GPU与加速计算

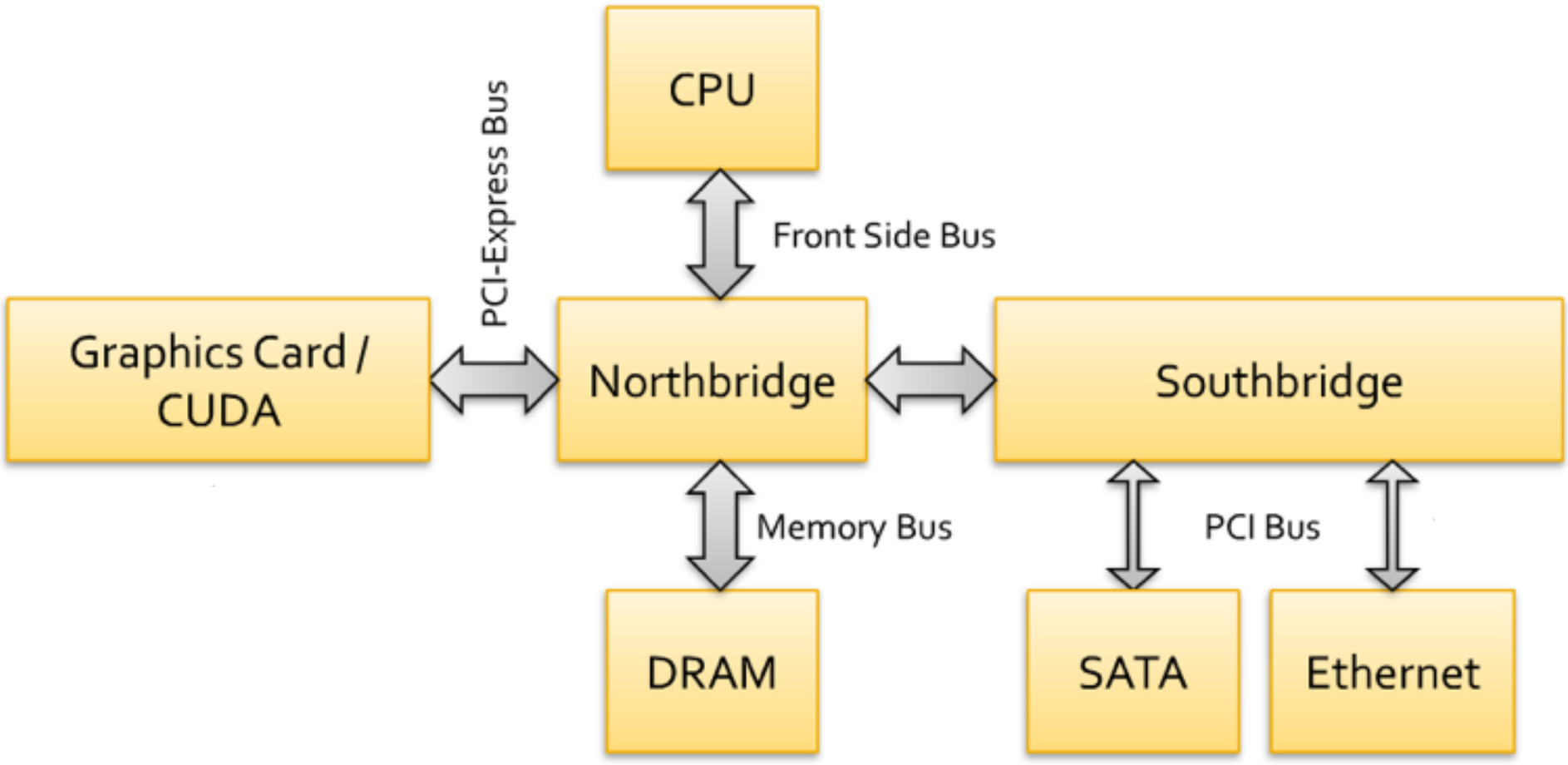
徐世明

清华大学 地球科学中心
xusm@mail.tsinghua.edu.cn

大纲

- 计算机系统与加速计算
- GPU与GPGPU的兴起
- NVIDA CUDA编程与性能优化
- GPU加速的局限性

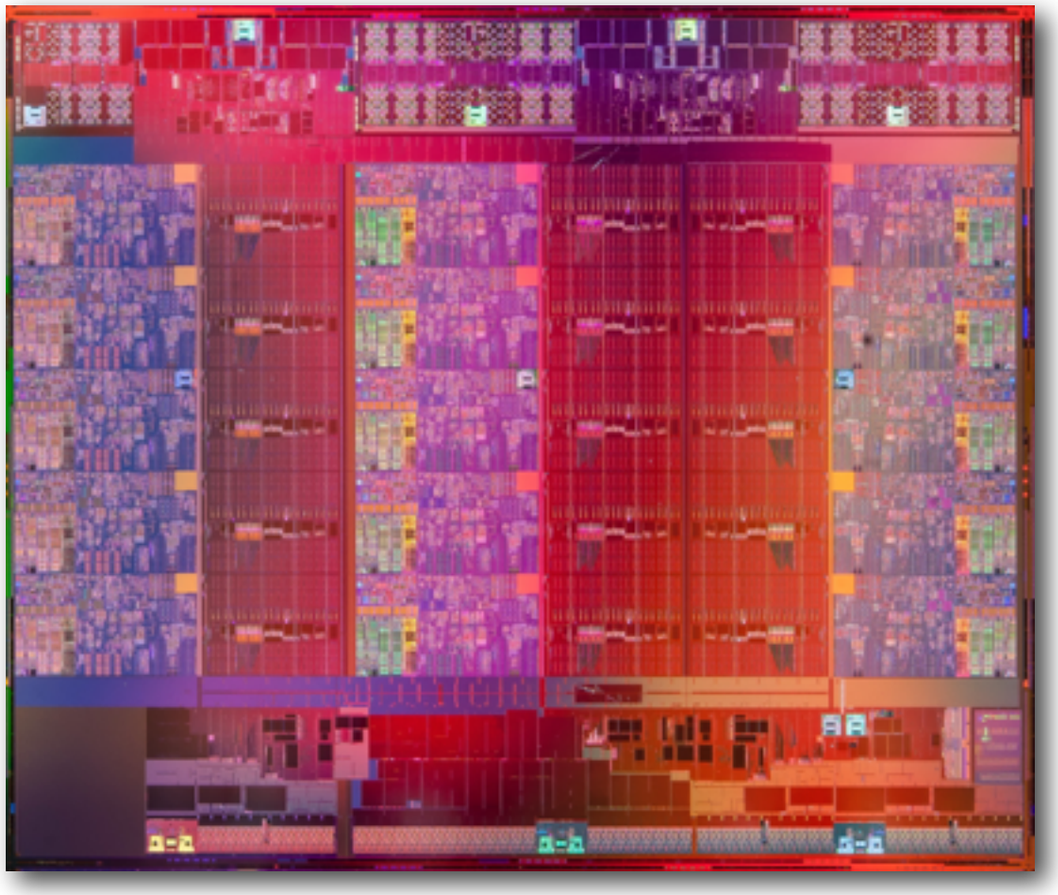
计算机系统


CPU
Front Side Bus
PCI-Express Bus
Graphics Card / CUDA
Northbridge
Southbridge
Memory Bus
PCI Bus
DRAM
SATA
Ethernet

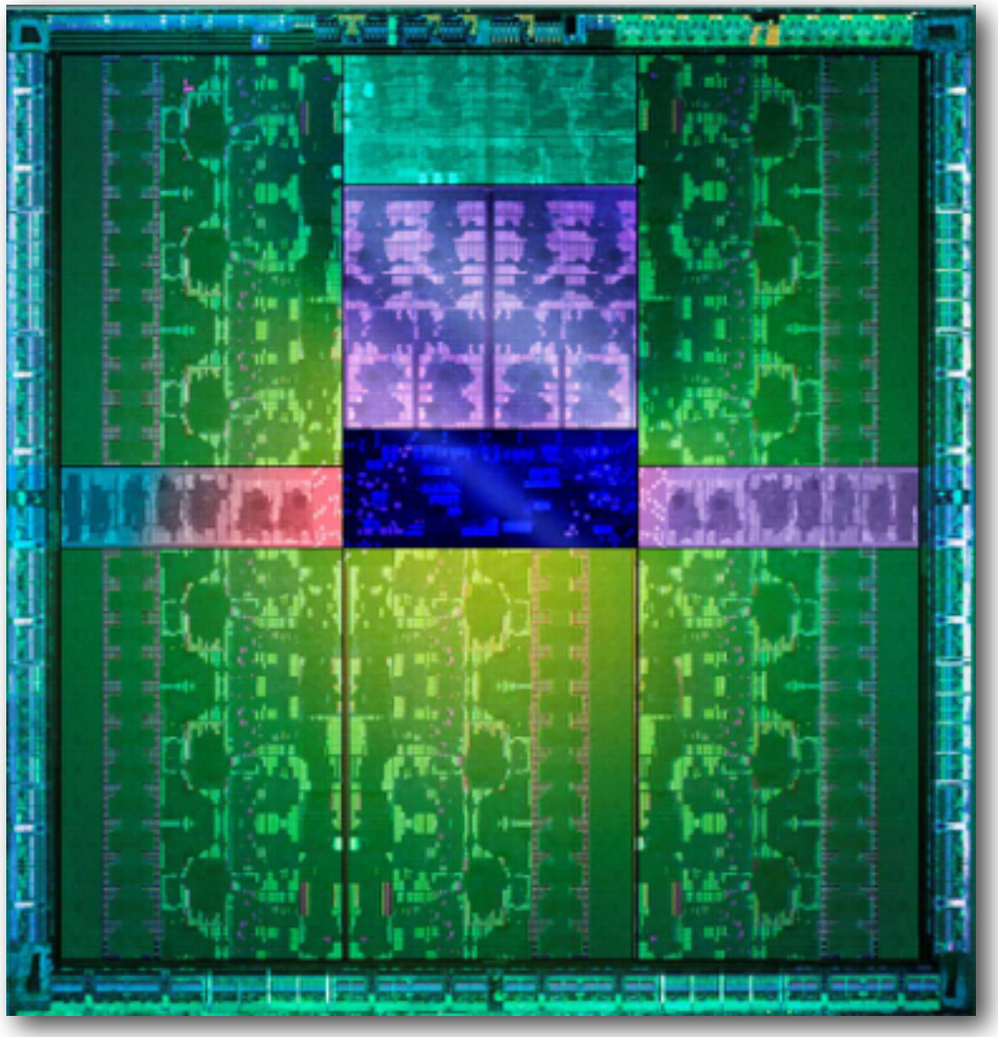
处理器

- 基于硅基和CMOS大规模集成电路
- 完成以下功能：
  - 翻译指令序列为硬件层面的功能
  - 完成指令所指定的功能
  - 输出/写回结果
- 例子：
  - 中央处理器(CPU)，图形处理器(GPU)

处理器是人类的一大创举



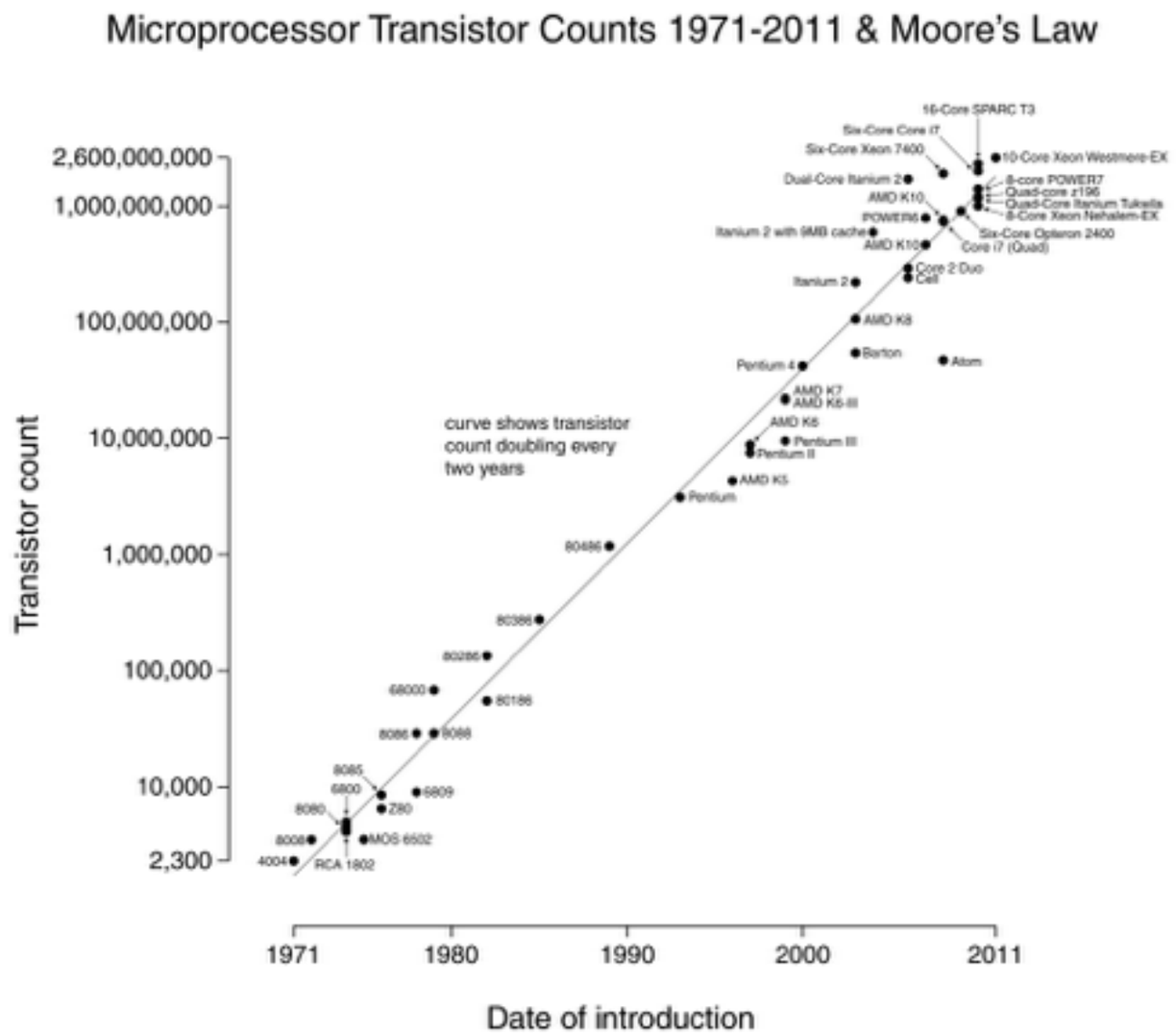
Intel Xeon E7(v2) (2013)
15-core, ~4.3B Trans.
541mm^2 @ 25nm



