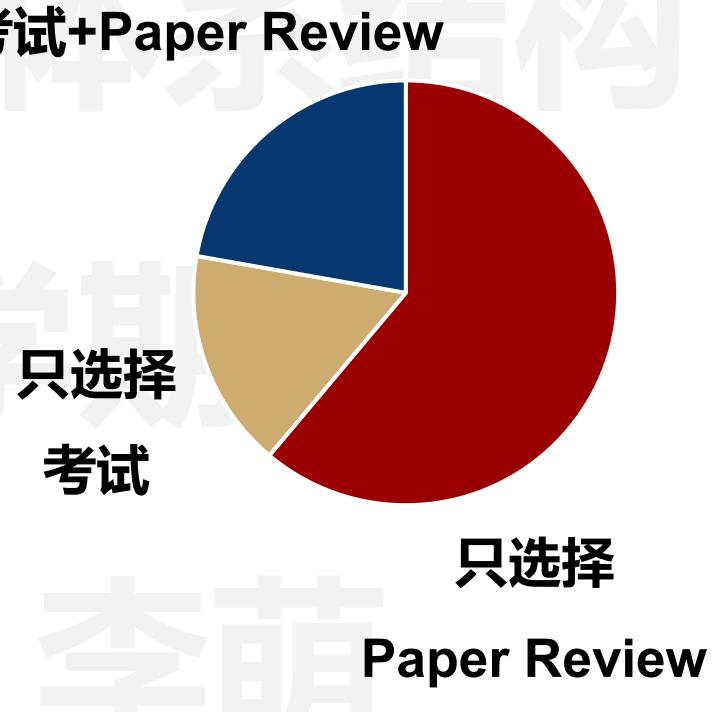
2024年秋季

注意事项

北京大学 PEKING UNIVERSITY

• 课程作业情况

- 第3次作业答案将于本周放出
- 第4次作业**12月2日**截止（缓存设计）
- 未来**2次作业（12.1-12.15、12.15-12.30）**
  - AI芯片架构、软硬加协同优化
- **Take Home考试**
  - **推迟到11月30日00:00 – 12月1日11:59（周六、日）**
  - 每迟交一天扣10分
- **Paper Review（12月25号/27号1-3点，每个同学10分钟）**
  - 时间有疑义请尽快联系老师或助教提出更换
  - 请自行阅读近5年内的体系结构相关论文1-2篇
  - 做8页左右的PPT，**每个同学10分钟演讲 + 1~2分钟答疑**
- 第二次实验将在**12月1号**放出，截止**12月31号**

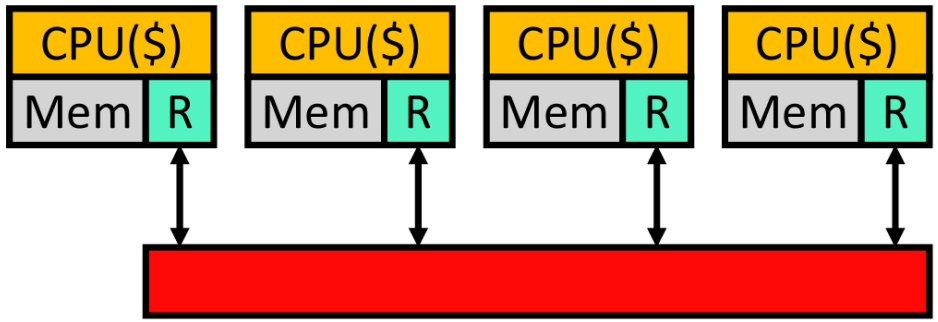


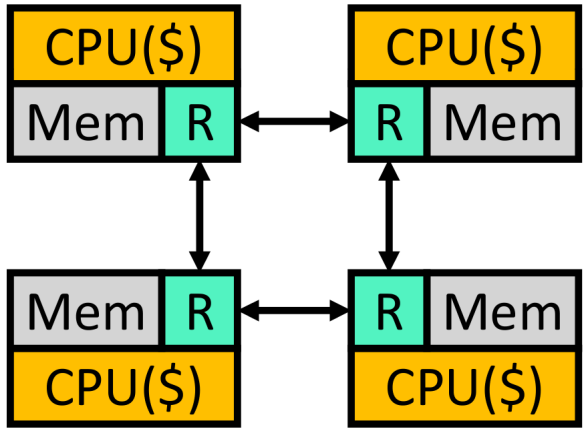
目 录

CONTENTS

多核架构的组织形式

- **Bus-based Multi-core 和 Network-on-chip：Chip Multiprocessor (CMP)**



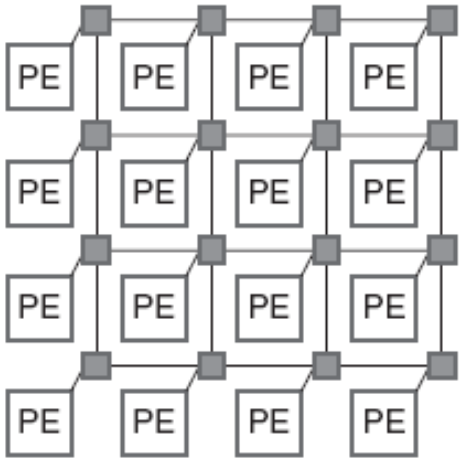


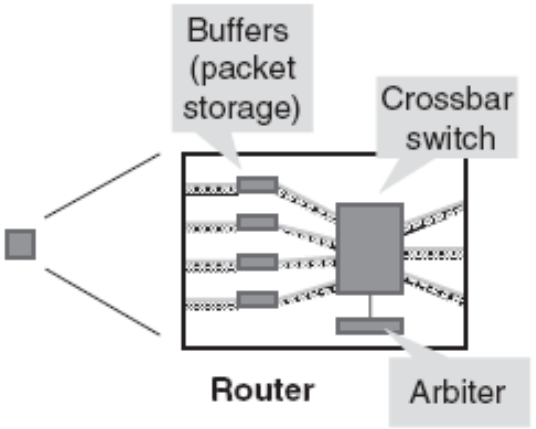
- Shared network: e.g., bus
  - Low latency
  - Low bandwidth: doesn’ t scale beyond ~16 processors
  - Shared property simplifies cach e coherence protocols (later)

- Point-to-point network: e.g., mesh or ring
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are complex

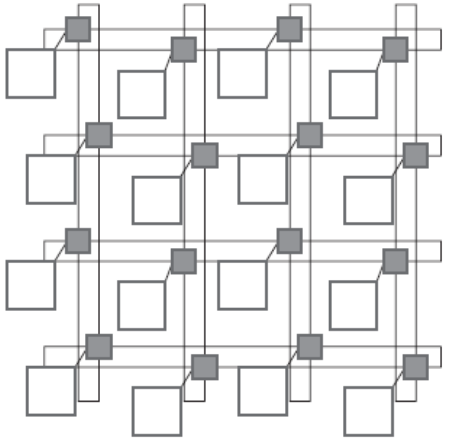
多核架构的组织形式

- Network-on-chip：Chip Multiprocessor (CMP)

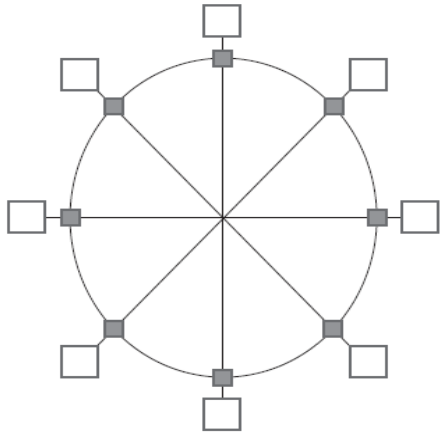




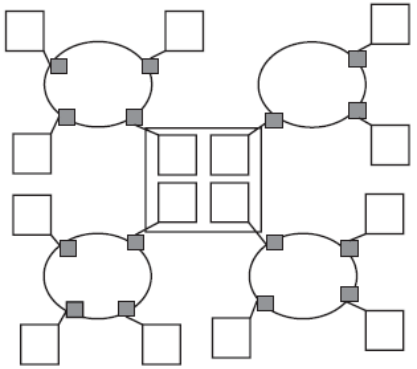
Mesh Topology



2-D torus topology



Octagon topology



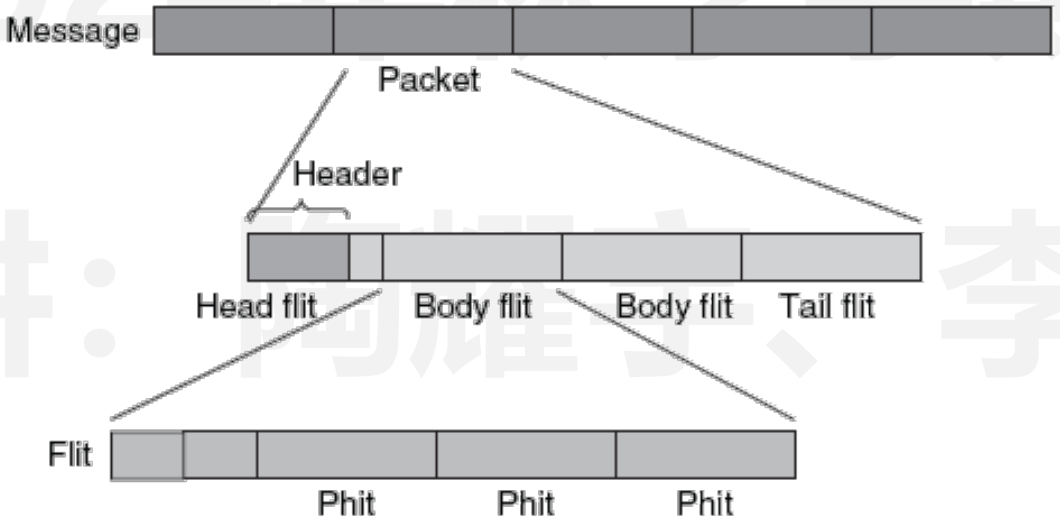
Irregular or ad hoc network topologies

多核架构的组织形式

- **Network-on-chip：Chip Multiprocessor (CMP)**

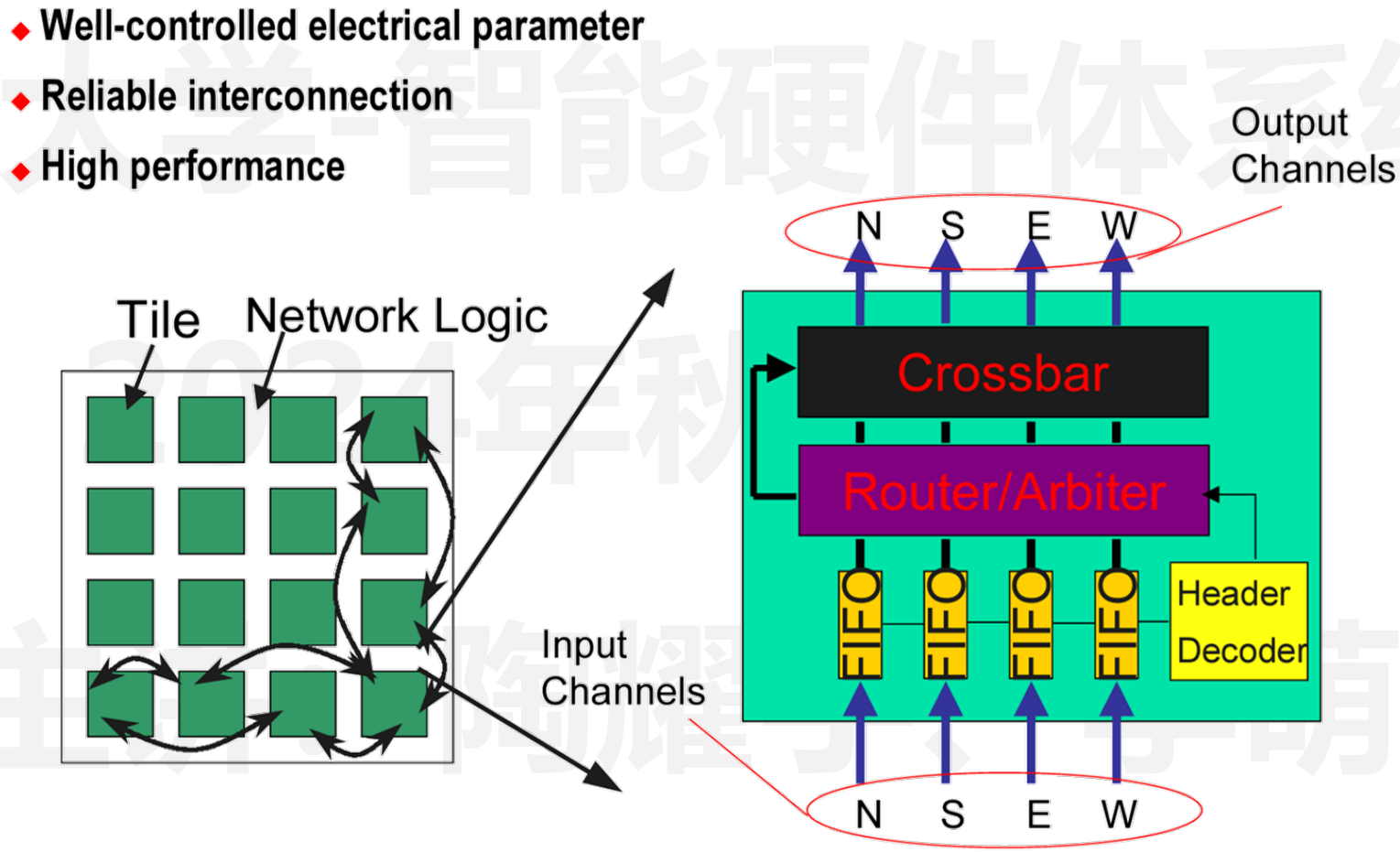
Switching strategies

- Determine how data flows through routers in the network
- Define granularity of data transfer and applied switching technique
  - phit is a unit of data that is transferred on a link in a single cycle
  - typically, phit size = flit size