NVIDIA GK-110B (2013)
15-SMX, 2880-SP, ~7.1B Trans.
551mm^2 @ 28nm

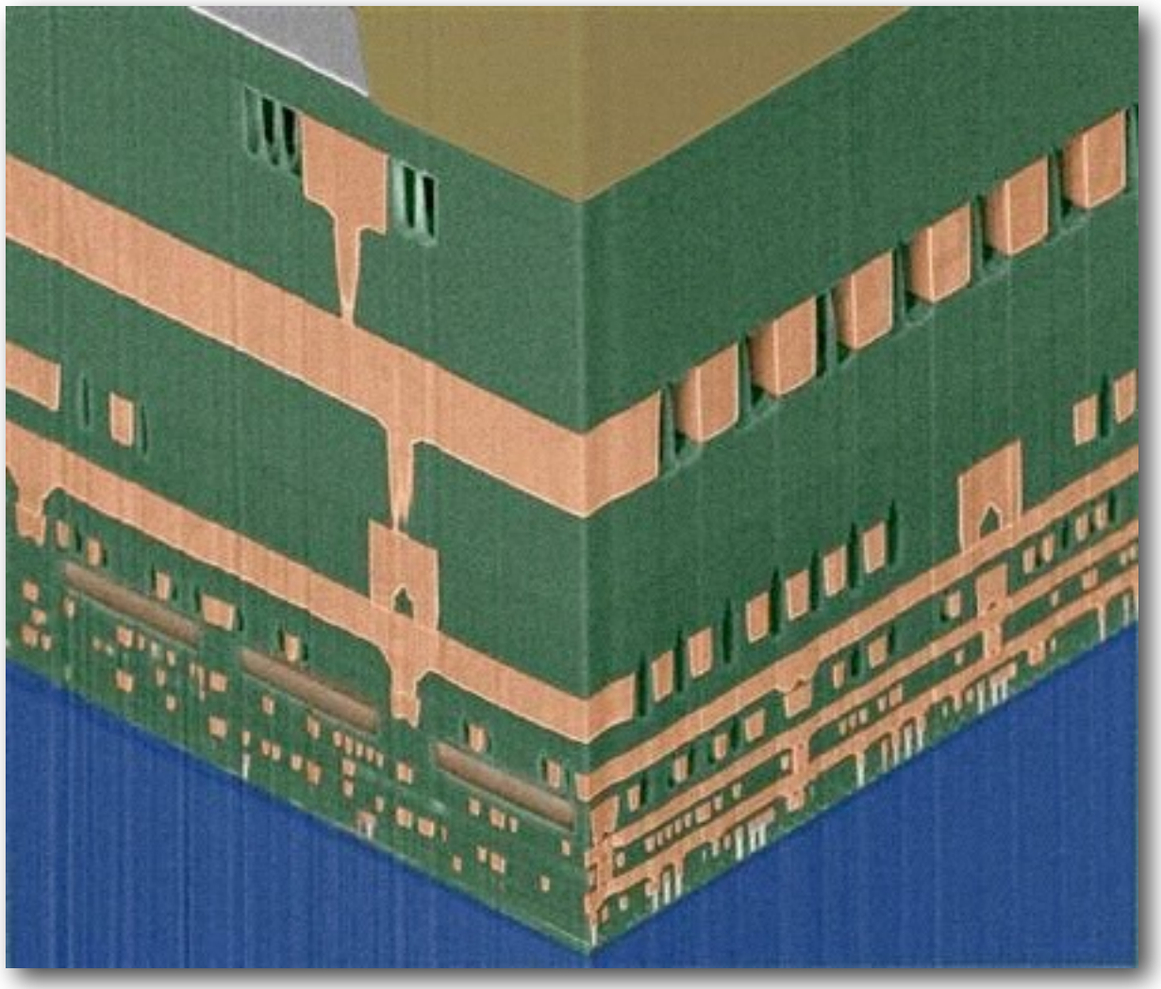
摩尔定律

- 摩尔于1965年提出
- 处理器（CPU）集成度每12~24个月翻番
- 经验总结，规律而非客观定律



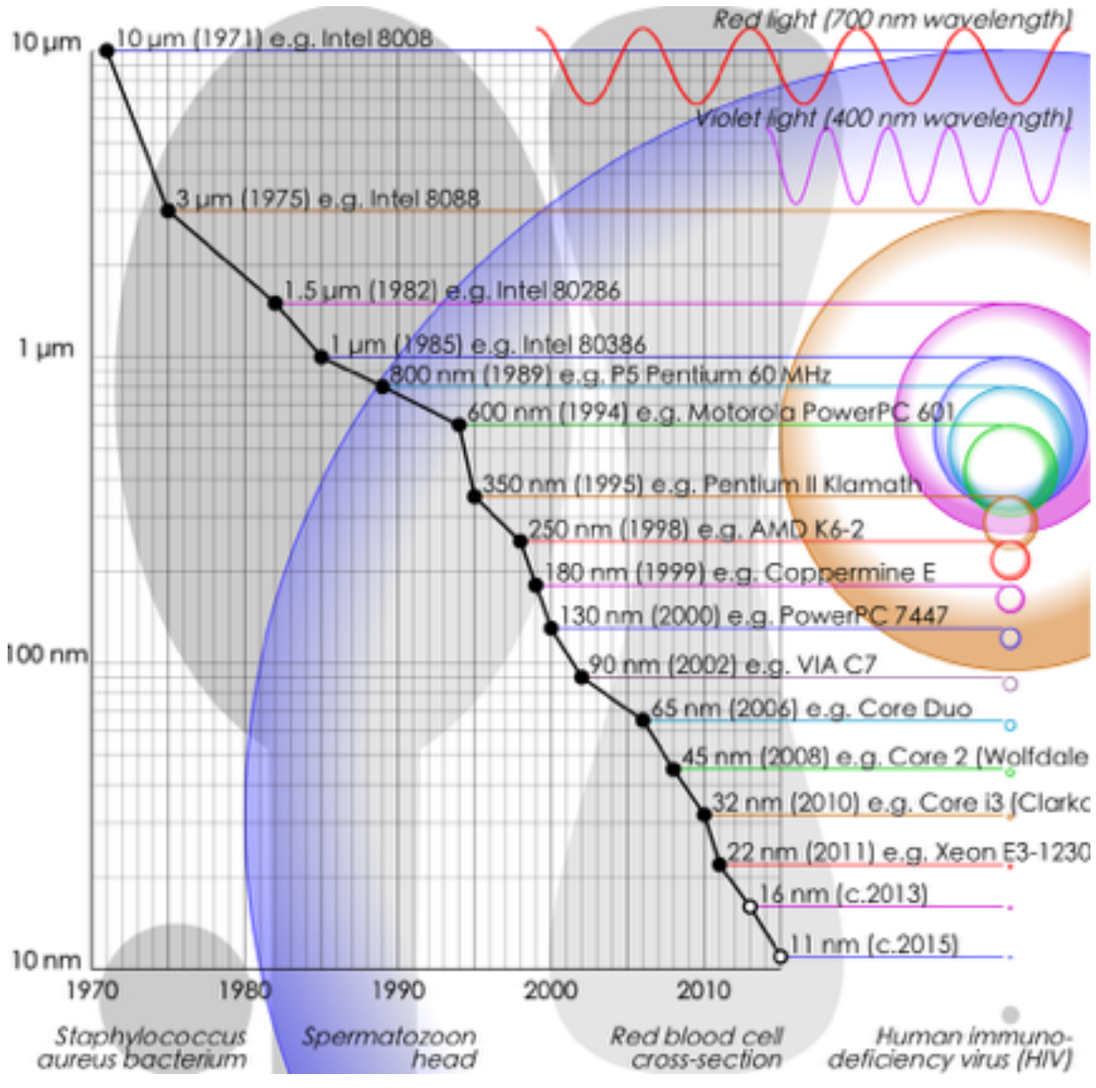
摩尔定律与制程进步

- 制程（Fabrication Process）
  - 典型尺寸：典型线宽
  - 可间接表达晶体管的密度
- 制程的进步保证摩尔定律的指数级增长速度



摩尔定律与制程进步

- 制程（Fabrication Process）
  - 典型尺寸：典型线宽
    - 可间接表达晶体管的密度
- 制程的进步保证摩尔定律的指数级增长速度



处理器的功能与结构

- 硬件-软件接口：指令集
- 例子：x86(x86-64)，ARM，MIPS，PTX
- 是一种功能抽象
- 缓慢演化的指令集vs快速发展的处理器

CPU所处理的并行度

- 任务级并行
- 线程级并行（Multiple threads）
- 指令级并行： Instruction Level Parallelism
- 数据并行指令： SIMD

任务与线程并行

- 任务级并行
  - MPI
- 线程级并行
  - OpenMP

指令级并行性

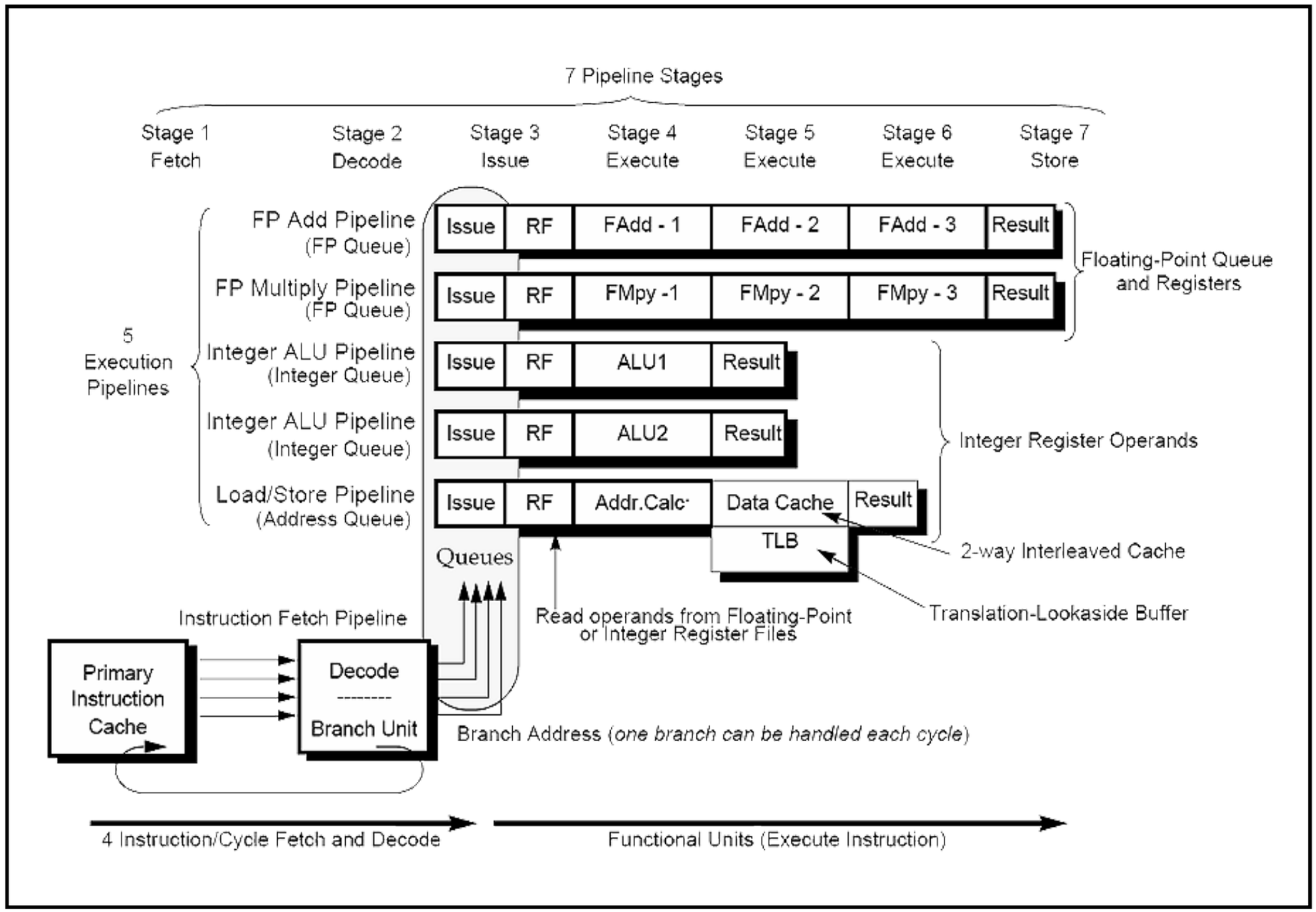
- 由硬件开发指令级别并行性（Instruction-level parallelism）
- 软件透明性

```
1. e = a + b
2. f = c + d
3. g = e * f
```

典型的CPU结构

- 流水线结构
  - 目的：增加吞吐率
- 复杂的控制系统
  - 用于开发指令级并行度
- 访存机制
  - Cache，TLB，等等

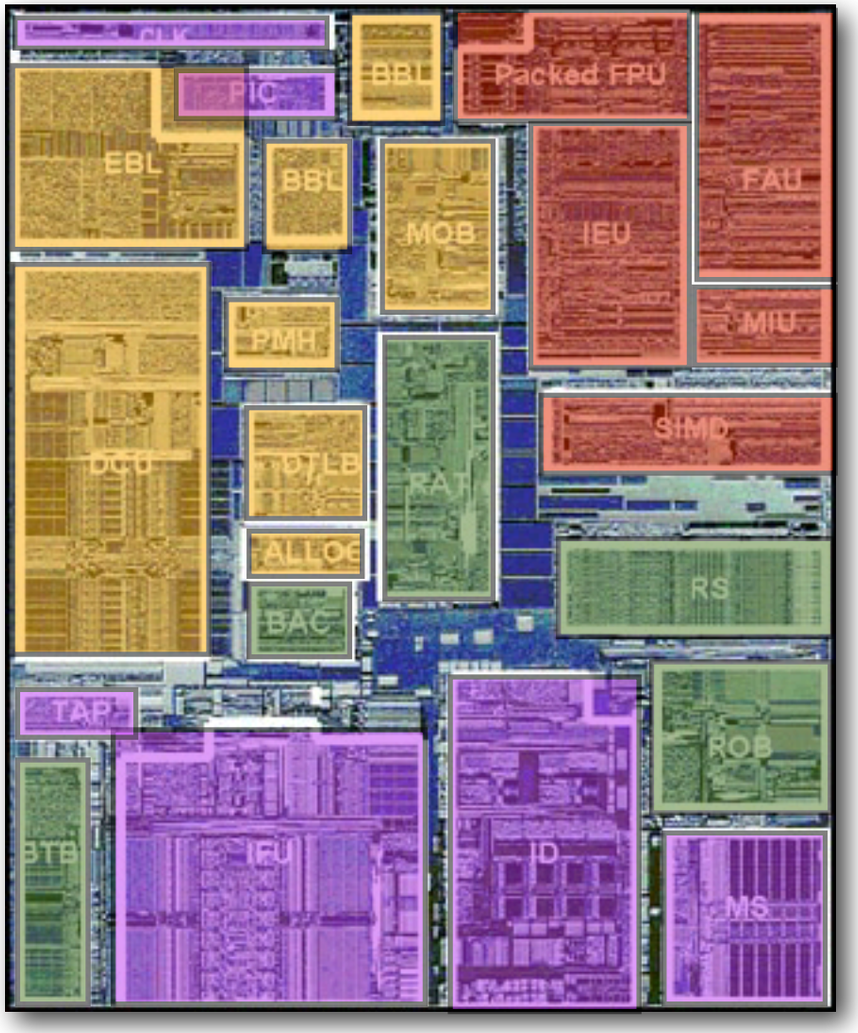
典型的CPU结构-2


7 Pipeline Stages
Stage 1 Fetch
Stage 2 Decode
Stage 3 Issue
Stage 4 Execute
Stage 5 Execute
Stage 6 Execute
Stage 7 Store
FP Add Pipeline (FP Queue)
FP Multiply Pipeline (FP Queue)
Integer ALU Pipeline (Integer Queue)
Integer ALU Pipeline (Integer Queue)
Load/Store Pipeline (Address Queue)
5 Execution Pipelines
Issue
RF
FAdd - 1
FAdd - 2
FAdd - 3
Result
FMpy -1
FMpy - 2
FMpy - 3
ALU1
ALU2
Addr.Calc
Data Cache
TLB
Floating-Point Queue and Registers
Integer Register Operands
2-way Interleaved Cache
Translation-Lookaside Buffer
Queues
Instruction Fetch Pipeline
Primary Instruction Cache
Decode
Branch Unit
Read operands from Floating-Point or Integer Register Files
Branch Address (one branch can be handled each cycle)
4 Instruction/Cycle Fetch and Decode
Functional Units (Execute Instruction)

指令级并行性-2

- 超标量结构（Superscalar）
- 寄存器重命名（Register Renaming）
- 乱序执行（Out-of-Order Execution）
- 预测执行（Speculative Execution）
- 分支预测（Branch Prediction）

芯片效率



EBL/BBL - Bus logic, Front, Back
MOB - Memory Order Buffer
Packed FPU - MMX Fl. Pt. (SSE)
IEU - Integer Execution Unit
FAU - Fl. Pt. Arithmetic Unit
MIU - Memory Interface Unit
DCU - Data Cache Unit
PMH - Page Miss Handler
DTLB - Data TLB
BAC - Branch Address Calculator
RAT - Register Alias Table
SIMD - Packed Fl. Pt.
RS - Reservation Station
BTB - Branch Target Buffer
IFU - Instruction Fetch Unit (+I$)
ID - Instruction Decode
ROB - Reorder Buffer
MS - Micro-instruction Sequencer

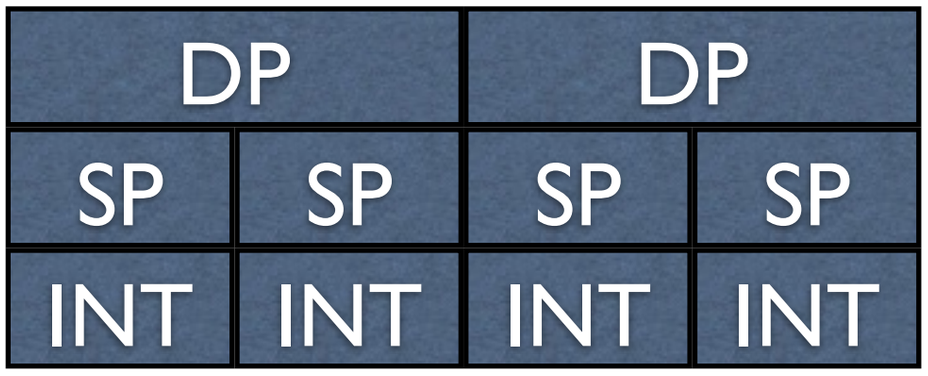
数据并行指令

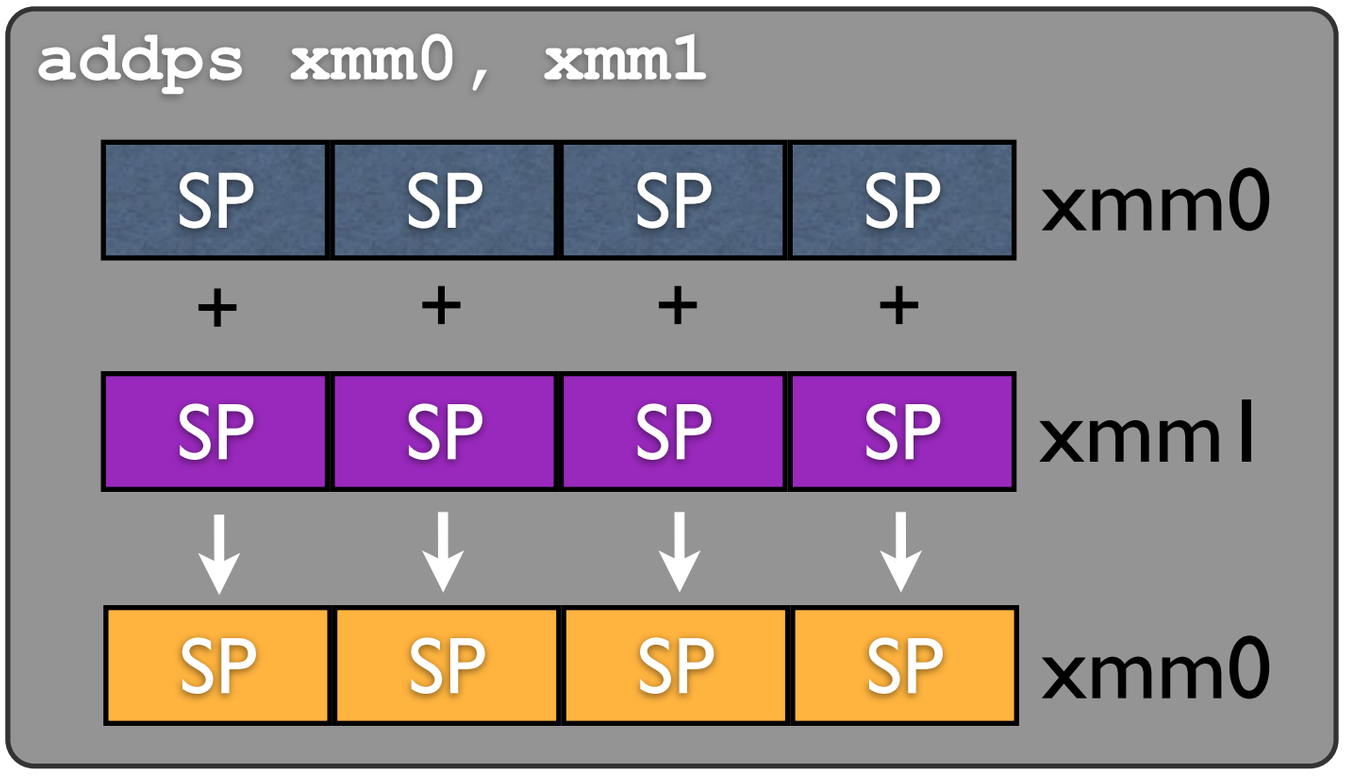
- 单指令-多数据（SIMD）
  - 最常见的处理器指令扩展方式
  - Intel: MMX ~ SSE ~ AVX. IBM: AltiVec.

指令集	MMX	SSE	SSE2/3/4	AVX
年份	1996	1999	2001/2004/2007	2011
宽度	64	128		256
个数	8	8		16
数据支持	INT	FP	DP	DP

数据并行指令-2

xmm寄存器结构





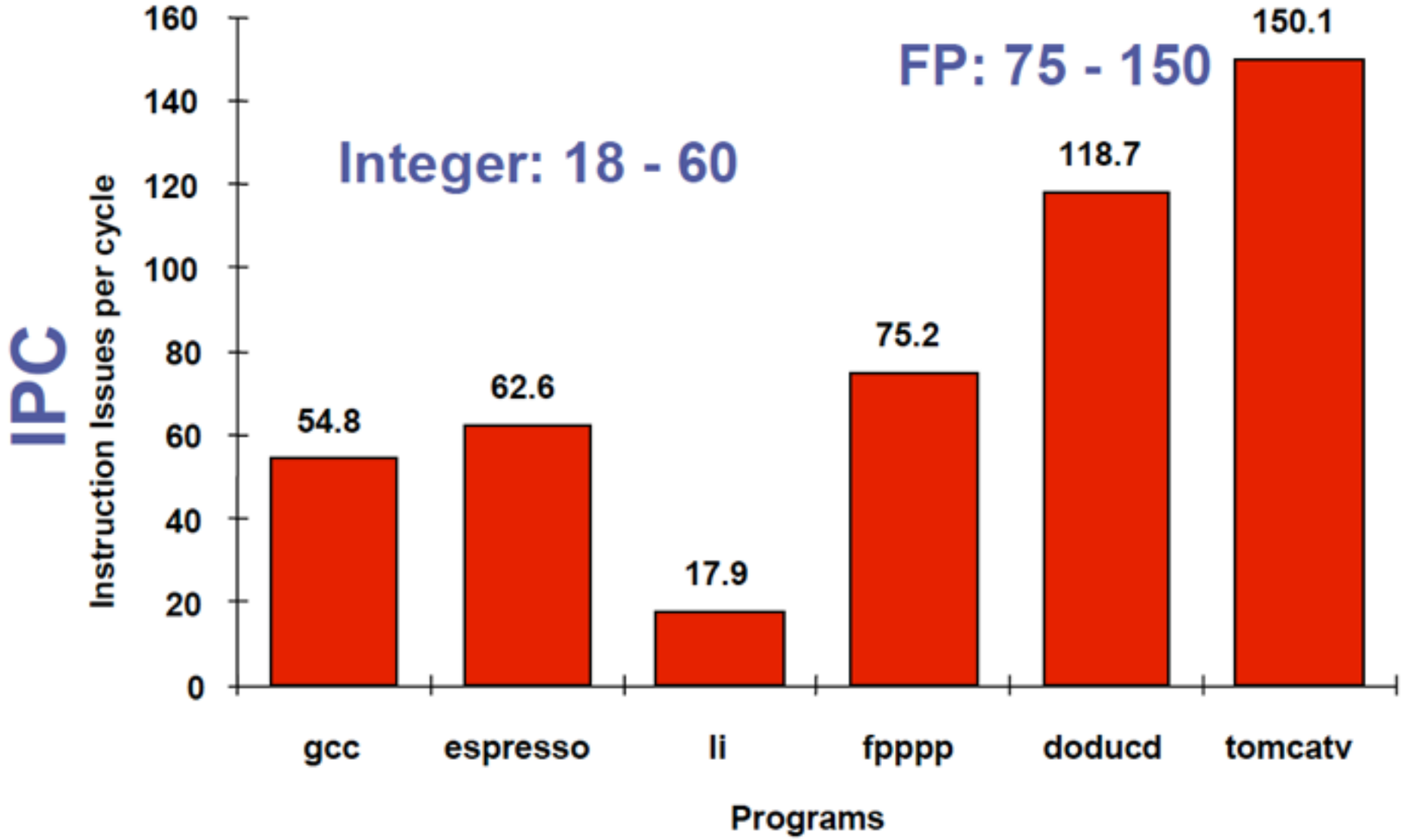

```
float a[N], b[N], c[N];

   ...

   for (i = 0; i < N; i++)
     a[i] = b[i] + c[i];
```

```
Back:
  movaps xmm0, _b[ecx]
  movaps xmm1, _b[ecx]
  addps  xmm0, xmm1
  movaps _a[ecx], xmm0
  add    ecx, 16
  cmp    ecx, edi
  jl     Back
```

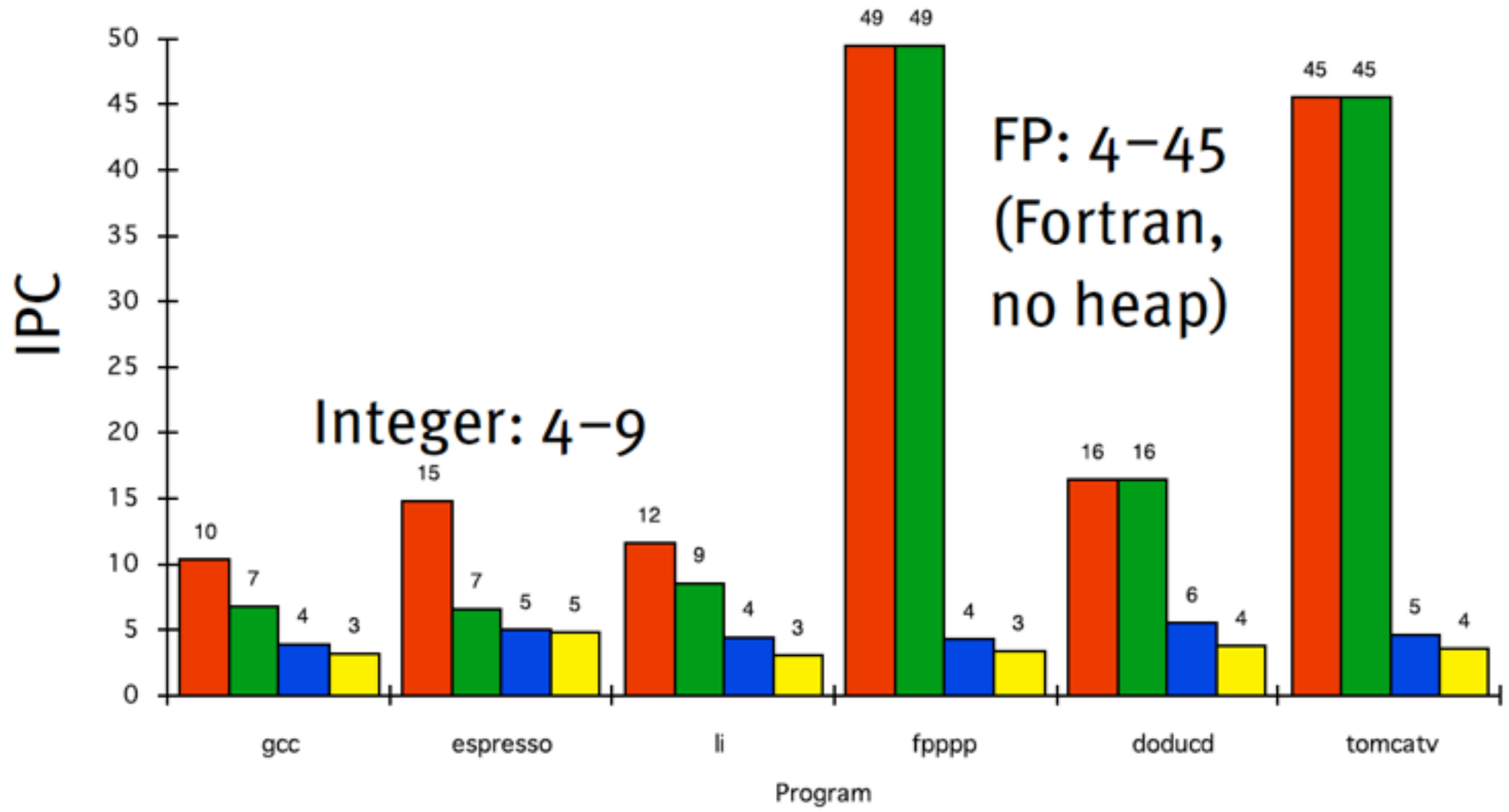
CPU发展面临的问题

- 指令级并行度有限
- 处理器-访存差异增大
- 功耗效率过低，发展无更大空间

(I) 指令级并行度有限


IPC
Instruction Issues per cycle
160
140
120
100
80
60
40
20
0
Integer: 18 - 60
FP: 75 - 150
54.8
62.6
17.9
75.2
118.7
150.1
gcc
espresso
li
fpppp
doducd
tomcatv
Programs