多核架构的组织形式

- Network-on-chip：Chip Multiprocessor (CMP)



多核架构的组织形式

北京大学 PEKING UNIVERSITY

- **Network-on-chip：Chip Multiprocessor (CMP)**

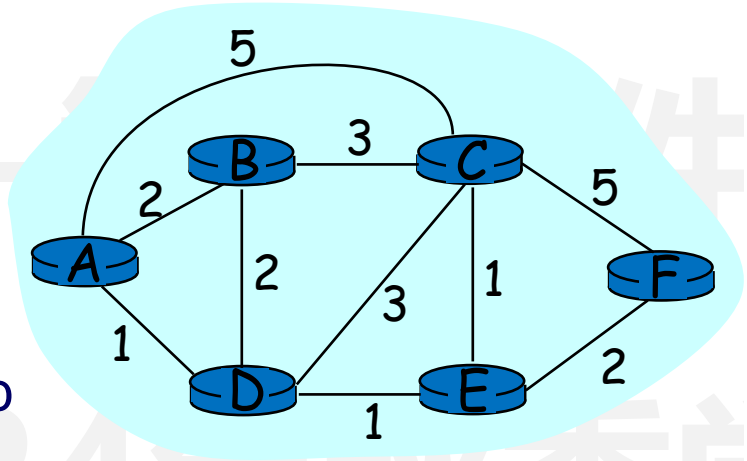
- Static and dynamic routing
  - **static routing**: fixed paths are used to transfer data between a particular source and destination
    - does not take into account current state of the network
  - advantages of static routing:
    - easy to implement, since very little additional router logic is required
    - in-order packet delivery if single path is used
  - **dynamic routing**: routing decisions are made according to the current state of the network
    - considering factors such as availability and load on links
  - path between source and destination may change over time
    - as traffic conditions and requirements of the application change
  - more resources needed to monitor state of the network and dynamically change routing paths
  - able to better distribute traffic in a network

多核架构的组织形式

- **Static Routing Tables**

Dijkstra's algorithm

- net topology, link costs known to all nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node ('source") to all other nodes
  - gives routing table for that node
- iterative: after k iterations, know least cost path to k dest.'s



CENTRAL ROUTING DIRECTORY

From Node

To Node	1	2	3	4	5	6
1	—	1	5	2	4	5
2	2	—	5	2	4	5
3	4	3	—	5	3	5
4	4	4	5	—	4	5
5	4	4	5	5	—	5
6	4	4	5	5	6	—

Node 1 Directory

Destination	Next Node
2	2
3	4
4	4
5	4
6	4

Node 2 Directory

Destination	Next Node
1	1
3	3
4	4
5	4
6	4

Node 3 Directory

Destination	Next Node
1	5
2	5
4	5
5	5
6	5

Node 4 Directory

Destination	Next Node
1	2
2	2
3	5
5	5
6	5

Node 5 Directory

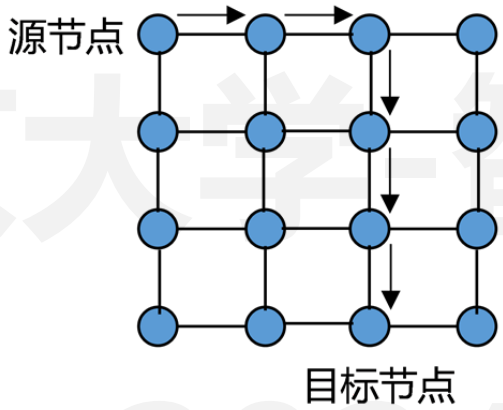
Destination	Next Node
1	4
2	4
3	3
4	4
6	6