实际可开发的ILP



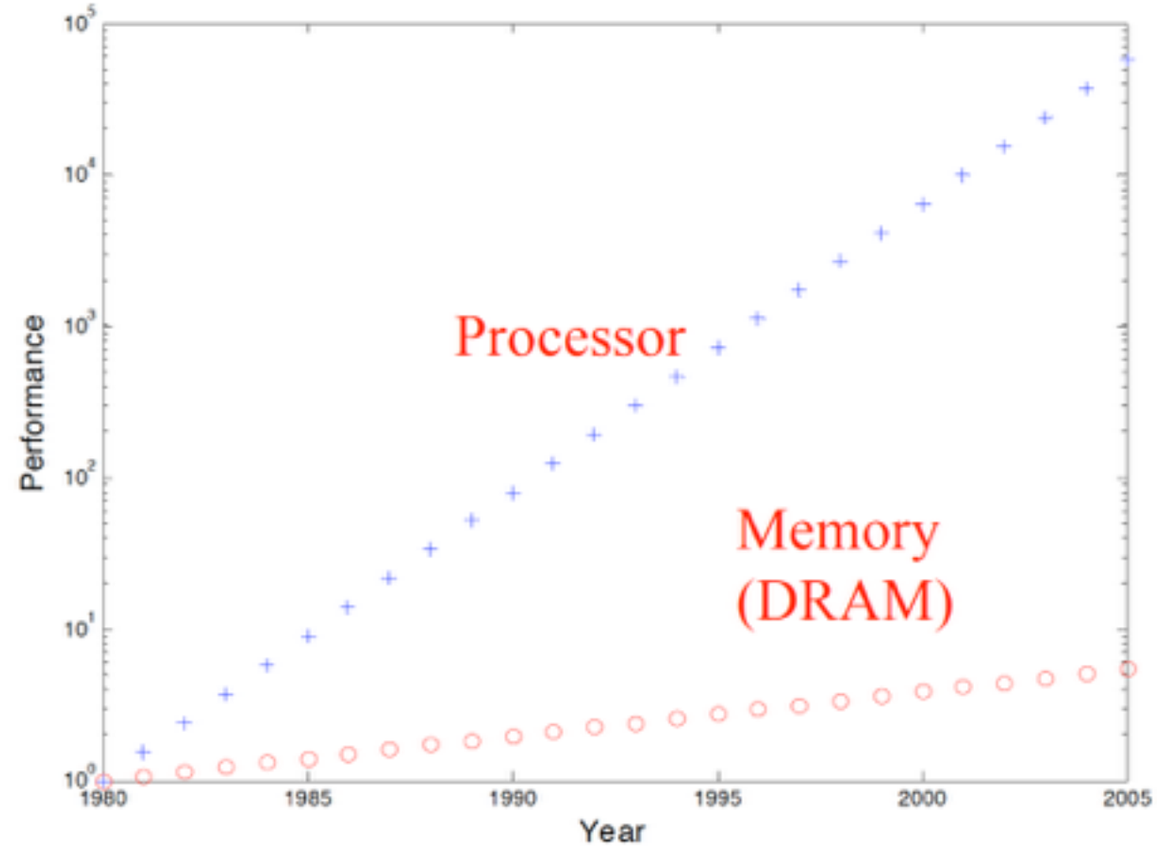
2048 inst. window, 64-issue, 8K b-pre, 256 int reg., 256 fp reg.

原因？硬件限制

- 分支预测精确度不足
- 指令窗口不够大
- 重命名寄存器数量有限
- 内存操作重名（Write-After-Write）

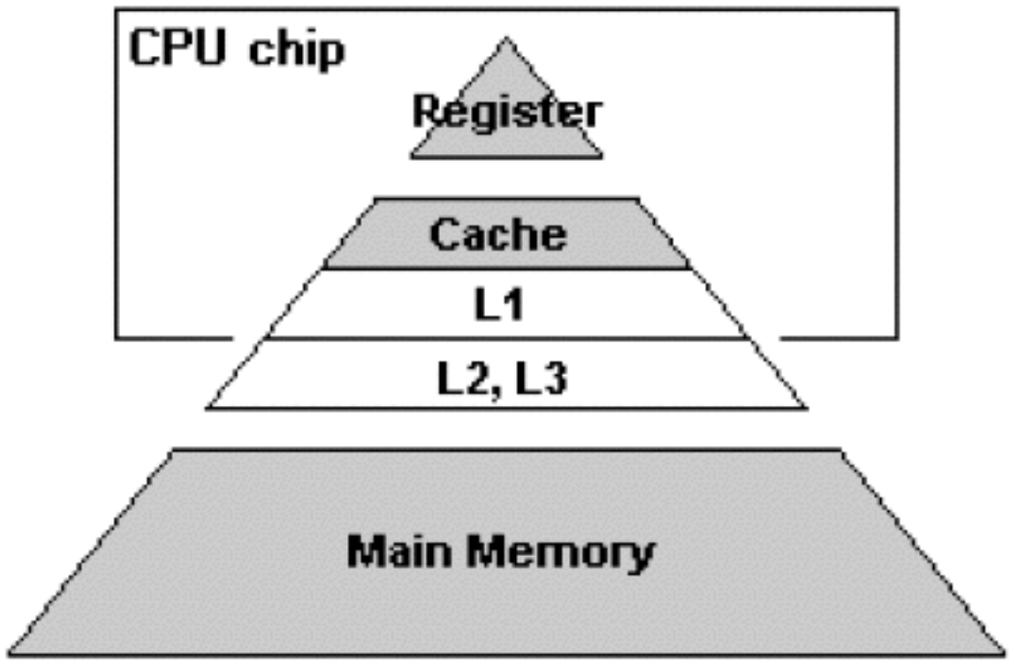
(2) CPU-内存发展不协调

1987	FPM	50ns
1996	EDO	50ns
1997	SDRAM	15ns
1998		10ns
1999		7.5ns
2000	DDR SDRAM	3.75ns
2001		3.00ns
2002		2.30ns
2003		2ns



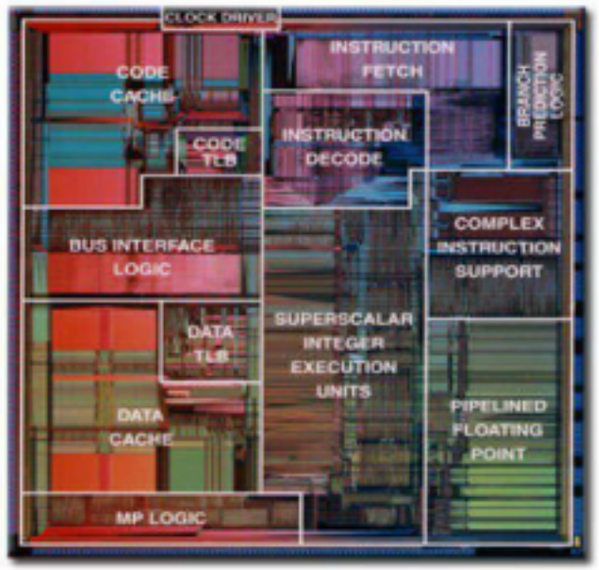
解决方法？Cache

- 透明的内存管理
- 速度？基于SRAM
  - 越小越快
  - 越大越慢
- 层级结构

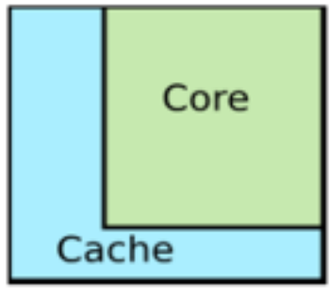


CPU-缓存发展

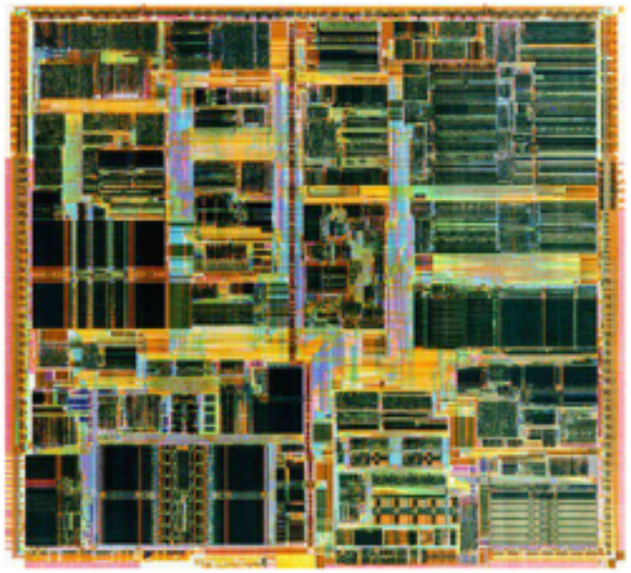
Pentium I

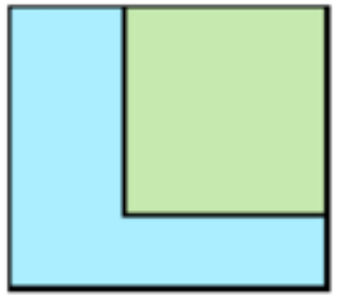


Chip area breakdown

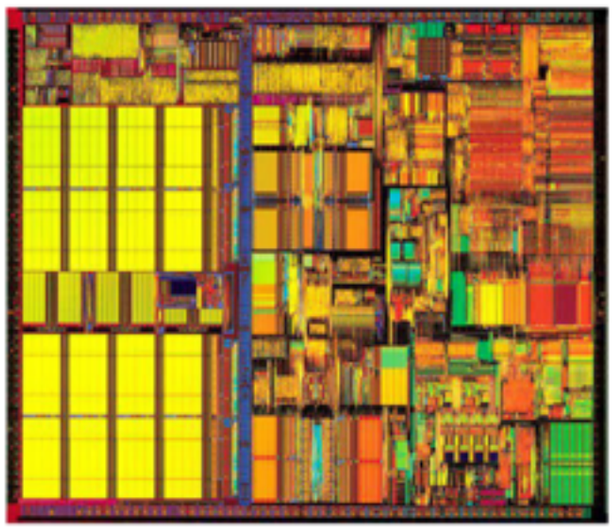


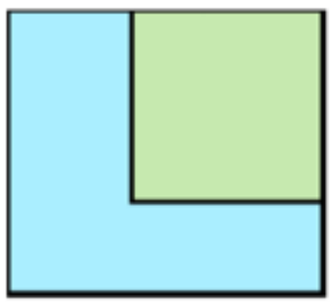
Pentium II



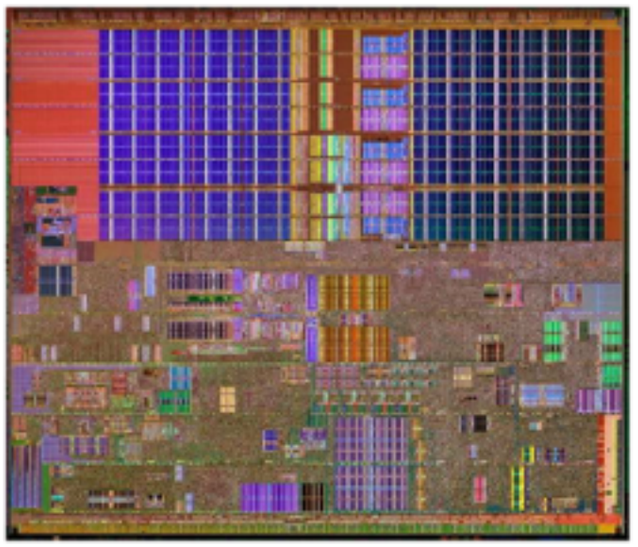


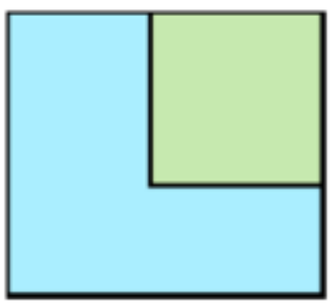
Pentium III





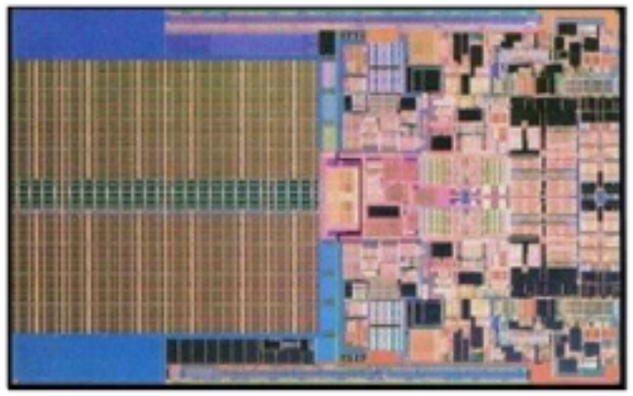
Pentium IV



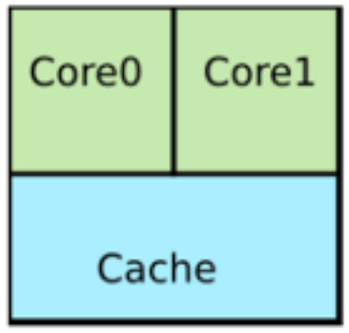


CPU-缓存发展

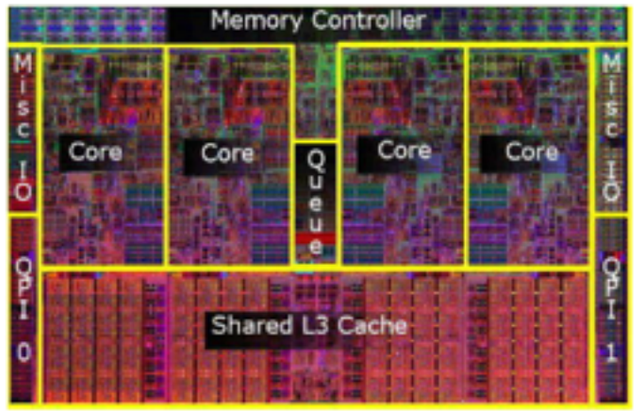
Penryn

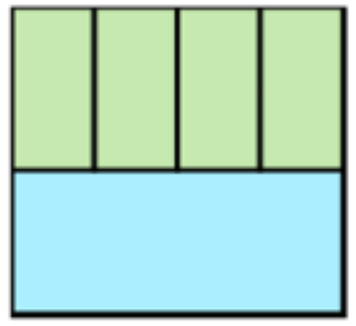


Chip area breakdown

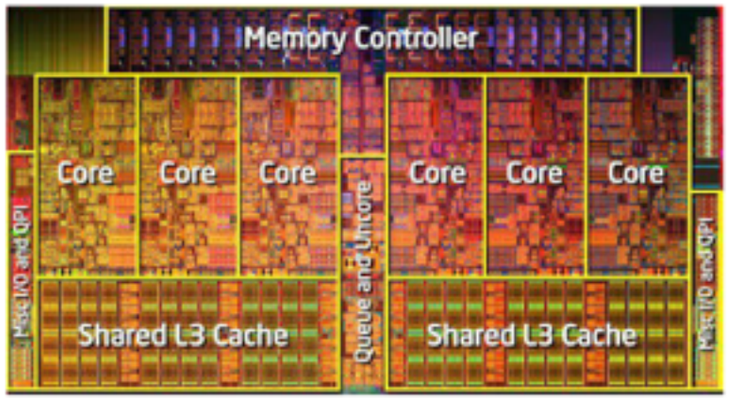


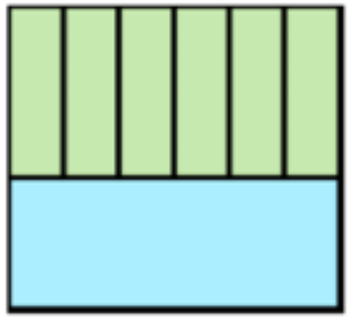
Bloomfield



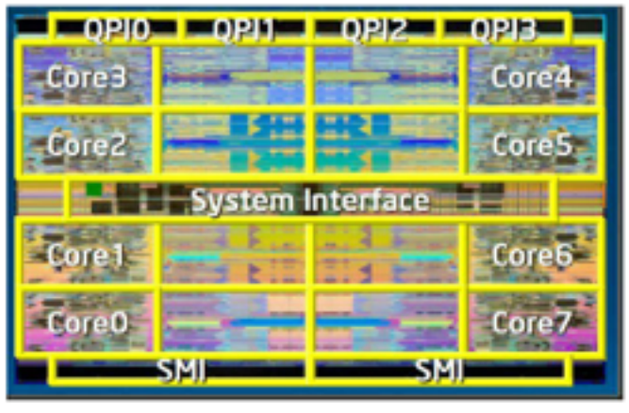


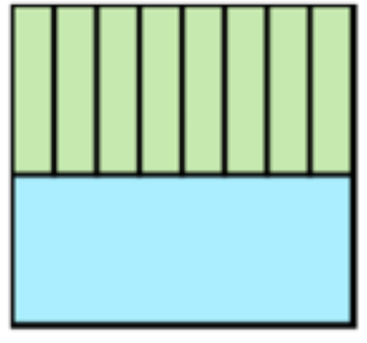
Gulftown



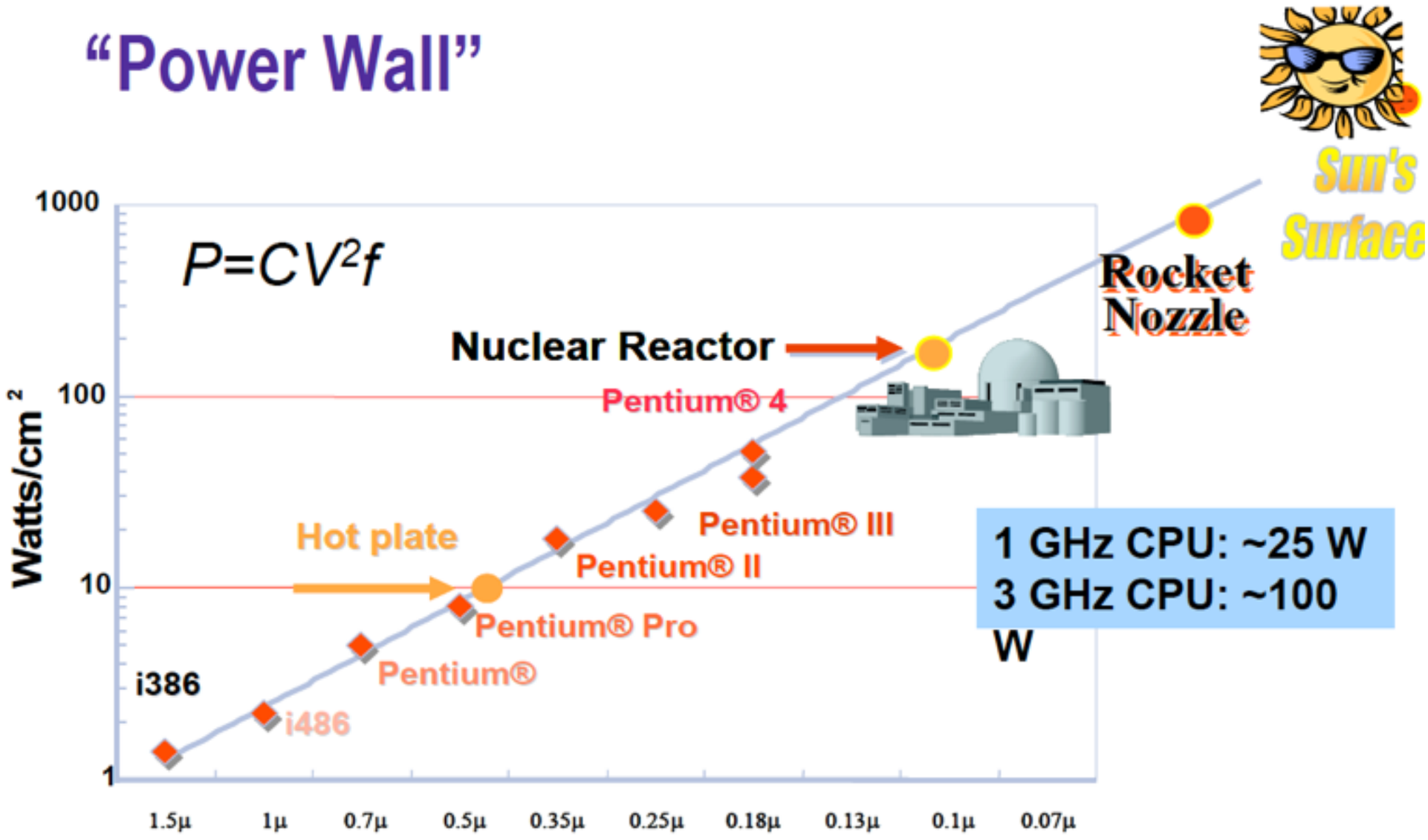


Beckton

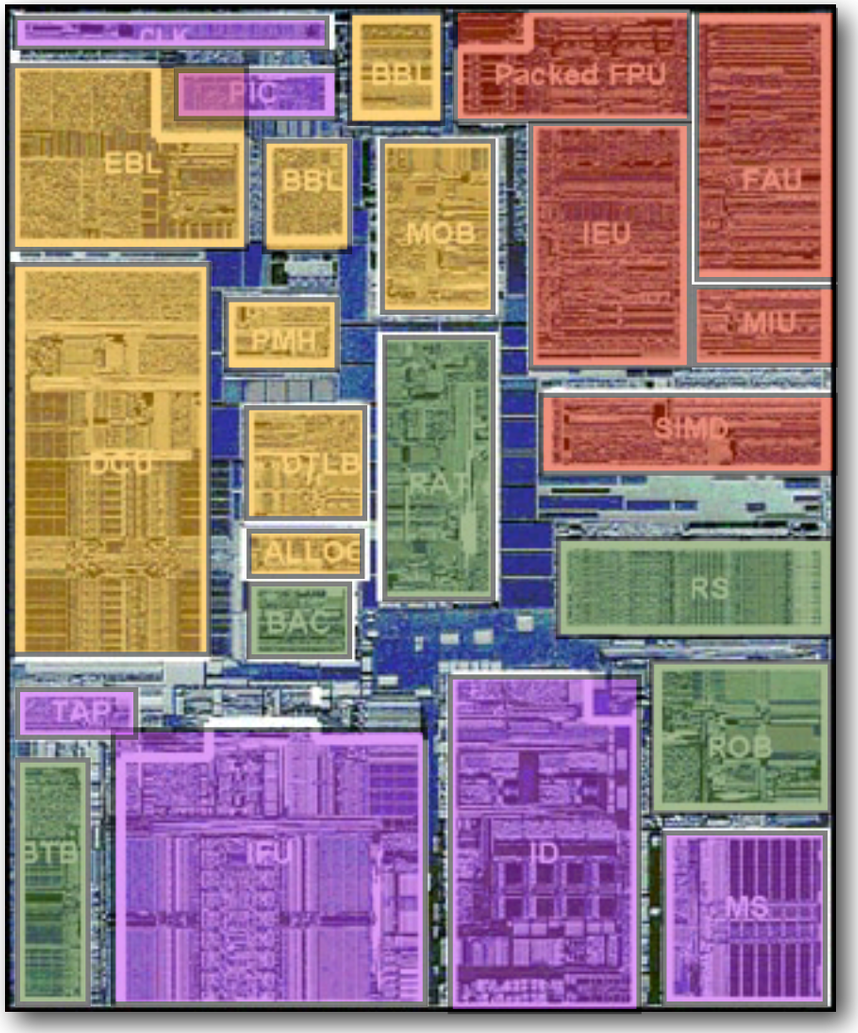




(3) 功耗大，效率低


"Power Wall"
$P=CV^2f$
Watts/cm^2
1000
100
10
1
Nuclear Reactor
Pentium® 4
Hot plate
Pentium® III
Pentium® II
Pentium® Pro
Pentium®
i386
i486
1 GHz CPU: ~25 W
3 GHz CPU: ~100 W
Sun's Surface
Rocket Nozzle
1.5µ
1µ
0.7µ
0.5µ
0.35µ
0.25µ
0.18µ
0.13µ
0.1µ
0.07µ

芯片效率



EBL/BBL - Bus logic, Front, Back
MOB - Memory Order Buffer
Packed FPU - MMX Fl. Pt. (SSE)
IEU - Integer Execution Unit
FAU - Fl. Pt. Arithmetic Unit
MIU - Memory Interface Unit
DCU - Data Cache Unit
PMH - Page Miss Handler
DTLB - Data TLB
BAC - Branch Address Calculator
RAT - Register Alias Table
SIMD - Packed Fl. Pt.
RS - Reservation Station
BTB - Branch Target Buffer
IFU - Instruction Fetch Unit (+I$)
ID - Instruction Decode
ROB - Reorder Buffer
MS - Micro-instruction Sequencer

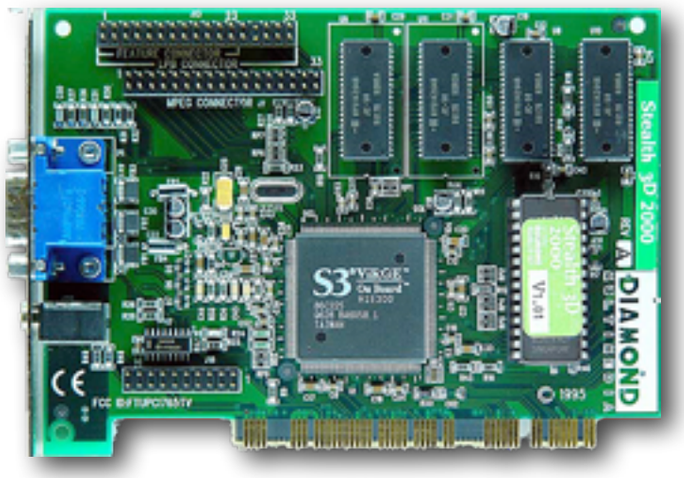
CPU发展面临的问题

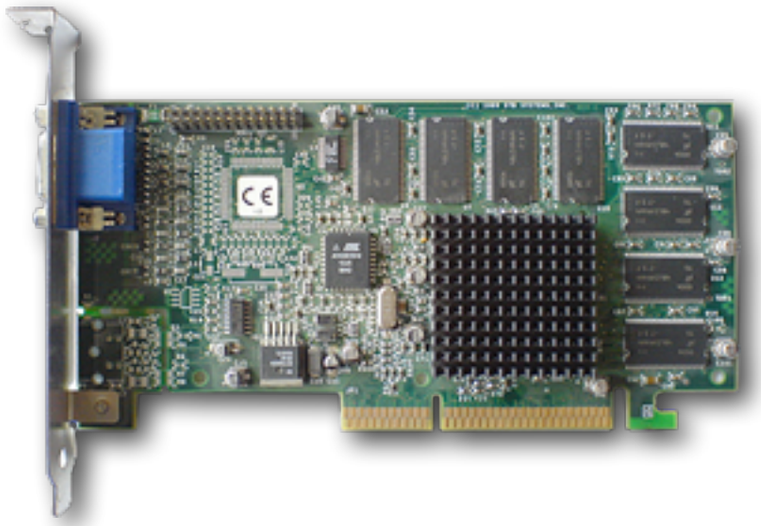
- 更多资源用于控制电路
- 更多资源用于缓存数据
- 较少资源用于实际运算
- 芯片资源效率低