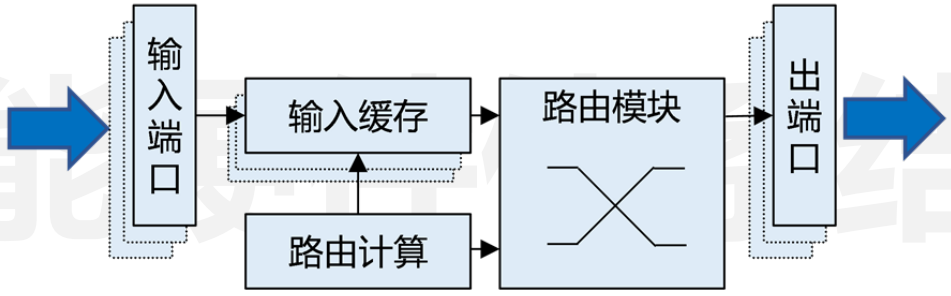
Node 6 Directory

Destination	Next Node
1	5
2	5
3	5
4	5
5	5

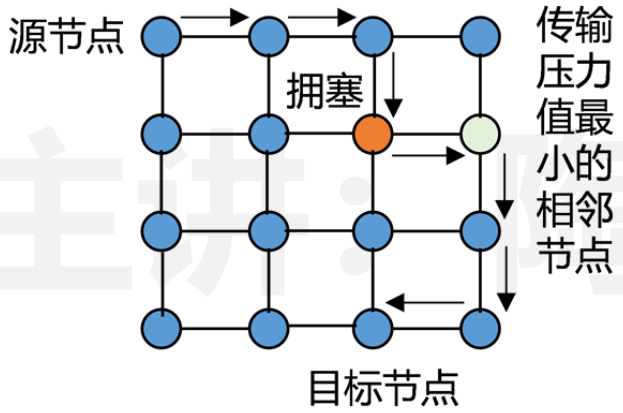
多核架构的组织形式

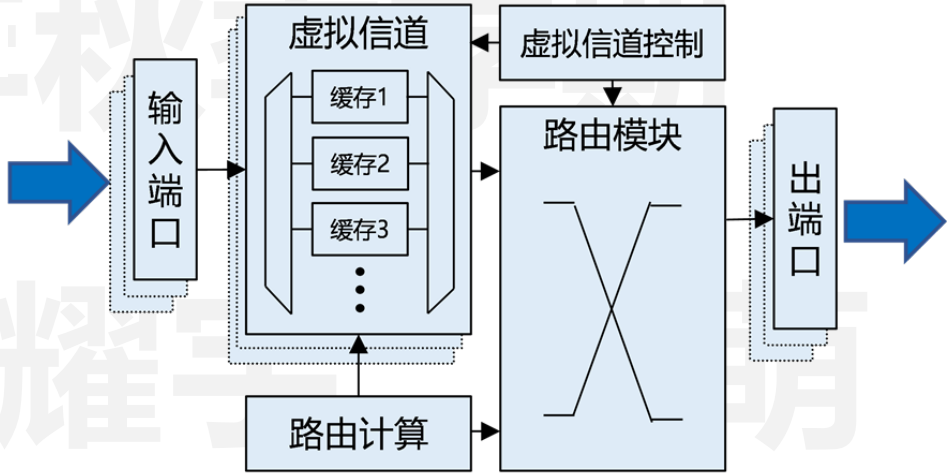
- Dynamic Routing





传统路由模块

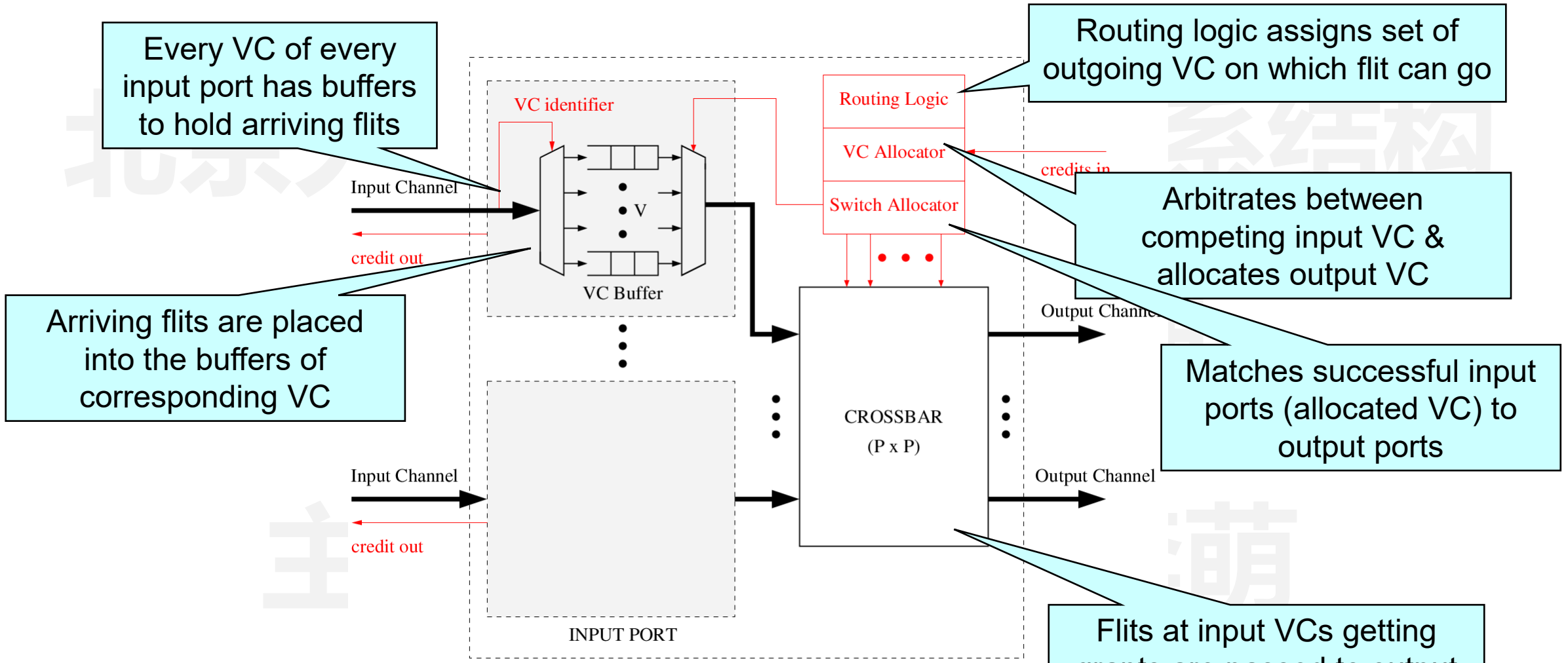




虚拟信道路由模块

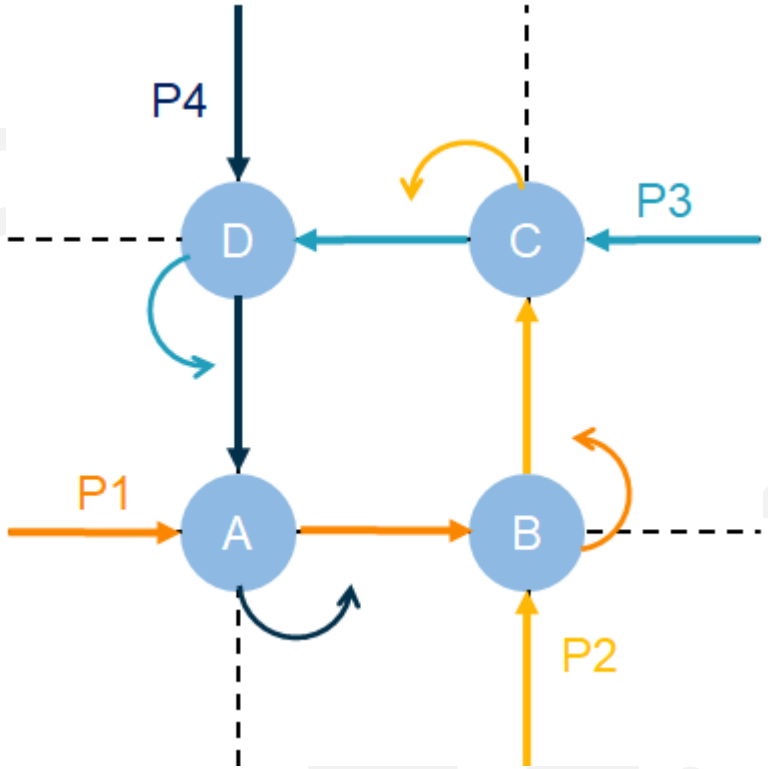
多核架构的组织形式

- 虚拟信道路由



多核架构的组织形式

- **路由死锁Deadlock**

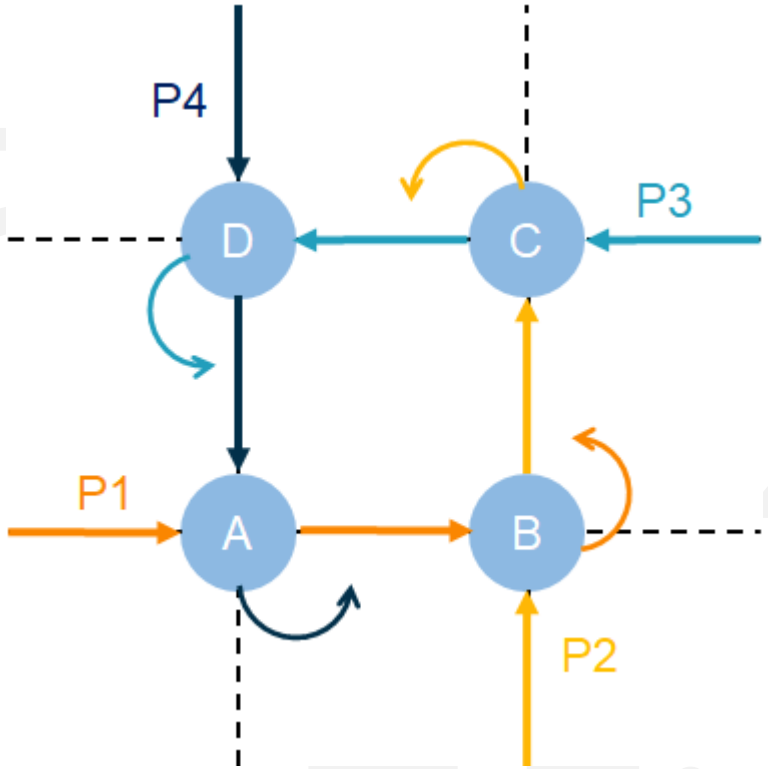


假设路由规则要求全路径资源空闲才可发送数据包

- 死锁是指多个数据包或任务在网络中争夺资源导致的路由间的永久阻塞，从而导致数据或消息任务在没有外界干扰的情况下永远无法转发到目的地。
- P1、P2、P3和P4四个数据包依次占用了一条路径资源，并请求额外的资源，但请求的资源又被相邻数据包占据
- 以P1和P2数据包为例，P1 数据包从路由器A发往B并最终期望达到路由器C
- P2 数据包从路由器B发往C并最终期望达到路由器D。
- P1、P2分别占用AB、BC链路，P1请求的BC链路目前没有得到释放，原因在于P2请求的CD链路没有得到释放，导致P1、P2数据包无法进行后续转发。因循环的数据路径以及相互的资源占用导致四个数据包产生死锁

多核架构的组织形式

- 解决路由死锁Deadlock的方法



假设路由规则要求全路径资源空闲才可发送数据包

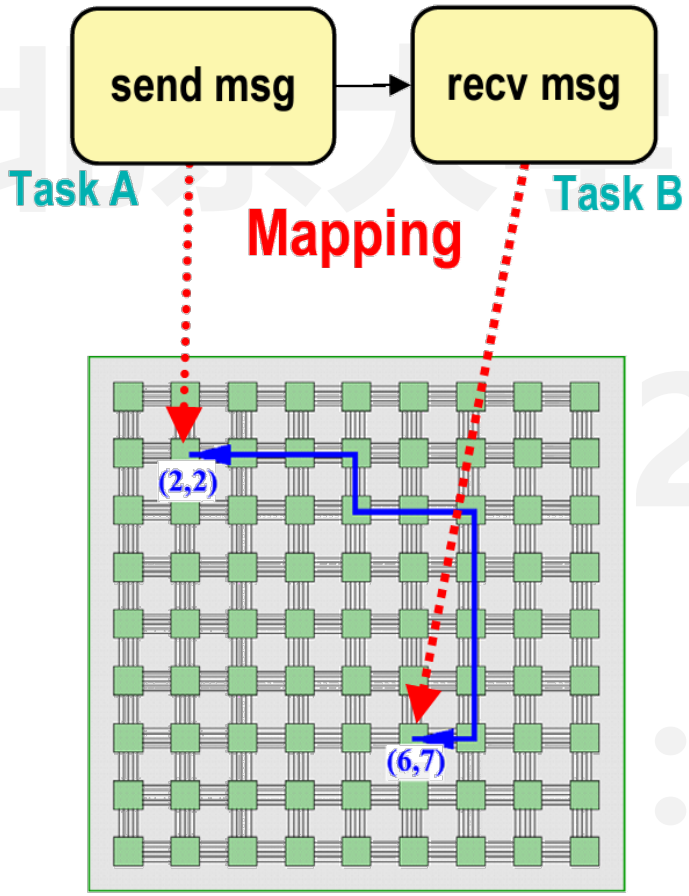
1.提供更多资源：该方法对应解决死锁条件中的互斥，主要采用两种策略：增加虚拟通道和消费通道。前者通过在路由端口增加额外缓存，避免单一通道资源占用导致的死锁，但是并不增加额外的物理传输通道；后者与前者相反，通过增加物理传输通道，从而在转发过程中提供更多资源

2.避免数据环路：该方法对应解决死锁条件中的数据环路，主要通过设计合理的路由算法，来避免环路的发生

3.取消任务对资源的占用：该方法对应解决死锁条件中的资源占用，即取消阻塞数据包对通道的占用请求。一般考虑一个确定的等待时间，如果数据包在路由缓存中长期未转发，则网络考虑出现死锁，取消其占用请求，并利用额外的死锁缓存通道来进行缓存，使得路由资源重新得到释放

多核架构的组织形式

- Workload Mapping



- **Mapping issues:**
  - Assigning tasks to resources
  - Translating task addresses to resource/task addresses
  - Establishing and closing channels
  - Static allocation versus dynamic allocation
- **System software: NoC OS**
  - OS, RTOS, run-time scheduler
  - Component and network dependent
  - Application program

多核架构的组织形式

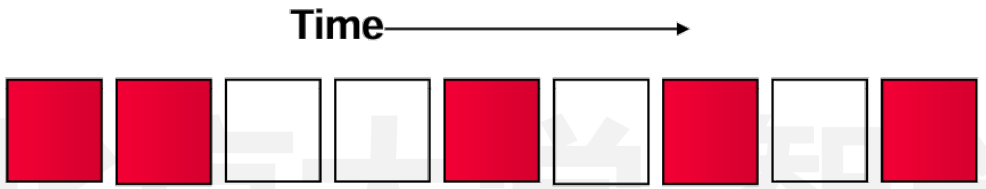
北京大学 PEKING UNIVERSITY

- NoC设计需要注意的问题

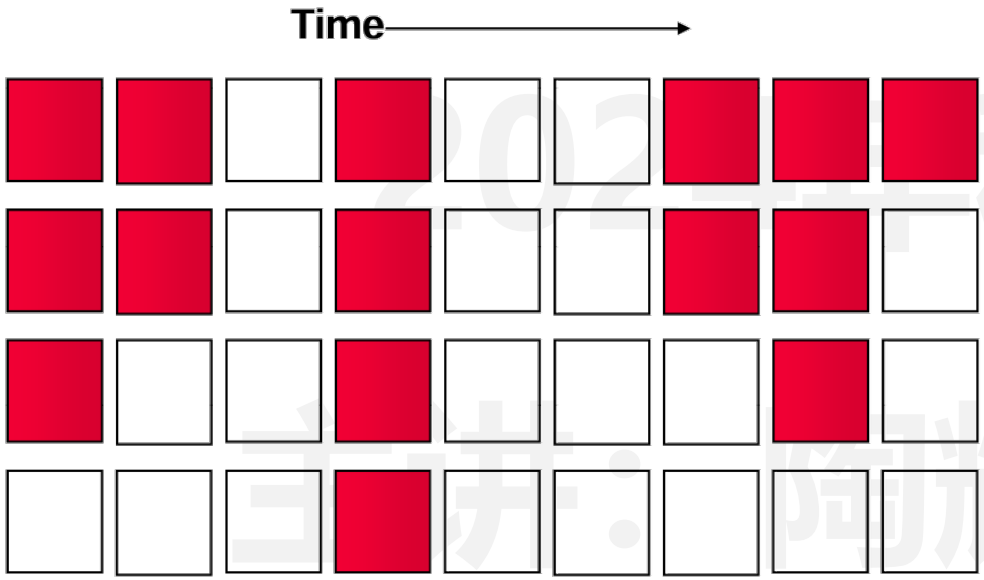
- **Scaling problem**
  - How big NoC is needed? What are the application area requirements?
- **Region definition problem**
  - What kind of regions are needed? What kind of interfaces between regions?
  - What are the capacity requirements for the regions?
- **Resource design problem**
  - What is needed inside resources? Internal computation type and internal communication?
- **Application mapping flow problem**
  - What kind of languages, models and tools must be supported?
  - How to validate and test the final products?

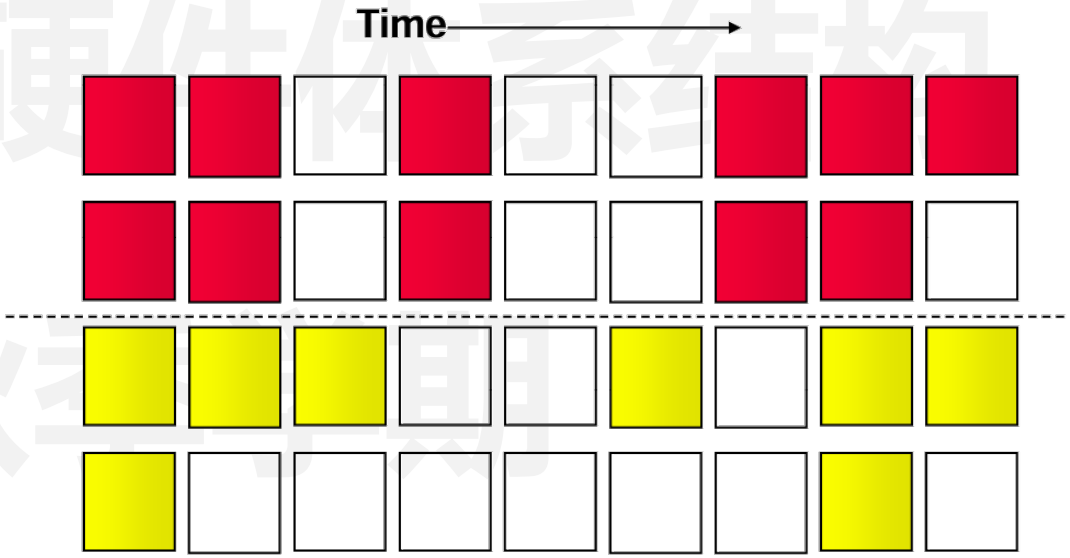
多线程组织形式

- Instruction Issue



Scalar Pipeline



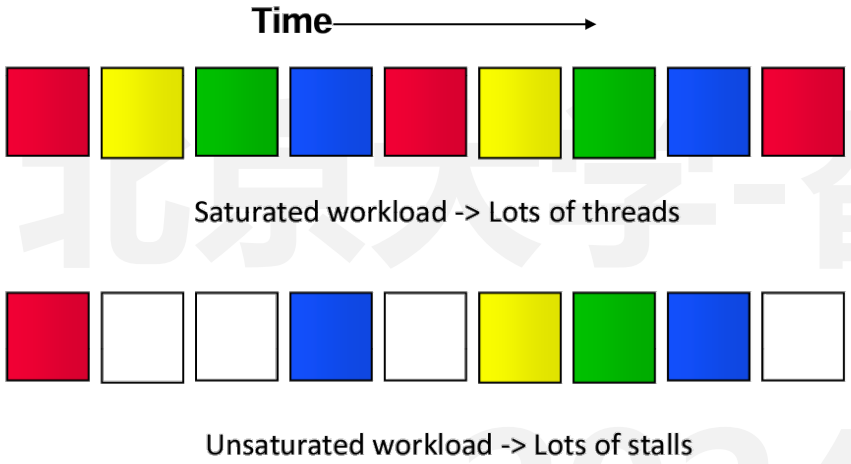


Limited utilization when only running one thread

Superscalar leads to more performance, but lower utilization

多线程组织形式

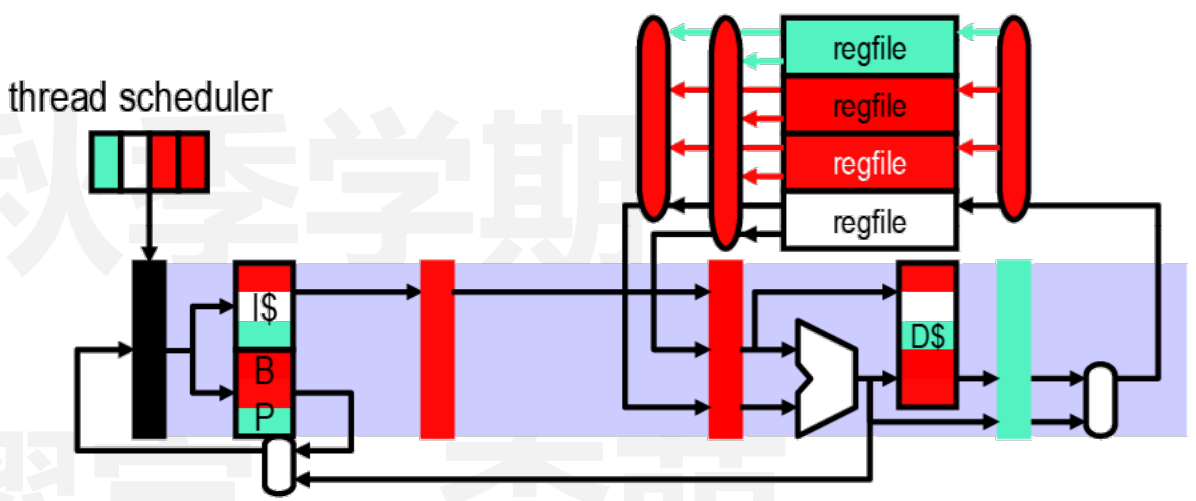
- **Fine Grained Multithreading (FGMT)**