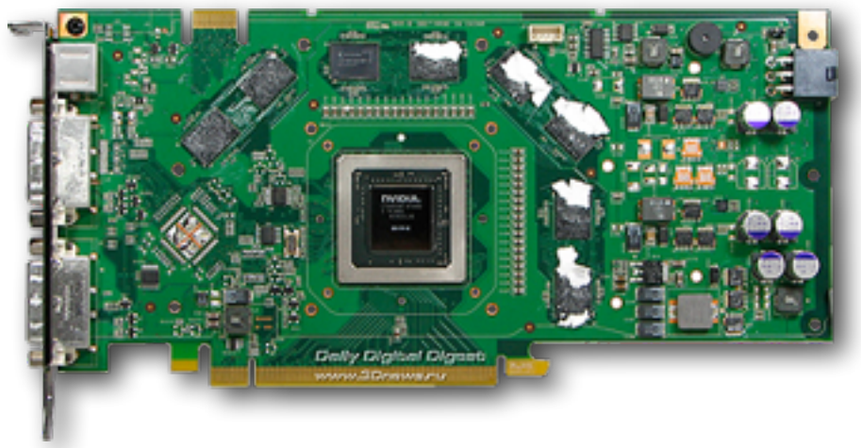
在另一方面 …

GPU也是一种处理器

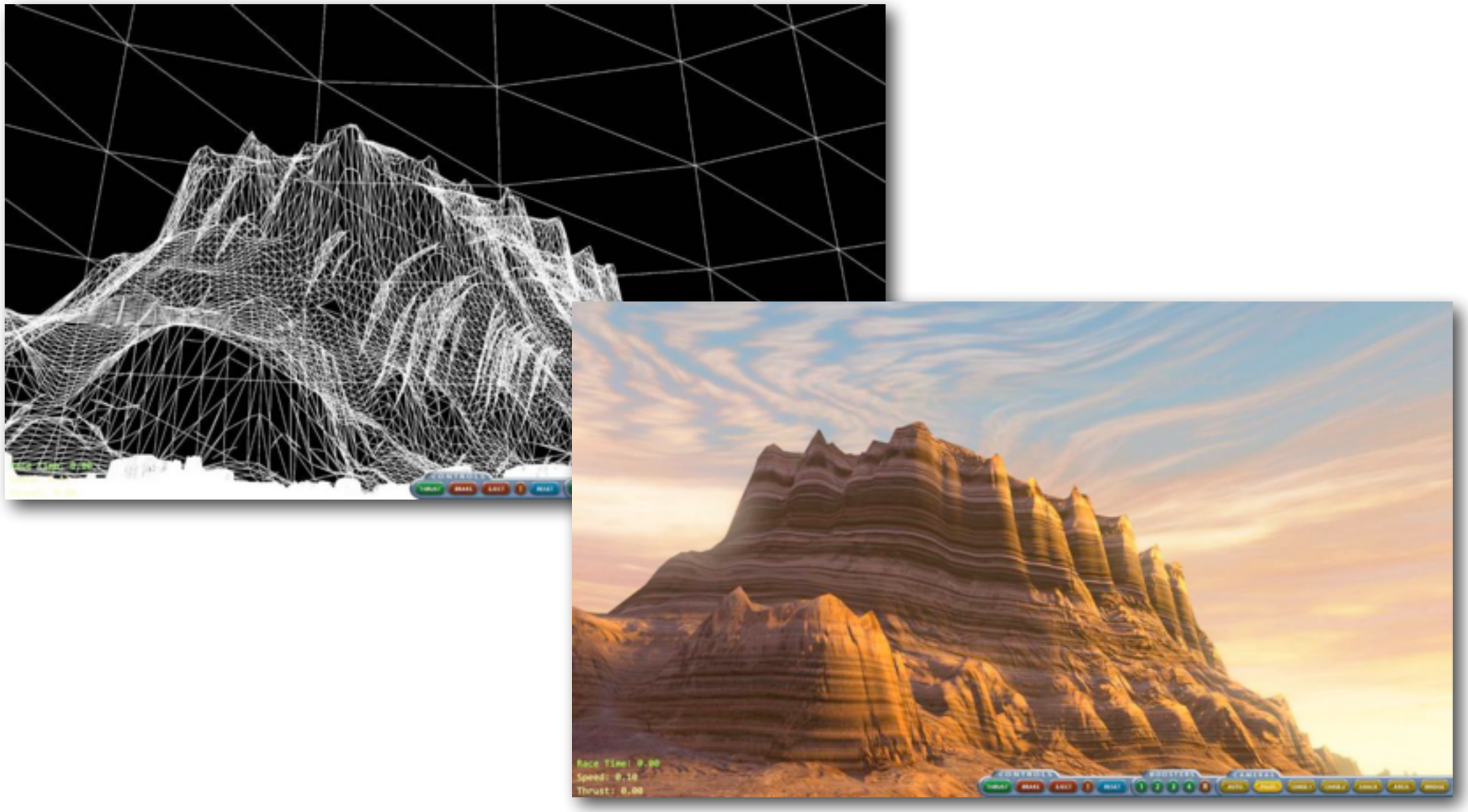
- GPU：Graphics Processing Unit
  - 针对图形应用
  - 作为扩展卡存在于计算机系统中







GPU的功用

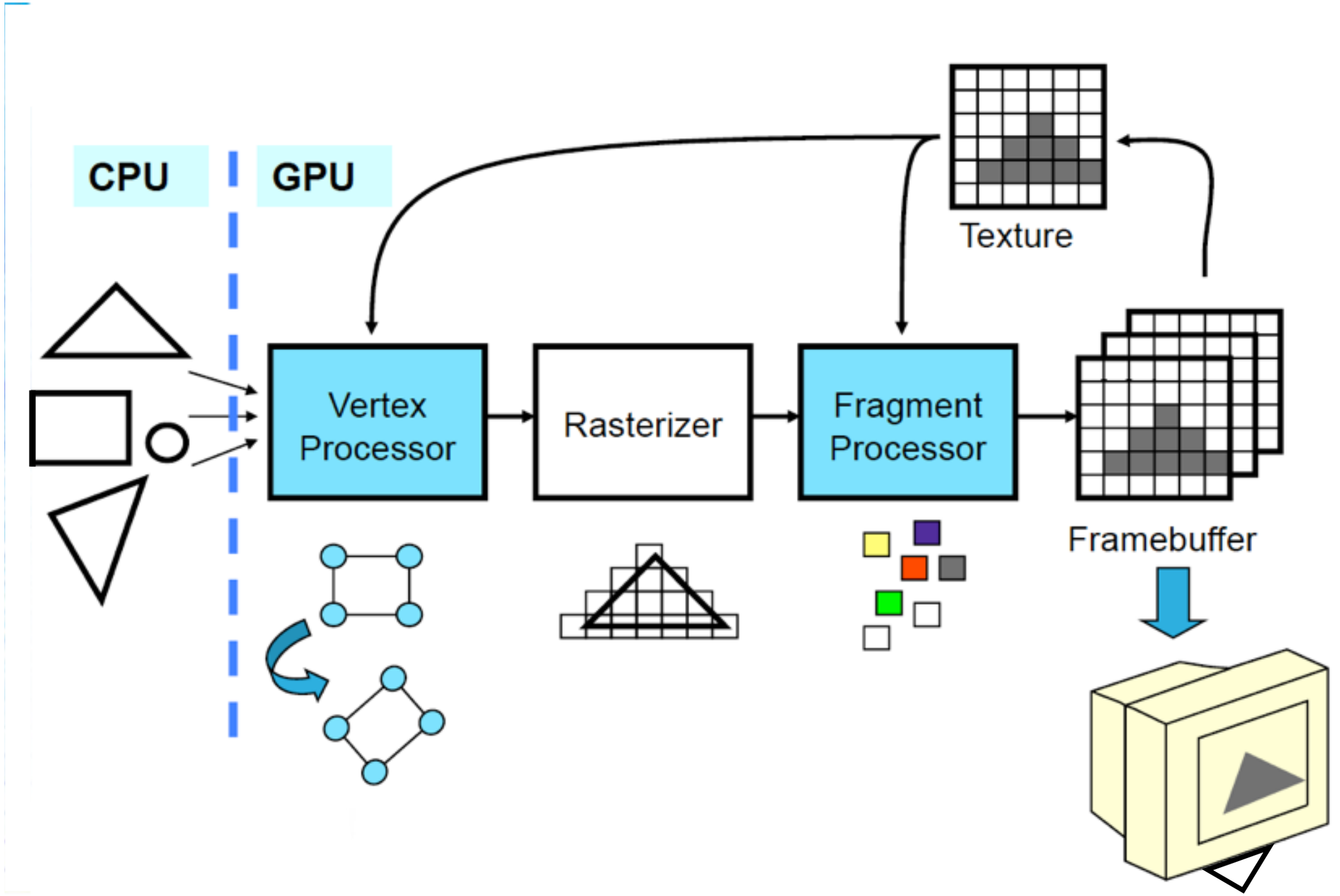


GPU 功能简介

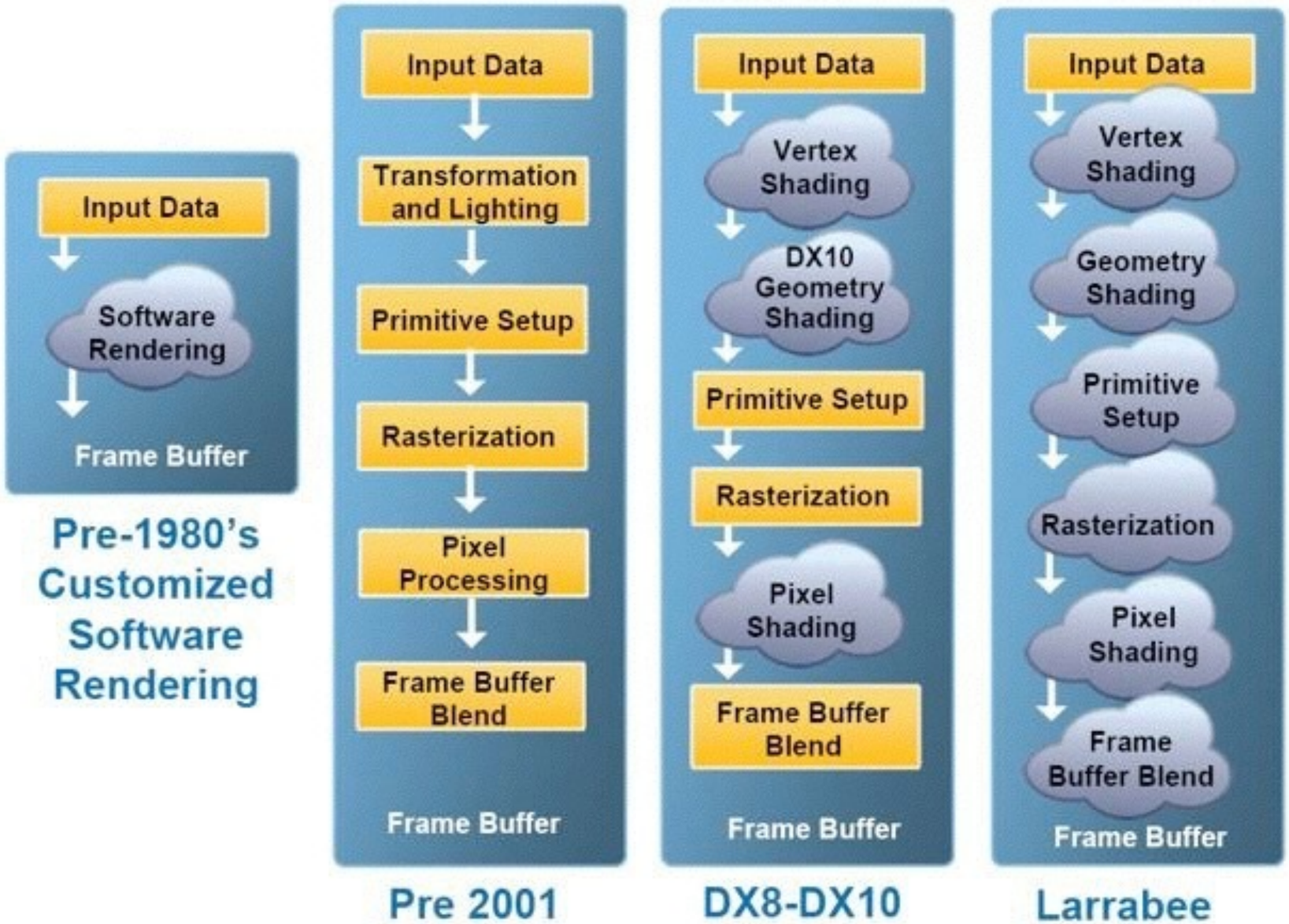
- 映射三维空间中的点/线/面到二维
- 施加光照操作
- 光栅化到像素空间
- 施加材质的影响
- 组合各个部分以输出

特点？大量简单的数据并行操作！

GPU结构与演化

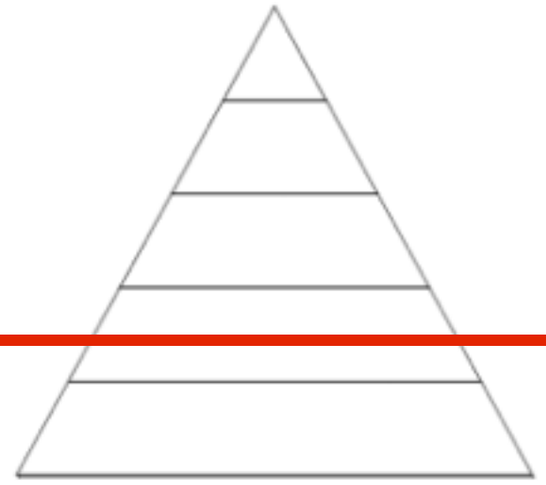

CPU
GPU
Texture
Vertex Processor
Rasterizer
Fragment Processor
Framebuffer

三维图形计算发展


Input Data
Software Rendering
Frame Buffer
Pre-1980's Customized Software Rendering
Input Data
Transformation and Lighting
Primitive Setup
Rasterization
Pixel Processing
Frame Buffer Blend
Frame Buffer
Pre 2001
Input Data
Vertex Shading
DX10 Geometry Shading
Primitive Setup
Rasterization
Pixel Shading
Frame Buffer Blend
Frame Buffer
DX8-DX10
Input Data
Vertex Shading
Geometry Shading
Primitive Setup
Rasterization
Pixel Shading
Frame Buffer Blend
Frame Buffer
Larrabee