- FGMT
  - (Many) more threads
  - Multiple threads in pipeline at once

Intra-thread dependencies still limit performance

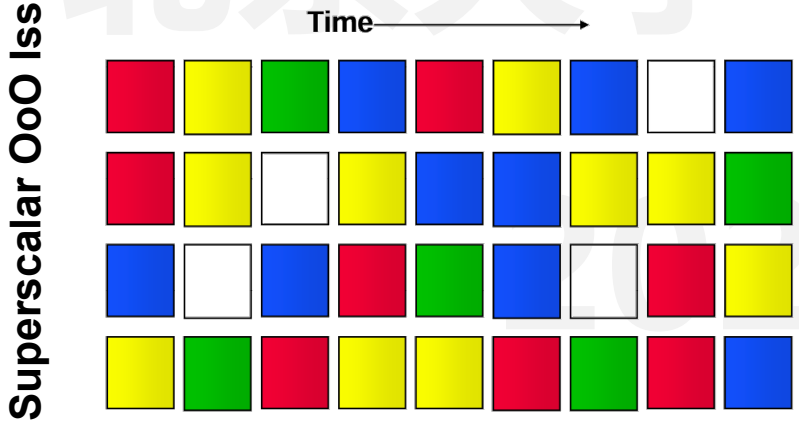
- 每个Thread（线程）个线程都可以单独执行一个运行任务
- 线程包含于进程当中，进程是线程的集合
- 当程序运行时，至少有一个进程会启动，而这个进程中往往包含了多个线程



Many threads → many register files

多线程组织形式

- Simultaneous Multithreading (SMT)

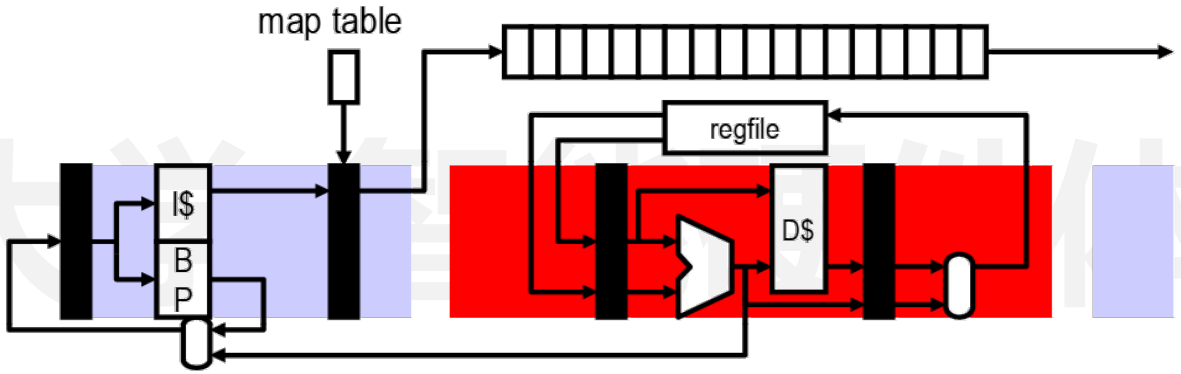


Maximum utilization of function units by independent operations

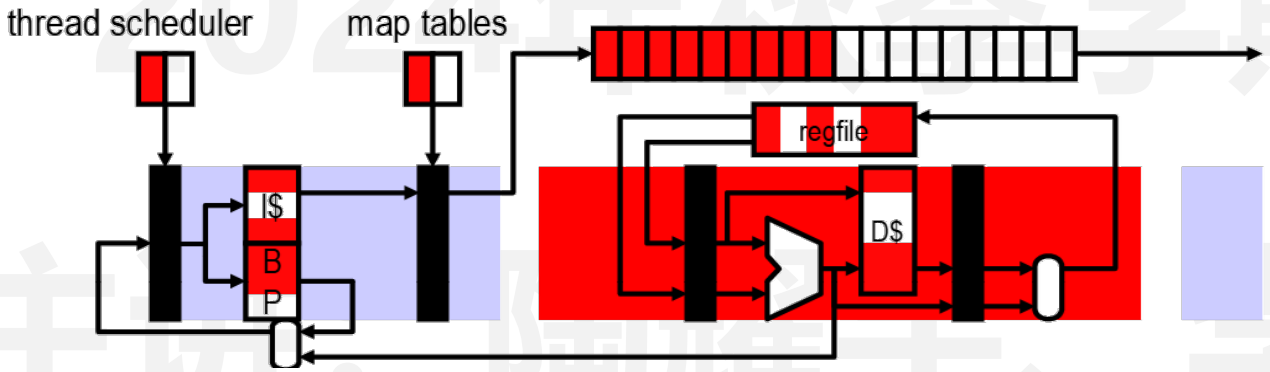
- Can we multithread an out-of-order machine?
  - Don't want to give up performance benefits
  - Don't want to give up natural tolerance of D$ (L1) miss latency
- **Simultaneous multithreading (SMT)**
  - Tolerates all latencies (e.g., L2 misses, mispredicted branches)
  - Sacrifices some single thread performance
  - Thread scheduling policy
    - Round-robin (just like FGMT)
  - Pipeline partitioning
    - Dynamic, hmmm…
  - Example: Pentium4 (hyper-threading): 5-way issue, 2 threads
  - Another example: Alpha 21464: 8-way issue, 4 threads (canceled)

多线程组织形式

- **Simultaneous Multithreading (SMT)**



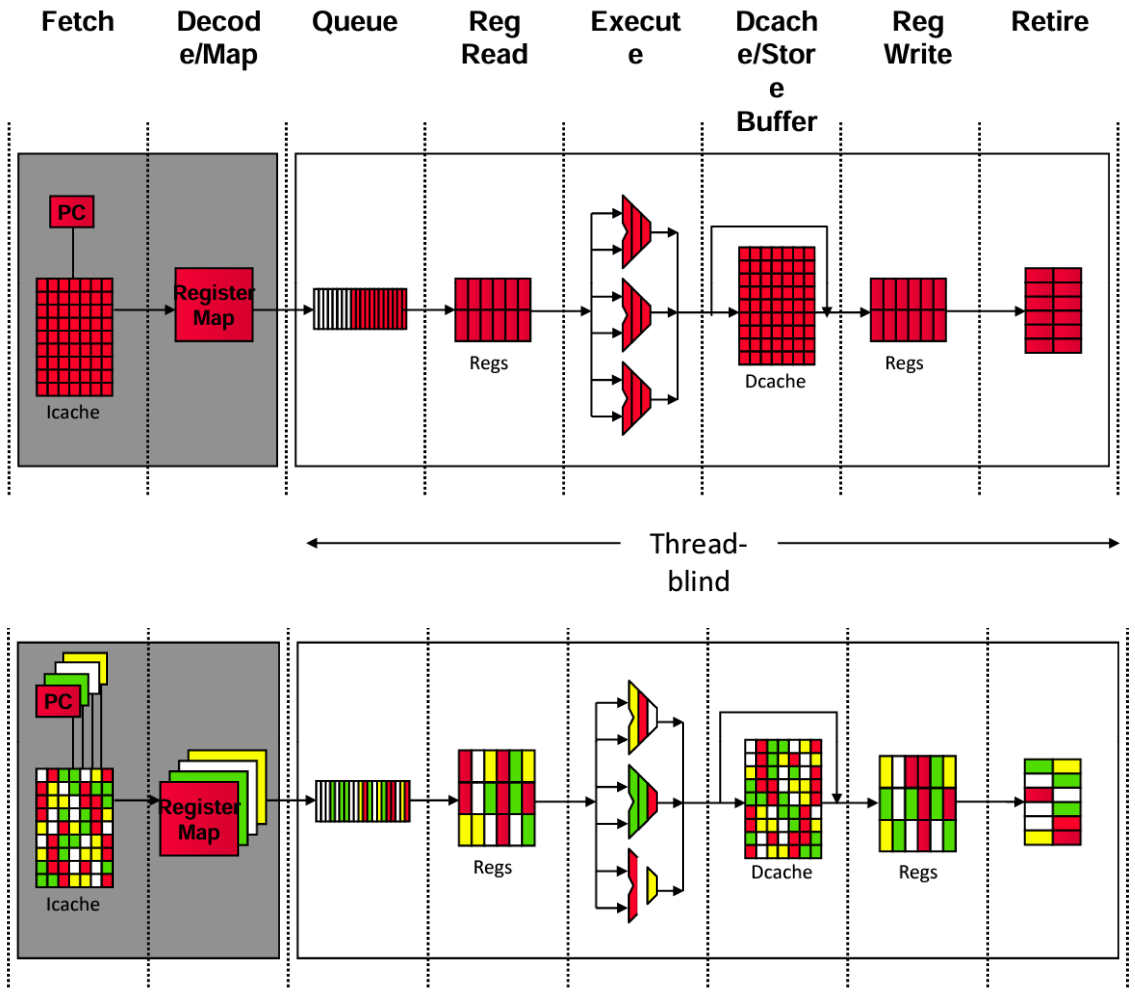
- SMT
  - Replicate map table, share physical register file



- Large map table and physical register file
  - #mt-entries = (#threads * #arch-regs)
  - #phys-regs = (#threads * #arch-regs) + #in-flight insns

多线程组织形式

- Simultaneous Multithreading (SMT) Pipeline



SMT Changes

Basic pipeline – unchanged

Replicated resources
- Program counters
- Register maps

Shared resources
- Register file (size increased)
- Instruction queue
- First and second level caches
- Translation buffers
- Branch predictor

多线程组织形式

北京大学 PEKING UNIVERSITY

- **Simultaneous Multithreading (SMT) vs Chip Multiprocessor (CMP)**
  - If you wanted to run multiple threads would you build a…
    - Chip multiprocessor (CMP): multiple separate pipelines?
    - A multithreaded processor (SMT): a single larger pipeline?
  - **Both will get you throughput on multiple threads**
    - CMP will be simpler, possibly faster clock
    - SMT will get you better performance (IPC) on a single thread
      - SMT is basically an ILP engine that converts TLP to ILP
      - CMP is mainly a TLP engine
  - **Again, do both**
    - Sun's Niagara (UltraSPARC T1)
    - 8 processors, each with 4-threads (coarse-grained threading)
    - 1Ghz clock, in-order, short pipeline (6 stages or so)
    - Designed for power-efficient "throughput computing"

目 录

CONTENTS

静态优化（编译）

- **截止目前，课程介绍了各类的硬件优化计算速度的方法**

- Spent a lot of time learning about dynamic optimizations
  - Finding ways to improve ILP in hardware
    - Out-of-order execution
    - Branch prediction
- But what can be done statically (at compile time)?
  - As hardware architects it behooves us to understand this.
    - Partly so we are aware what things software is likely to be better at.
    - But partly so we can find ways to find hardware/software “synergy”

静态优化（编译）

- Improve locality of data
- Remove instructions that aren’ t needed
- Reduce number of branches executed
- Many others

静态优化（编译）

- 提升局部性

- Examples:
  - **Loop interchange**—flip inner and outer loops
  - **Loop fission**—split into multiple loops

```
for i from 0 to 10
  for j from 0 to 20
    a[j,i] = i + j

for j from 0 to 20
  for i from 0 to 10
    a[j,i] = i + j
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
  a[i] = 1;
}
for (i = 0; i < 100; i++) {
  b[i] = 2;
}
```

静态优化（编译）

- **Improve locality of data**
- Remove instructions that aren’t needed
- Reduce number of branches executed
- Many others

静态优化（编译）

北京大学 PEKING UNIVERSITY

- 优化不需要的指令

- **Register optimization**
  - Registers are fast, and doing “spills and fills” is slow.
  - So keep the data likely to be used next in registers.

- **Common sub-expression elimination**
  - `(a + b) - (a + b)/4`
    - Just compute a+b once.
- **Constant folding**
  - Replace `(3+5)` with **8**.

- **Loop invariant code motion**
  - Move recomputed statements outside of the loop.

```
for (int i=0; i<n; i++) {
        x = y+z;
        a[i] = 6*i+x*x;
}
```

```
x = y+z;
for (int i=0; i<n; i++) {
        a[i] = 6*i+x*x;
}
```

静态优化（编译）

- Improve locality of data
- **Remove instructions that aren’t needed**
- Reduce number of branches executed
- Many others

静态优化（编译）

- 减少Branch次数

- **Using predicates or CMOVs instead of short branches**
- **Loop unrolling**

```
for(i=0;i<10000;i++)
{
   A[i]=B[i]+C[i];
 }

for(i=0;i<10000;i=i+2)
{
   A[i]=B[i]+C[i];
   A[i+1]=B[i+1]+C[i+1];
 }
```

静态优化（编译）

- Improve locality of data
- Remove instructions that aren’t needed
- **Reduce number of branches executed**
- Many others

静态优化（编译）

- 其他优化方法

- “Hoist” loads
  - That is move the loads up so if there is a miss we can hide that latency.
    - Very similar goal to our OoO processor.

```
xxxxx
xxxxx
LD R1=MEM[x]
R2=R1+R3
```

```
LD R1=MEM[x]
xxxxx
xxxxx
R2=R1+R3
```

静态优化（编译）

- Improve locality of data
- Remove instructions that aren’t needed
- Reduce number of branches executed
- **Many others**

静态优化（编译）

- 其他优化方法

Static dependency checking

- **A superscalar processor has to do certain dependency checking at issue (or dispatch)**
  - Is a given set of instructions dependent on each other?
  - If ALU resources are shared are there enough resources?
- Many of these issues can be resolved at compile time.
  - What can’ t be resolved?
  - Once resolved, how do you tell the CPU?

静态优化（编译）

- Improve locality of data
- Remove instructions that aren’ t needed
- Reduce number of branches executed
- **Many others**

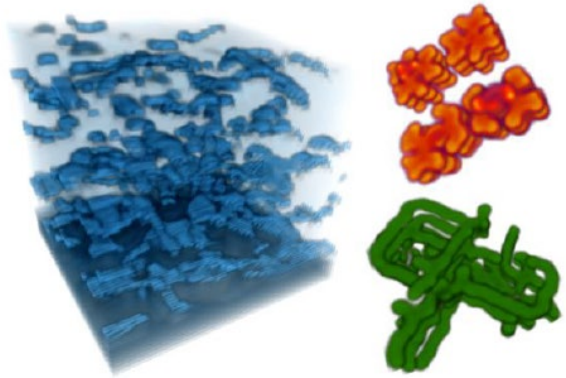
目 录

CONTENTS

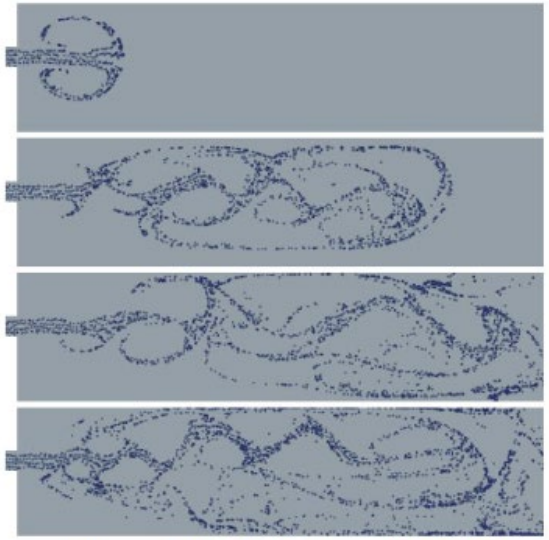
GPU: Overview

- Application

Great diversity of materials and lights in the world!



Coupled Map Lattice Simulation [Harris 02]

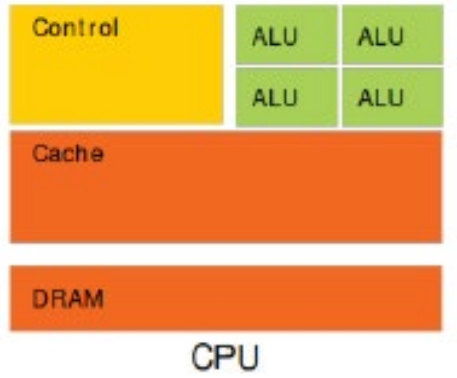


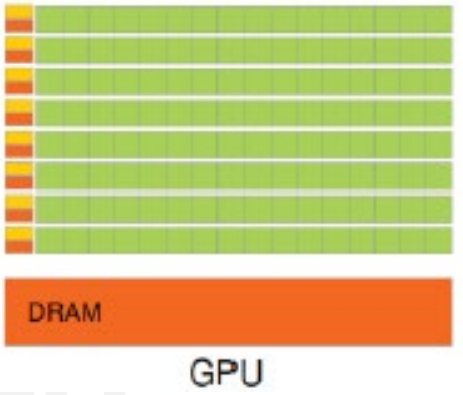
Sparse Matrix Solvers [Bolz 03]

GPU: Overview

- **Highly Parallel Coprocessor**

  - GPU特点
    - 有其专有的DRAM内存
    - 多个简单计算核心
    - 非常小的Cache
    - 并行运行多个线程
  - GPU Threads
    - GPU Threads十分轻量（几乎不需要creation/context switch）
    - 为了达到full efficiency，GPU需要至少几千个threads





GPU: Overview

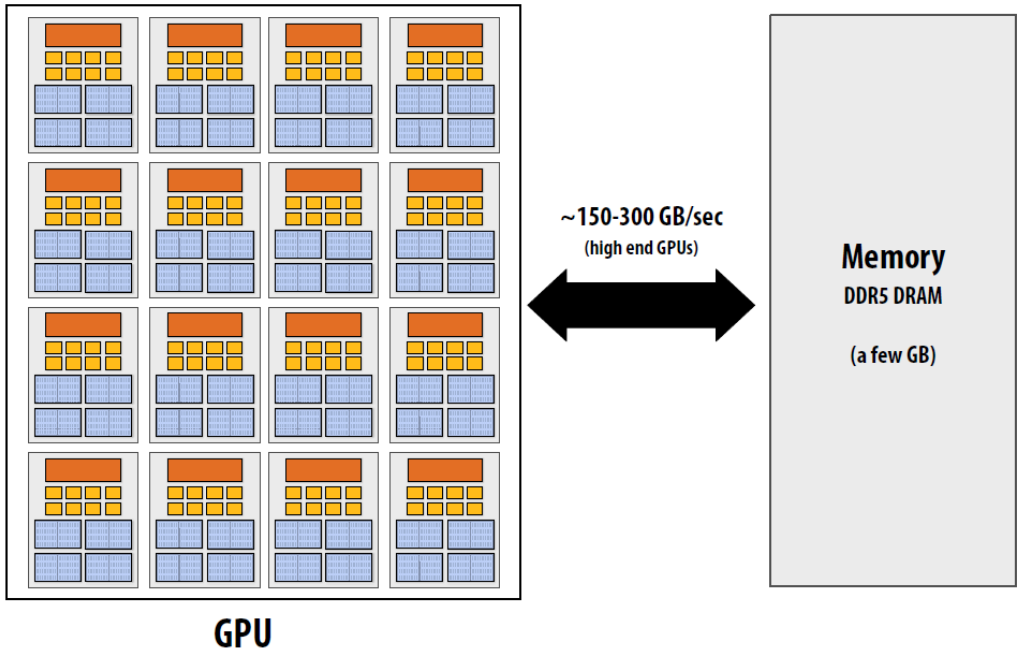
北京大学 PEKING UNIVERSITY

- What is GPU Good at

  - **GPU is good at data-parallel processing**
    - 对多个data element并行进行相同的计算操作——low control flow overhead
  - **High SP floating point arithmetic intensity**
    - Many calculations per memory access
    - 更多针对浮点数的运算而非整形计算
  - 更高的浮点数计算intensity和更多的data element意味着**memory access的延迟和计算的延时相比可以忽略不计**（不需要大的cache）

GPU: Overview

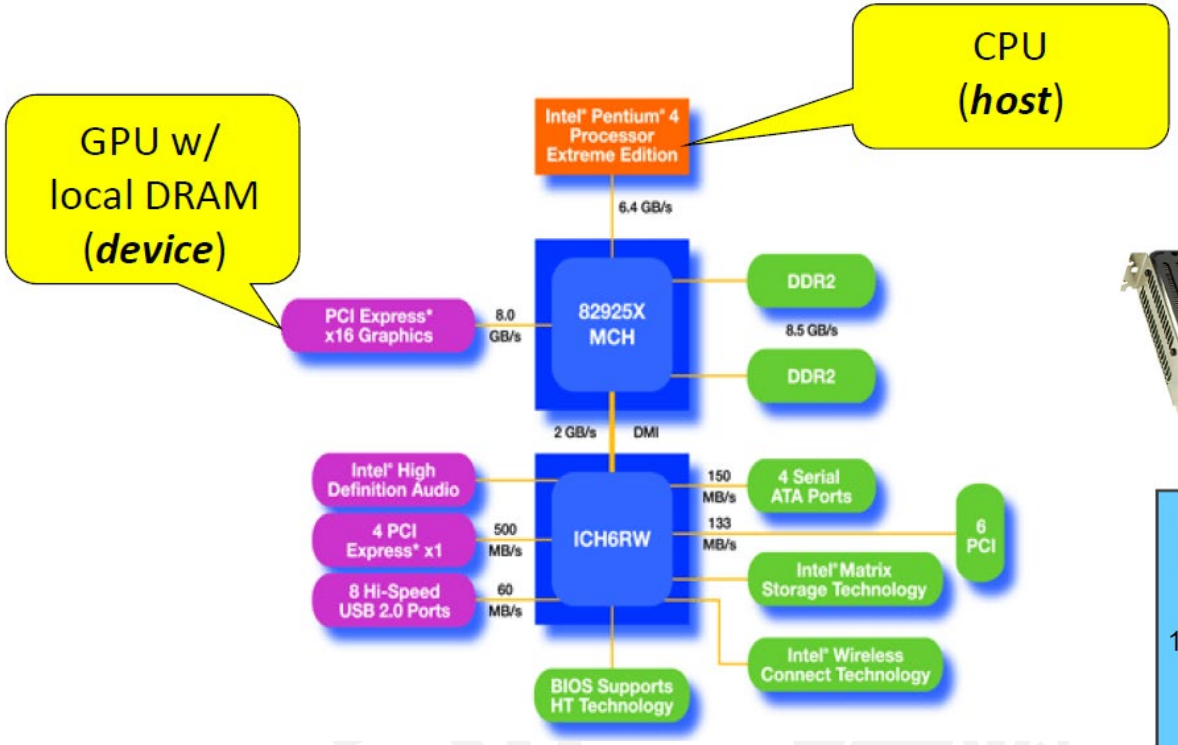
- General Purpose GPU

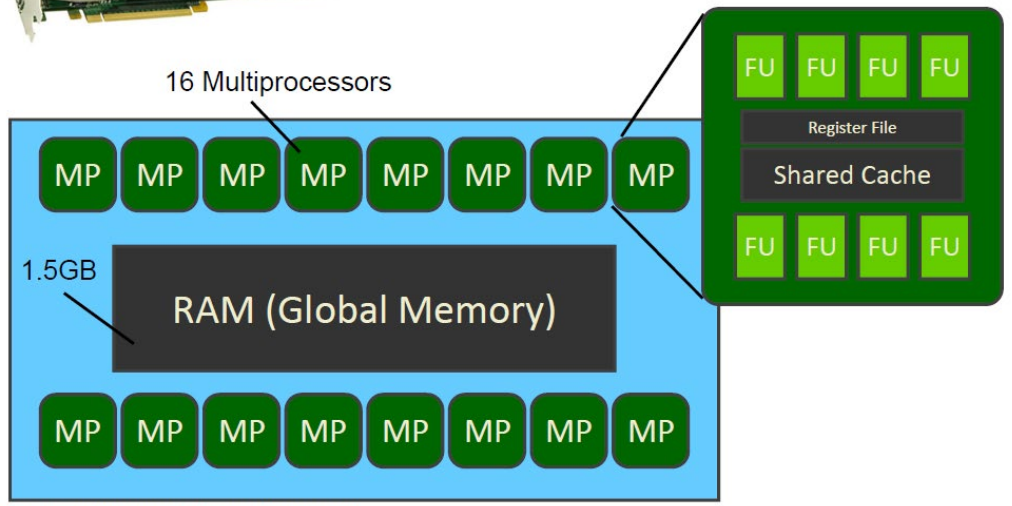


- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor -neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general -purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GPGPU for execution