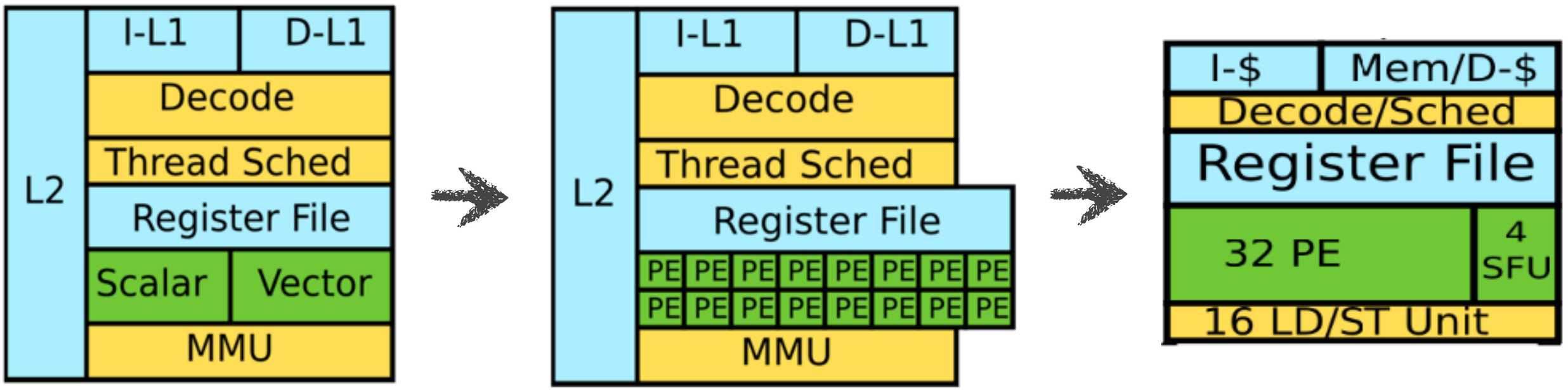
CPU与GPU-区别与联系

从CPU到GPU-1

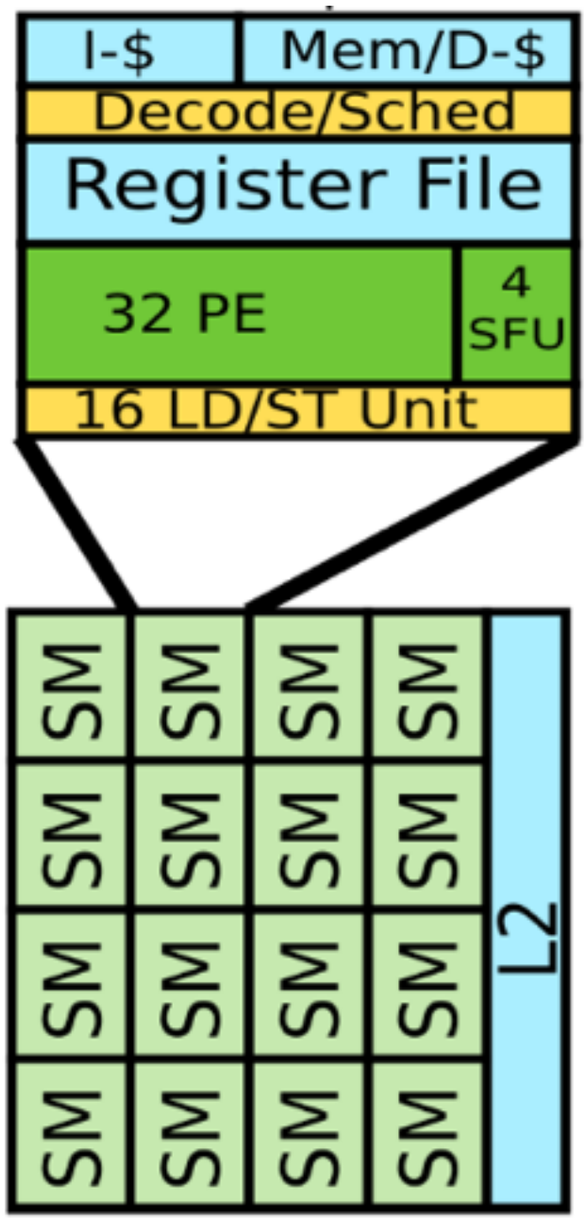


	Size (Byte)	Energy (pJ)	Delay (cycles)	Bandwidth (GB/s)
Reg	~~1K~~ 16K	~~10~~ 20	~~1~~ 2~3	1000
L1	32K	20	5	100
L2	256K	100	10	100
~~L3~~	~~8M~~	~~200~~	~~50~~	~~100~~
Off-chip	4G	2000	100	10

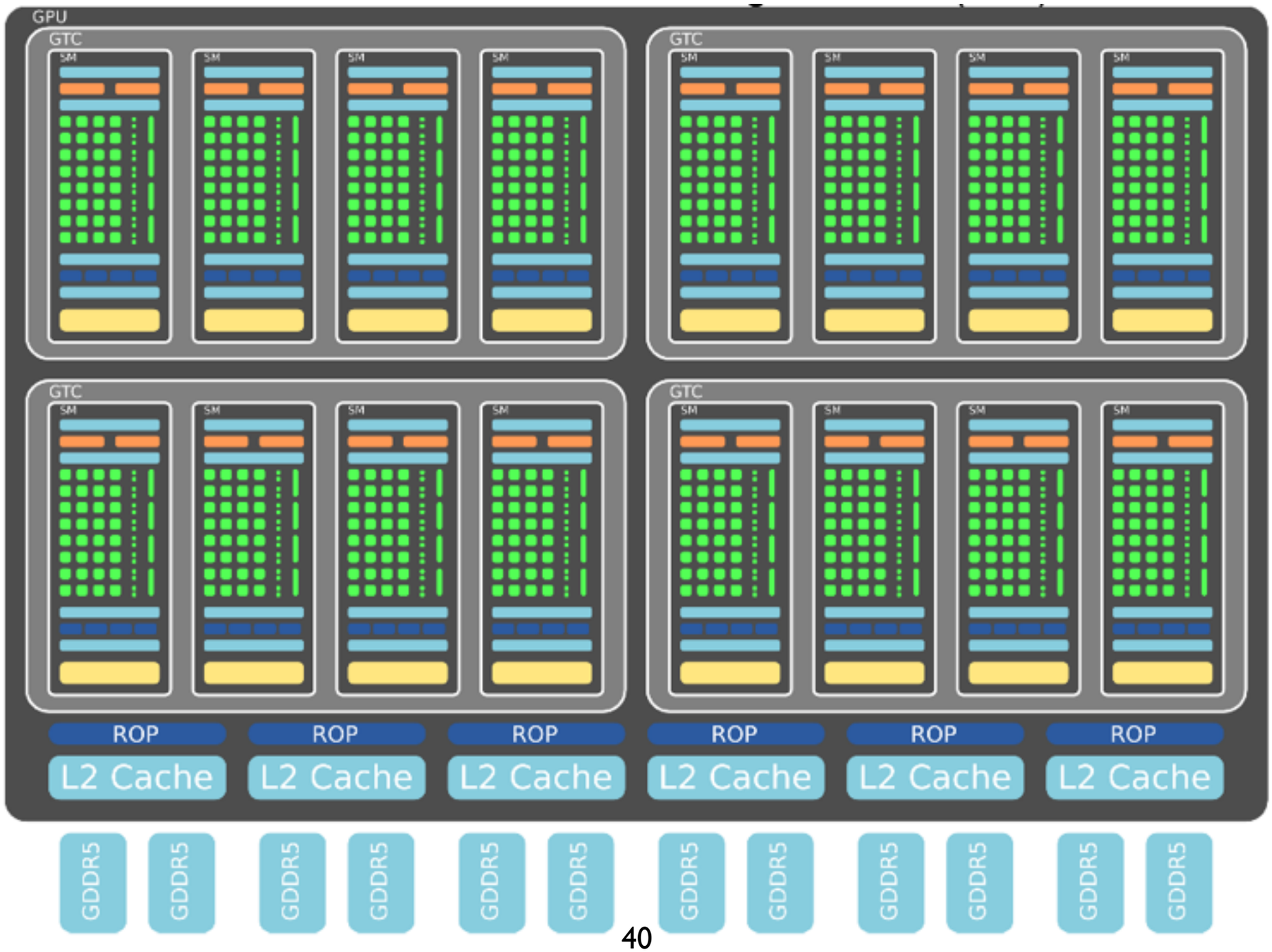
从CPU到GPU-2


I-L1
D-L1
Decode
Thread Sched
L2
Register File
Scalar
Vector
MMU
I-L1
D-L1
Decode
Thread Sched
L2
Register File
PE PE PE PE PE PE PE PE
PE PE PE PE PE PE PE PE
MMU
I-$
Mem/D-$
Decode/Sched
Register File
32 PE
4
SFU
16 LD/ST Unit

从CPU到GPU-3


I-$
Mem/D-$
Decode/Sched
Register File
32 PE
4
SFU
16 LD/ST Unit
SM
L2

GPU - NVIDIA GF100

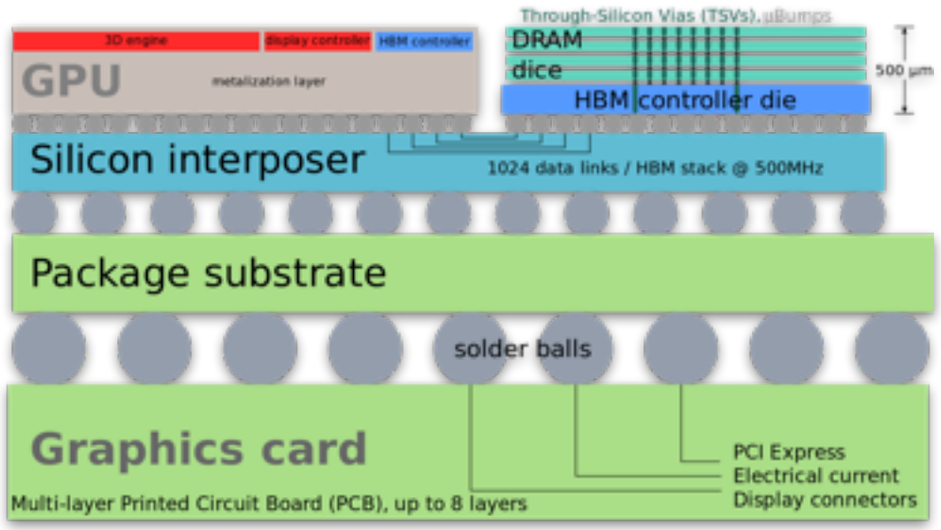


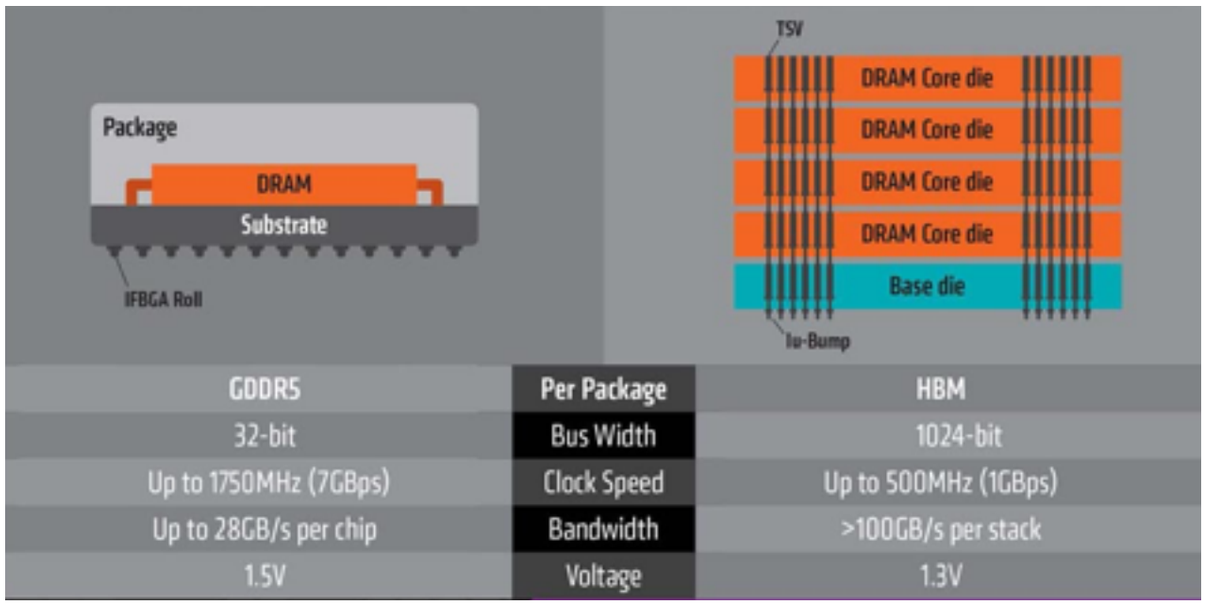
CPU-GPU 结构比较

	CPU	GPU
控制电路	指令级并行度	轻量级线程 快速切换
访存机制	多层Cache 一致性协议 容量巨大	Cache支持较弱 容量小
计算特性	真正用于计算的电路较少 SIMD支持	较多电路用于计算 轻量级线程以SIMD方式运行
优化目标	单个线程的执行效率	吞吐率，而非单个线程 的执行速度