GPU: Overview

- **Example GGPGPU System & Example GPU**





GPU: Architecture

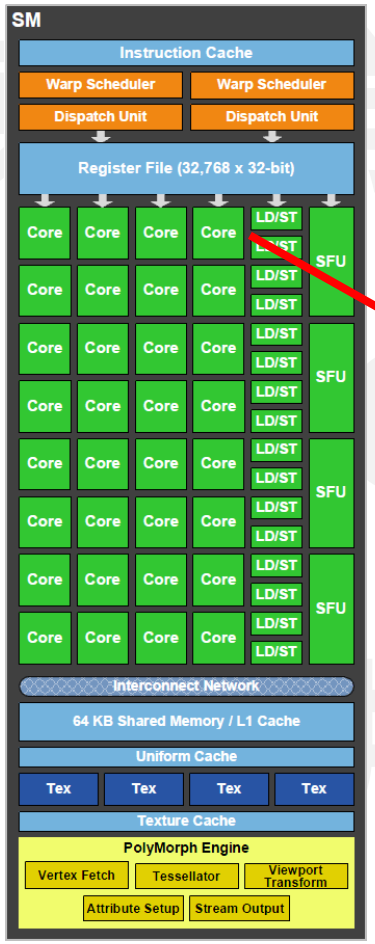
- **Example：NVIDIA Fermi Architecture**

Streaming Processor (SM)

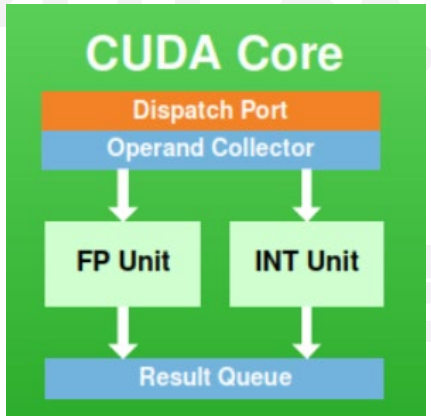
GPU: Architecture

- **Example：NVIDIA Fermi Architecture**

Streaming Processor (SM)



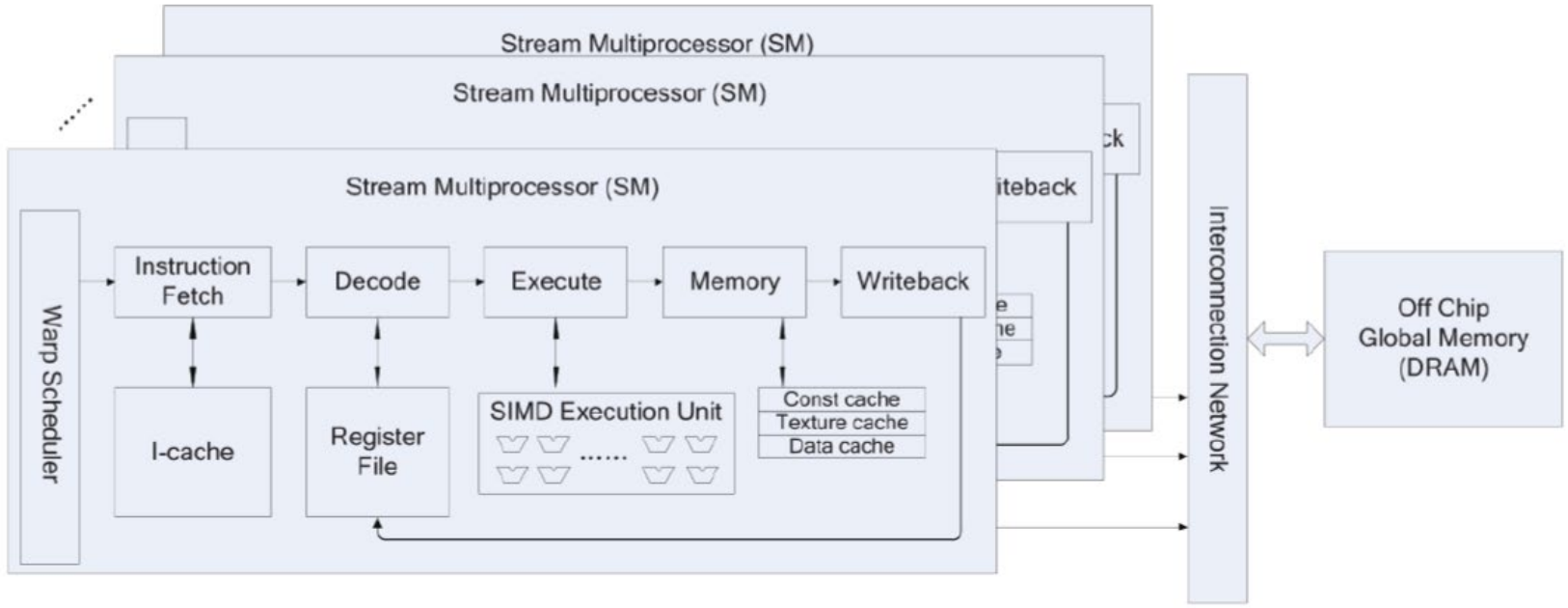
- 每个SM内部有32个Streaming Processor (SP)
- 单个芯片集成多至512个SP
- 峰值算力~200GOPS



GPU: Architecture

- **Warp**

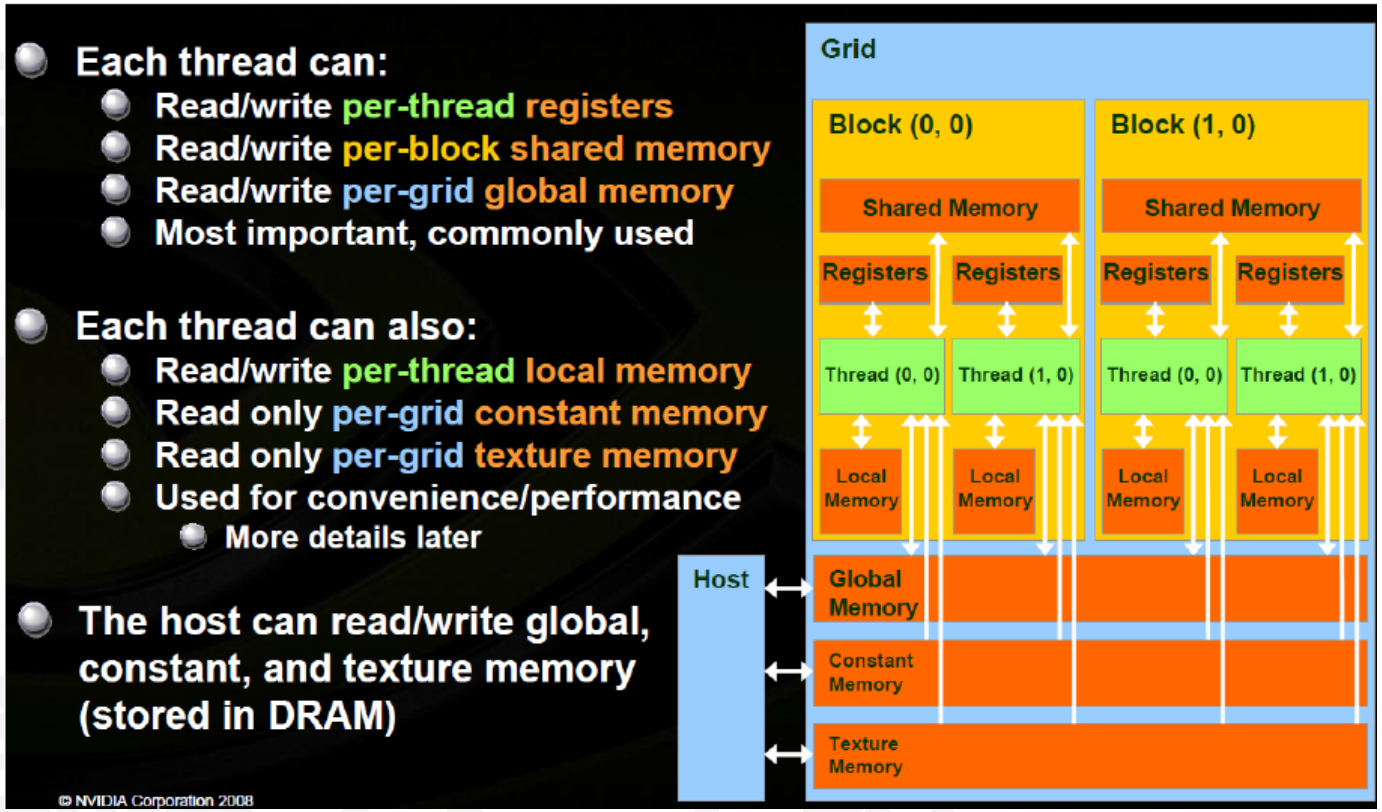
最小thread scheduling unit （对于NVIDIA GPU为32 threads）



GPU: Architecture

- **Memory Hierarchy**

- Shared memory/L1 cache: ~50 cycles
- L2 cache: ~150 cycles
- Global memory (GDDR5): ~500 cycles



GPU: H100 Overview

北京大学 PEKING UNIVERSITY

- **Overview**
  - 性能带宽相对前一代提升
    - 增加FP32 Core、FP64 Core、Tensor Core、SM Core
    - 采用HBM3、 HBM2e增加访存带宽

GPU Features	NVIDIA A100	NVIDIA H100 SXM5[1]	NVIDIA H100 PCIe[1]
GPU Architecture	NVIDIA Ampere	NVIDIA Hopper	NVIDIA Hopper
GPU Board Form Factor	SXM4	SXM5	PCIe Gen 5
SMs	108	132	114
TPCs	54	62	57
FP32 Cores / SM	64	128	128
FP32 Cores / GPU	6912	16896	14592
FP64 Cores / SM (excl. Tensor)	32	64	64
FP64 Cores / GPU (excl. Tensor)	3456	8448	7296
INT32 Cores / SM	64	64	64
INT32 Cores / GPU	6912	8448	7296
Tensor Cores / SM	4	4	4
Tensor Cores / GPU	432	528	456
GPU Boost Clock (Not Finalized for H100) [3]	1410 MHz	Not Finalized	Not Finalized
Peak FP8 Tensor TFLOPS with FP16 Accumulate[1]	NA	2000/4000[2]	1600/3200[2]
Peak FP8 Tensor TFLOPS with FP32 Accumulate[1]	NA	2000/4000[2]	1600/3200[2]
Peak FP16 Tensor TFLOPS with FP16 Accumulate[1]	312/624[2]	1000/2000[2]	800/1600[2]
Peak FP16 Tensor TFLOPS with FP32 Accumulate[1]	312/624[2]	1000/2000[2]	800/1600[2]
Peak BF16 Tensor TFLOPS with FP32 Accumulate[1]	312/624[2]	1000/2000[2]	800/1600[2]
Peak TF32 Tensor TFLOPS[1]	156/312[2]	500/1000[2]	400/800[2]
Peak FP64 Tensor TFLOPS[1]	19.5	60	48
Peak INT8 Tensor TOPS[1]	624/1248[2]	2000/4000[2]	1600/3200[2]
Peak FP16 TFLOPS (non-Tensor)[1]	78	120	96
Peak BF16 TFLOPS (non-Tensor)[1]	39	120	96
Peak FP32 TFLOPS (non-Tensor)[1]	19.5	60	48
Peak FP64 TFLOPS (non-Tensor)[1]	9.7	30	24
Peak INT32 TOPS[1]	19.5	30	24
Texture Units	432	528	456

GPU: H100 Improvements

- **FP8 Improvement**
  - 支持两种格式E5M2与E4M3，中间结果用FP32/FP16存储，减小误差

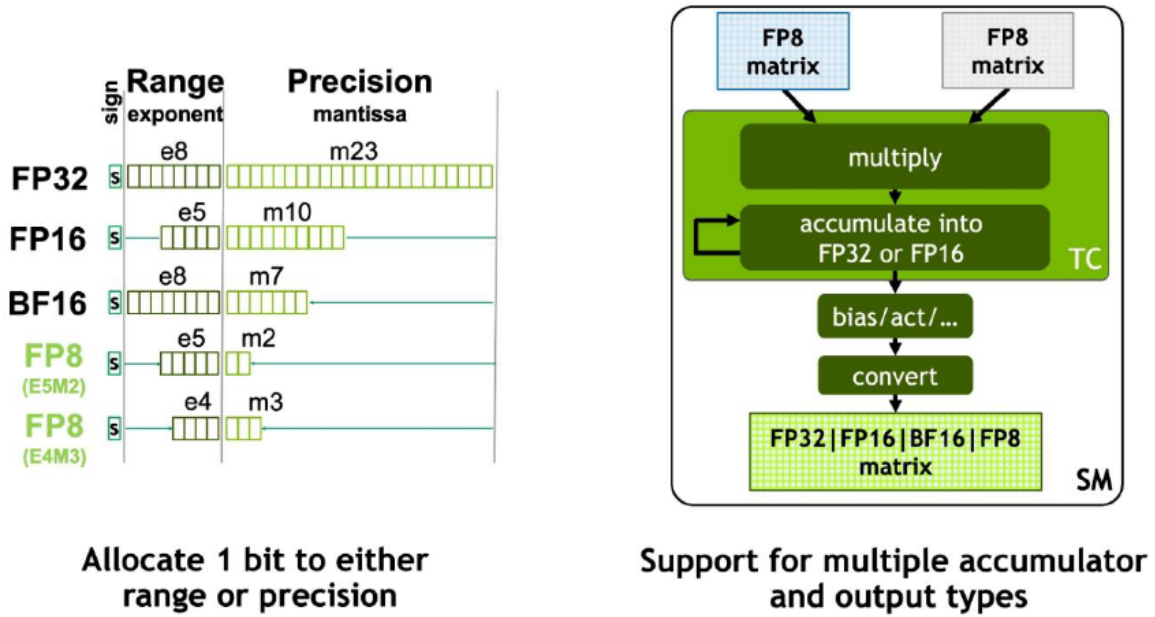


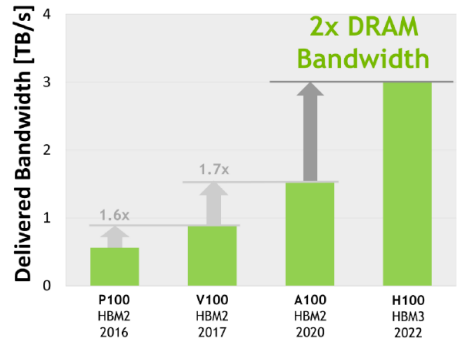
Figure 9. New Hopper FP8 Precisions - 2x throughput and half the footprint of H100 FP16 / BF16

GPU: H100 Improvements

北京大学 PEKING UNIVERSITY

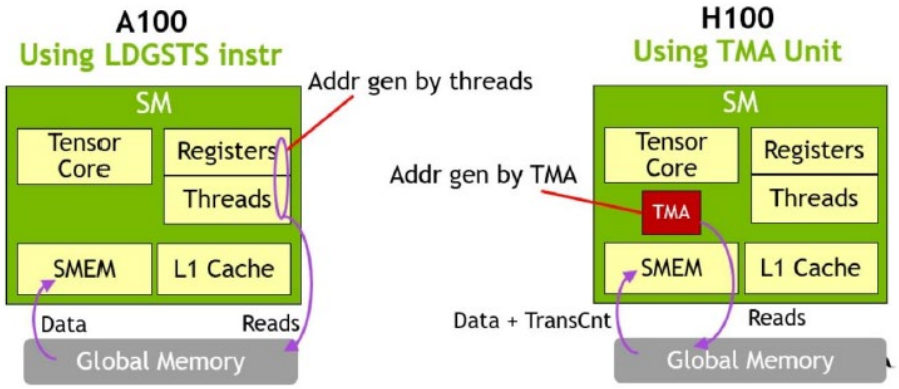
- **Memory Design**

  - Memory支持SECDED纠错码， Sideband ECC for HBM （1 stack for ECC）
  - HBM： Memory Row Remapping
  - L2 cache从40MB增大至50MB，并且支持Data Compression
  - Combined L1 Cache & Shared Memory
    - Total 256KB/SM
  - Tensor Memory Accelerator (TMA)
    - 将大块数据与Tensor在global memory和shared memory间传输
    - Common in domain specific accelerators (DSA)



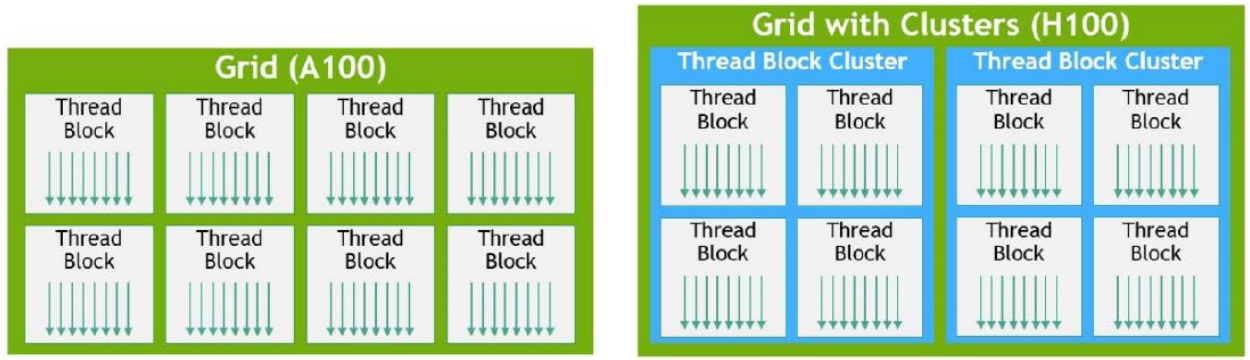
Memory data rates not finalized and subject to change in the final product.

Figure 21. World's First HBM3 GPU Memory Architecture, 2x Delivered Bandwidth

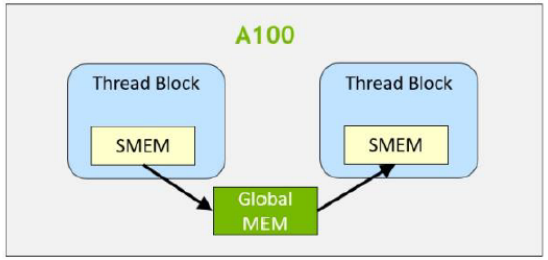


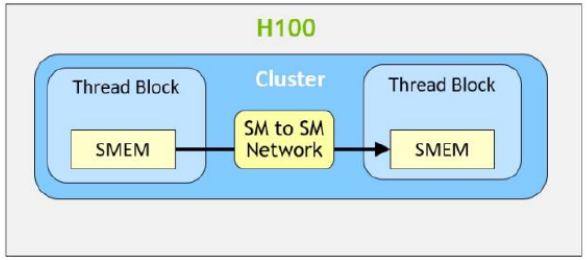
GPU: H100 Improvements

- **Thread Block Clusters**
  - A group of Thread Blocks that are concurrently scheduled onto a group of SMs
  - Clusters have hardware accelerated barriers and new memory access collaboration
  - A dedicated SM-to-SM network provides fast data sharing between threads in a Cluster



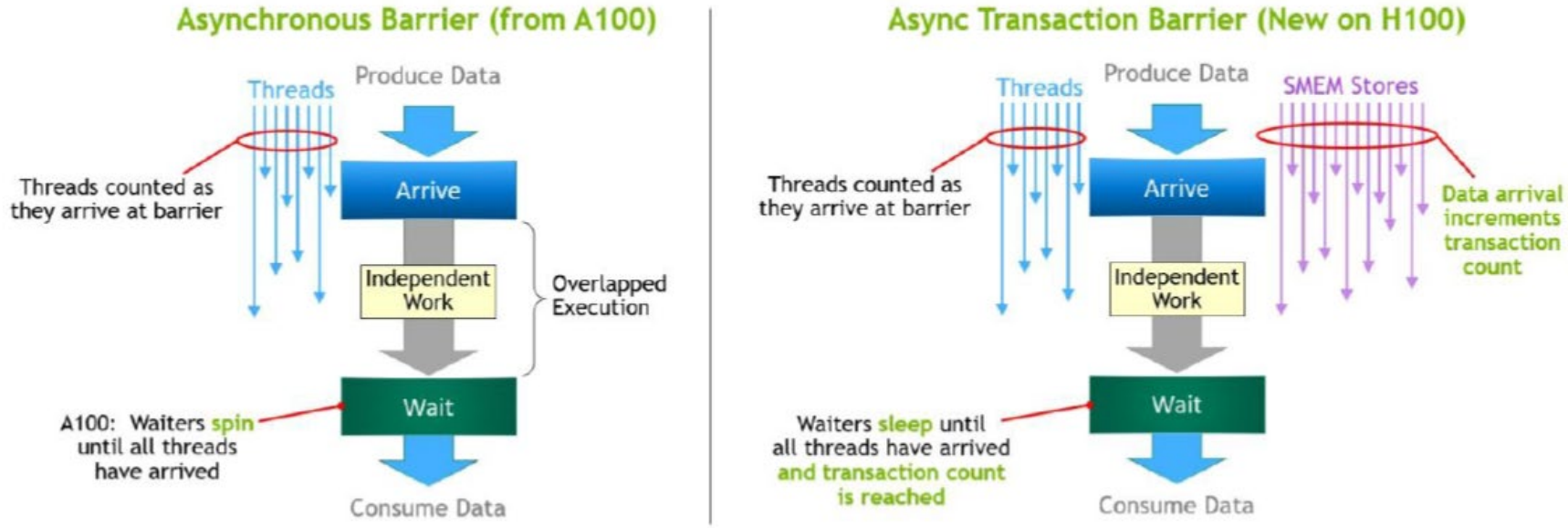
A Grid is composed of Thread Blocks in the legacy CUDA programming model as in A100, shown in the left half of the above diagram. The Hopper architecture adds an optional Cluster hierarchy, shown in the right half of the diagram.





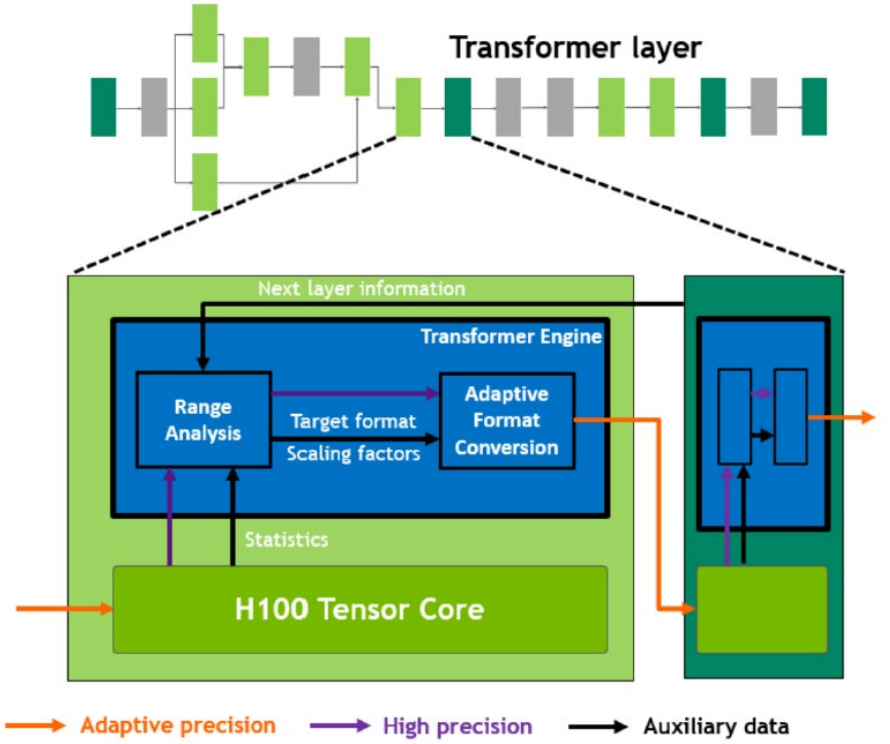
GPU: H100 Improvements

- **Asynchronous Execution**
  - Asynchronous Transaction Barrier
    - Block threads until all the producer threads have performed an Arrive, and the sum of all the transaction counts reaches an expected value



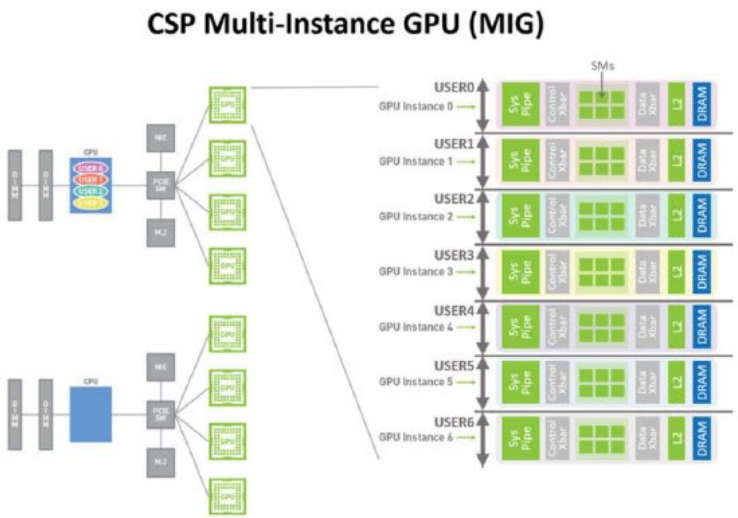
GPU: H100 Improvements

- Transformer Engine
  - 支持混合精度
  - 根据硬件统计分布动态调整scaling factor
    - 硬件统计每层的output range
      - 考虑下一层的range
      - 计算得到高精度的结果
      - scale回下一层的range



GPU: H100 Improvements

- MIG & TEE
  - 面向云端，对于自动驾驶等应用，考虑轻量化、multi instance的支持
    - Multi-Instance GPU (MIG)将一个GPU分为7个独立GPU Instance
    - Trusted Execution Environment (TEE) 建立安全的分GPU与host CPU交互环境



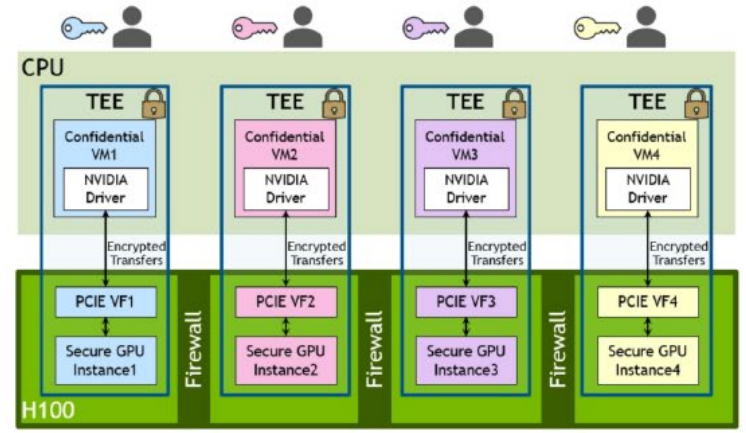
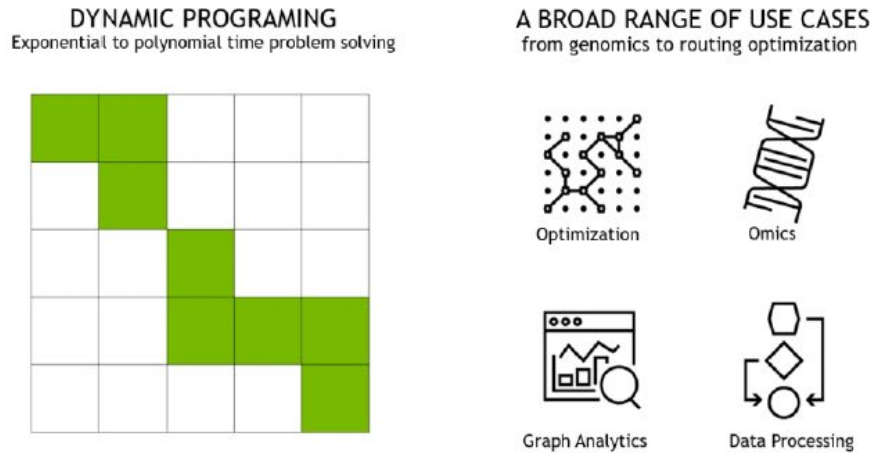
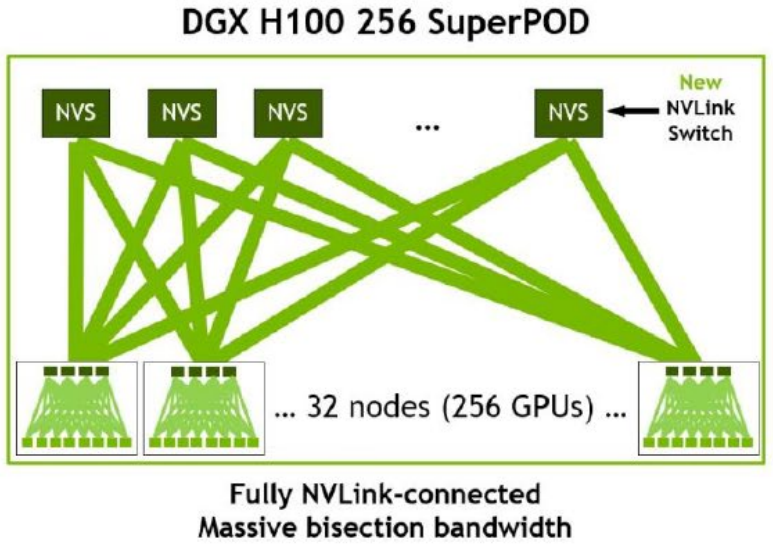


Figure 23. Secure MIG Example in Multi-Tenant Single GPU Configuration

GPU: H100 Improvements

- DPS, NVLink, PCIe Gen5
  - **DPX指令加速Dynamic Programming算法**
  - Gen4 NVLink提供900GB/s的带宽（PCIe Gen5的7倍，A100的1.5倍），支持最多256 GPU的互联
  - Gen3 NVSwitch支持64端口的NVLink，由7.2Tbits/s增加至13.6Tbits/s
  - PCIe Gen5 由16个lane interface提供128GB/s带宽（Gen4两倍）





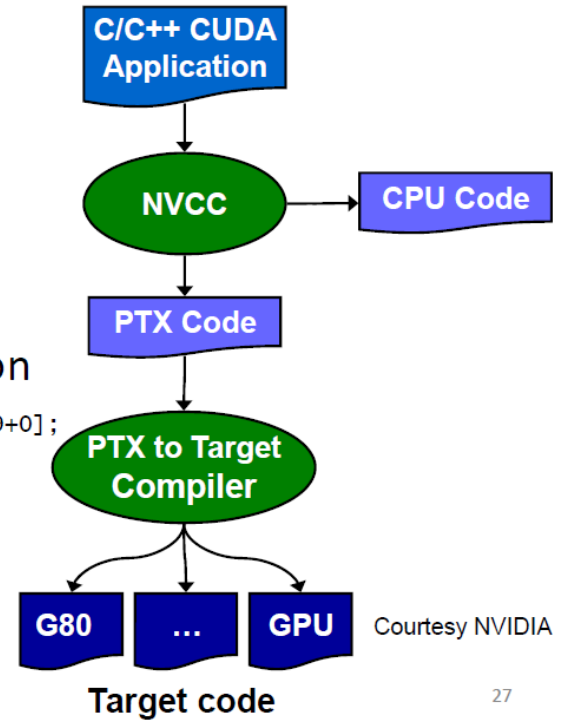
GPU：CUDA

- **C-extension Programming Language**

  - 类似C的编程语言，用于编写在GPU上运行的程序
  - CUDA的抽象十分贴近GPU的实际capabilities和performance characteristics
  - 两种Program Types
    - Device program (Kernel): run on GPU
    - Host program: run on CPU to call device programs

- nvcc
  - Compiler driver
  - Invoke cudacc, g++, cl
- PTX
  - Parallel Thread eXecution

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

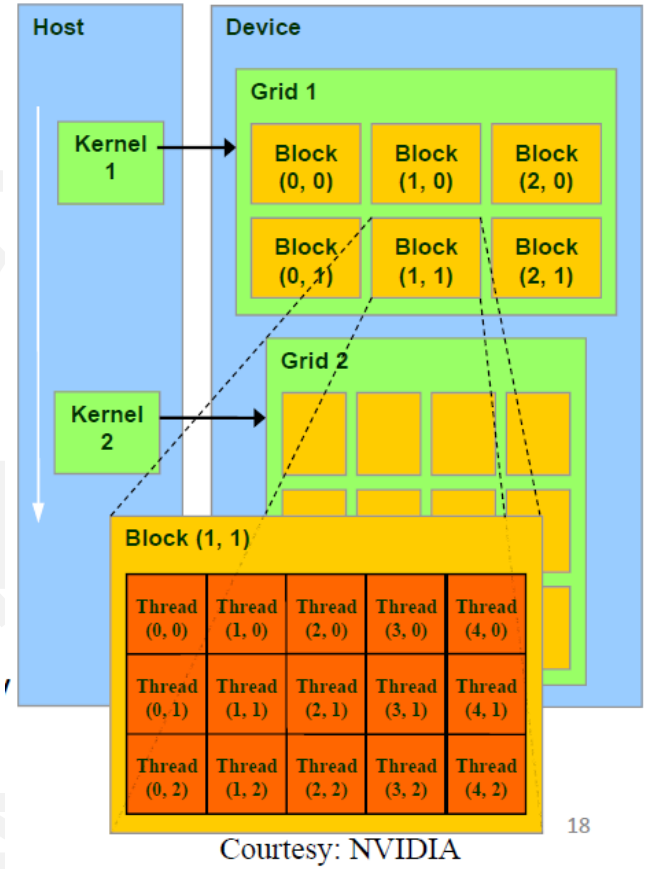


Adapted from Utah SCI Institute

GPU：CUDA

- **Thread Batching**

  - 程序中数据并行的部分以kernel形式在不同thread上被执行
  - Kernel is executed as grid of thread blocks
    - 所有threads共享data memory空间
  - A thread block is a batch of threads that can cooperate with each other by:
    - Synchronizing their execution
    - For hazard-free shared memory accesses
    - Efficiently sharing data through a low latency shared memory



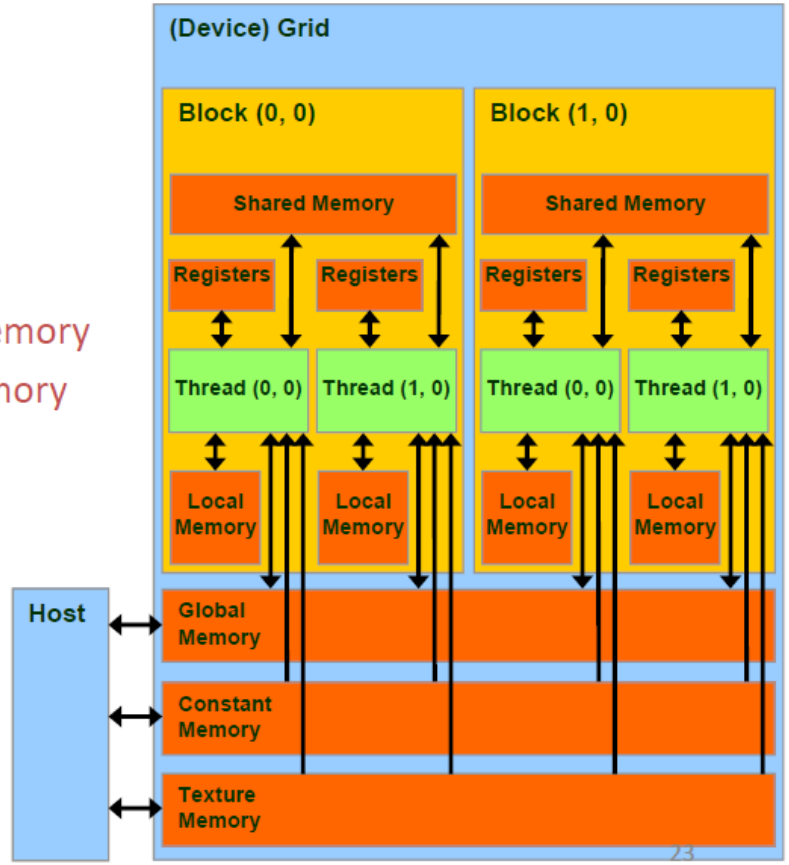
Courtesy: NVIDIA

北京大学 PEKING UNIVERSITY

- Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



目 录

CONTENTS

期末报告

北京大学 PEKING UNIVERSITY

• Paper Review的推荐方向

- **AI相关的神经网络加速器**
  - 传统AI加速器：**Fused-layer cnn accelerators、Eyeriss，Google TPU等**
  - 新兴AI加速器（**大模型Transformer、Neural ODE、MANN/DNC、PINN等**）
- **GPGPU等并行架构**
  - 流式多处理器（**Multithreaded Streaming Multiprocessors，CUDA的来源**）
  - FPGA架构等（**可编程逻辑块、可编程路由等**）
- **安全与通信领域处理器架构**
  - 各类**密钥编码（AES、RSA）、视频编码（MPEG等）、通信编码（LDPC、Polar等）**
- **传统CPU架构**
  - **优化Branch Predictor、Load-Store、缓存预读取、众核缓存一致性等**
- **新兴架构**
  - **存算一体/感存算一体、量子计算、生物信息处理、高维NoC、区块链**
  - **基于后摩尔非CMOS器件的架构（模拟计算架构、动力学计算架构等）**