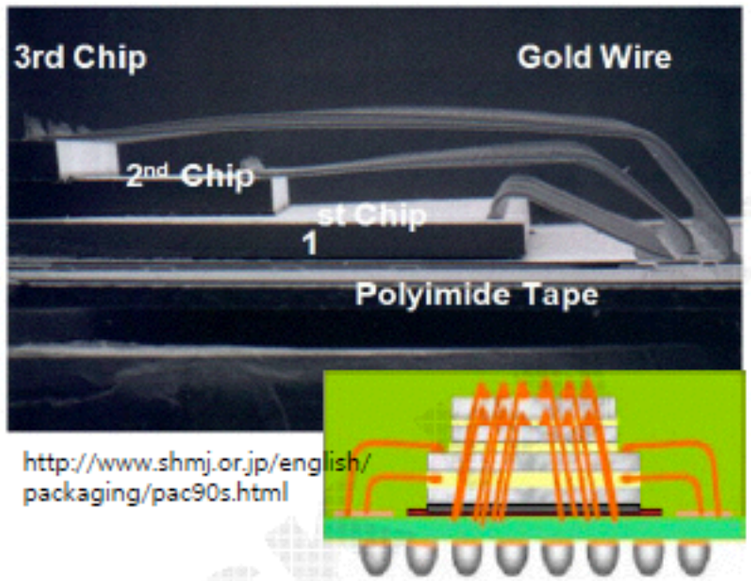
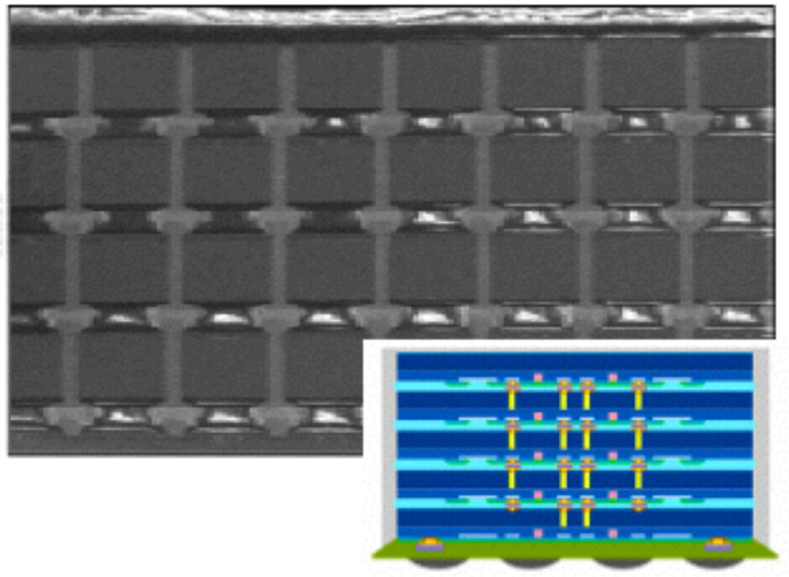
CPU与GPU的新发展

- 共同的趋势：
  - 使用HBM内存，提高访存特征
  - 使用简化的核心控制电路，提高并行度



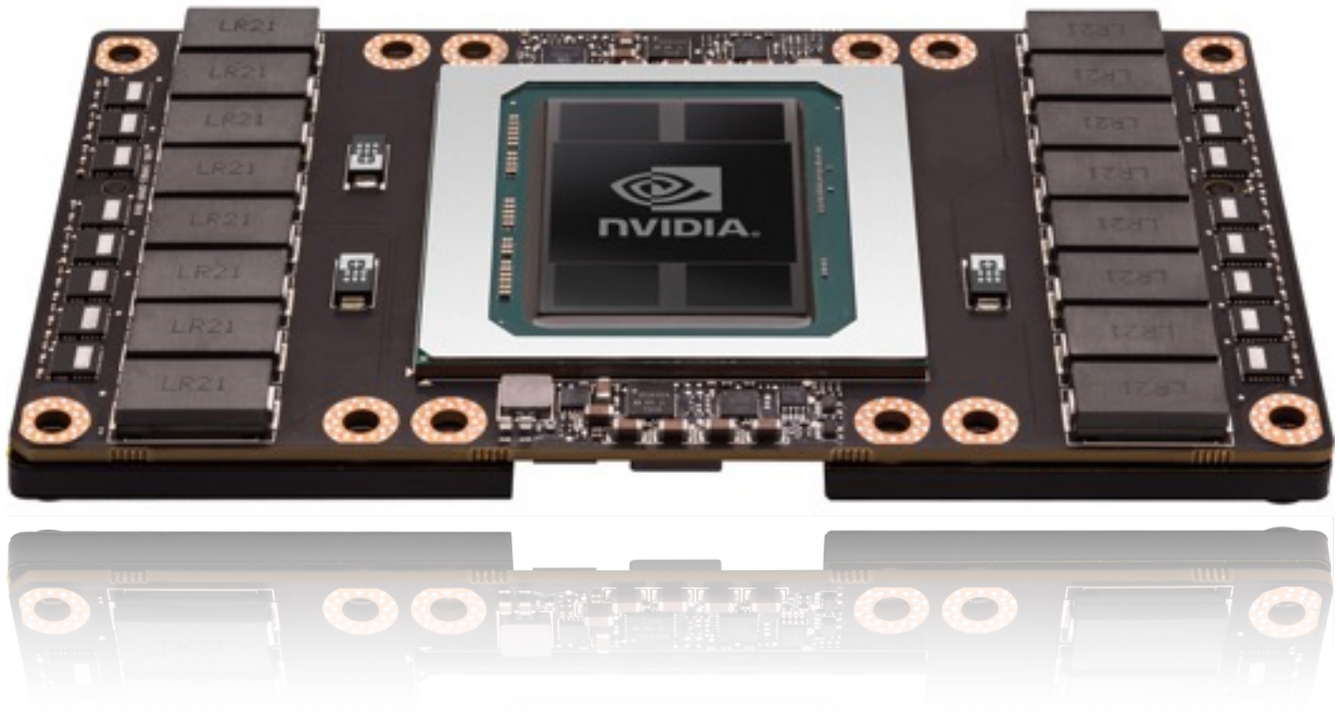


HBM is Promising

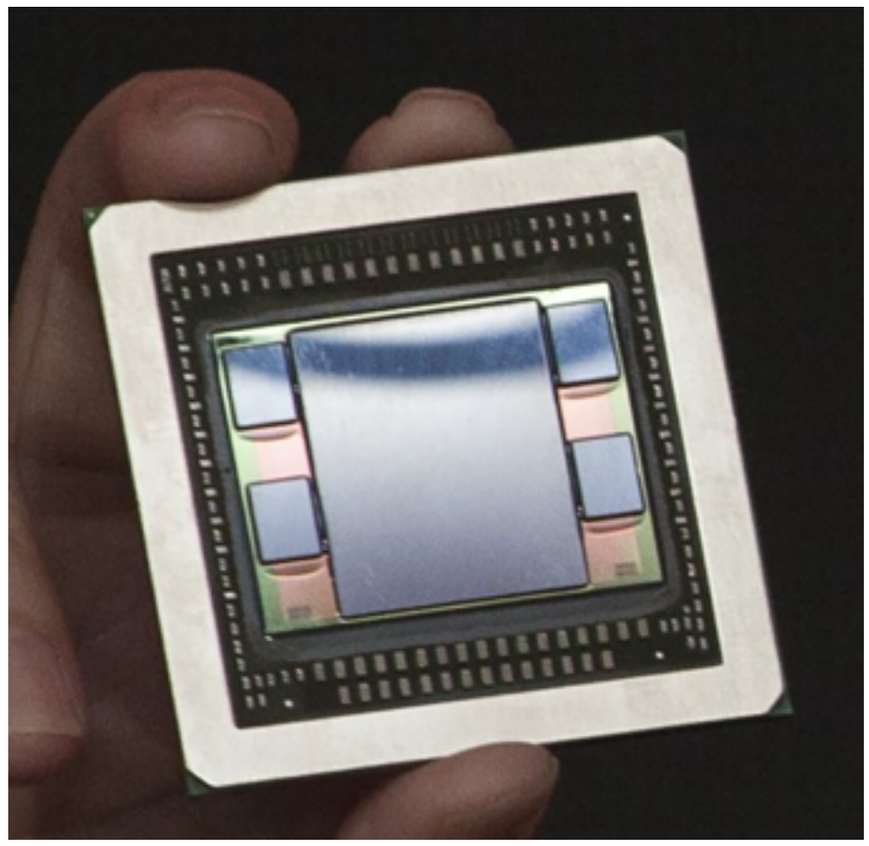
	Wire bonding - DDR3	TSV - HBM
Image		
PKG Size@die	100% (117mm^2)	36% (42mm^2)
mm^2@128GB/s	100% (3744mm^2)	11% (42mm^2)
Power Consumption* @128GB/s	100% (6.4W)	51% (3.3W)

[Kim & Kim, 2014]

GPUs - NV & AMD



NVIDIA GP100



AMD Fiji

Details on GP100

NVIDIA Tesla Family Specification Comparison				
	Tesla P100	Tesla K80	Tesla K40	Tesla M40
Stream Processors	3584	2 x 2496	2880	3072
Core Clock	1328MHz	562MHz	745MHz	948MHz
Boost Clock(s)	1480MHz	875MHz	810MHz, 875MHz	1114MHz
Memory Clock	1.4Gbps HBM2	5GHz GDDR5	6GHz GDDR5	6GHz GDDR5
Memory Bus Width	4096-bit	2 x 384-bit	384-bit	384-bit
Memory Bandwidth	720GB/sec	2 x 240GB/sec	288GB/sec	288GB/sec
VRAM	16GB	2 x 12GB	12GB	12GB
Half Precision	21.2 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	6.8 TFLOPS
Single Precision	10.6 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	6.8 TFLOPS
Double Precision	5.3 TFLOPS (1/2 rate)	2.91 TFLOPS (1/3 rate)	1.43 TFLOPS (1/3 rate)	213 GFLOPS (1/32 rate)
GPU	GP100 (610mm2)	GK210	GK110B	GM200
Transistor Count	15.3B	2 x 7.1B(?)	7.1B	8B
TDP	300W	300W	235W	250W
Cooling	N/A	Passive	Active/Passive	Passive
Manufacturing Process	TSMC 16nm FinFET	TSMC 28nm	TSMC 28nm	TSMC 28nm
Architecture	Pascal	Kepler	Kepler	Maxwell 2

from: www.anandtech.com

On Intel's Side

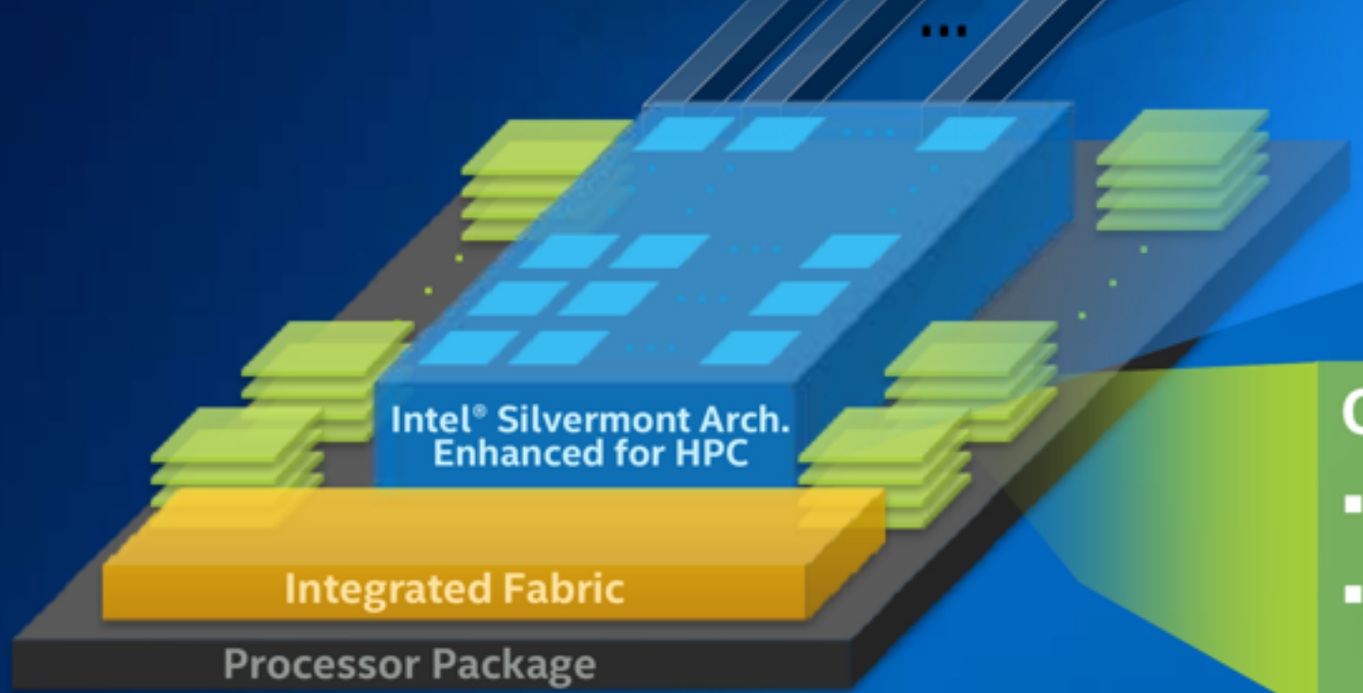
Unveiling Details of Knights Landing

(Next Generation Intel® Xeon Phi™ Products)

2nd half '15
1st commercial systems

3+ TFLOPS[1]
In One Package
Parallel Performance & Density

Platform Memory: DDR4 Bandwidth and Capacity Comparable to Intel® Xeon® Processors



Compute: Energy-efficient IA cores[2]

- *Microarchitecture enhanced for HPC[3]*
- ***3X** Single Thread Performance vs Knights Corner[4]*
- *Intel Xeon Processor Binary Compatible[5]*

On-Package Memory:

- *up to **16GB** at launch*
- ***1/3X** the Space[6]*
- ***5X** Bandwidth vs DDR4[7]*
- ***5X** Power Efficiency[6]*

Jointly Developed with Micron Technology

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. [1]Over 3 Teraflops of peak theoretical double-precision performance is preliminary and based on current expectations of cores, clock frequency and floating point operations per cycle. FLOPS = cores x clock frequency x floating-point operations per second per cycle. . [2]Modified version of Intel® Silvermont microarchitecture currently found in Intel® Atom™ processors. [3]Modifications include AVX512 and 4 threads/core support. [4]Projected peak theoretical single-thread performance relative to 1st Generation Intel® Xeon Phi™ Coprocessor 7120P (formerly codenamed Knights Corner). [5]Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX) . [6]Projected results based on internal Intel analysis of Knights Landing memory vs Knights Corner (GDDR5). [7]Projected result based on internal Intel analysis of STREAM benchmark using a Knights Landing processor with 16GB of ultra high-bandwidth versus DDR4 memory only with all channels populated.

intel

Conceptual—Not Actual Package Layout

GPU、加速计算、以及高性能计算

加速计算

- 加速计算（Accelerated Computing）
  - 基于流处理器（CELL，GPU等）
    - 往往用于某种特殊应用
  - 以扩展卡的方式存在于计算机系统中
  - 对CPU的计算能力起扩展和补充的作用

GPGPU与CUDA

- GPGPU
  - General-Purposed computing with GPU
    - 将GPU用于通用计算
    - 源于GPU可编程性逐渐提高的特征
    - 逐渐演变为HPC的潮流
- 流行的API：CUDA，OpenCL

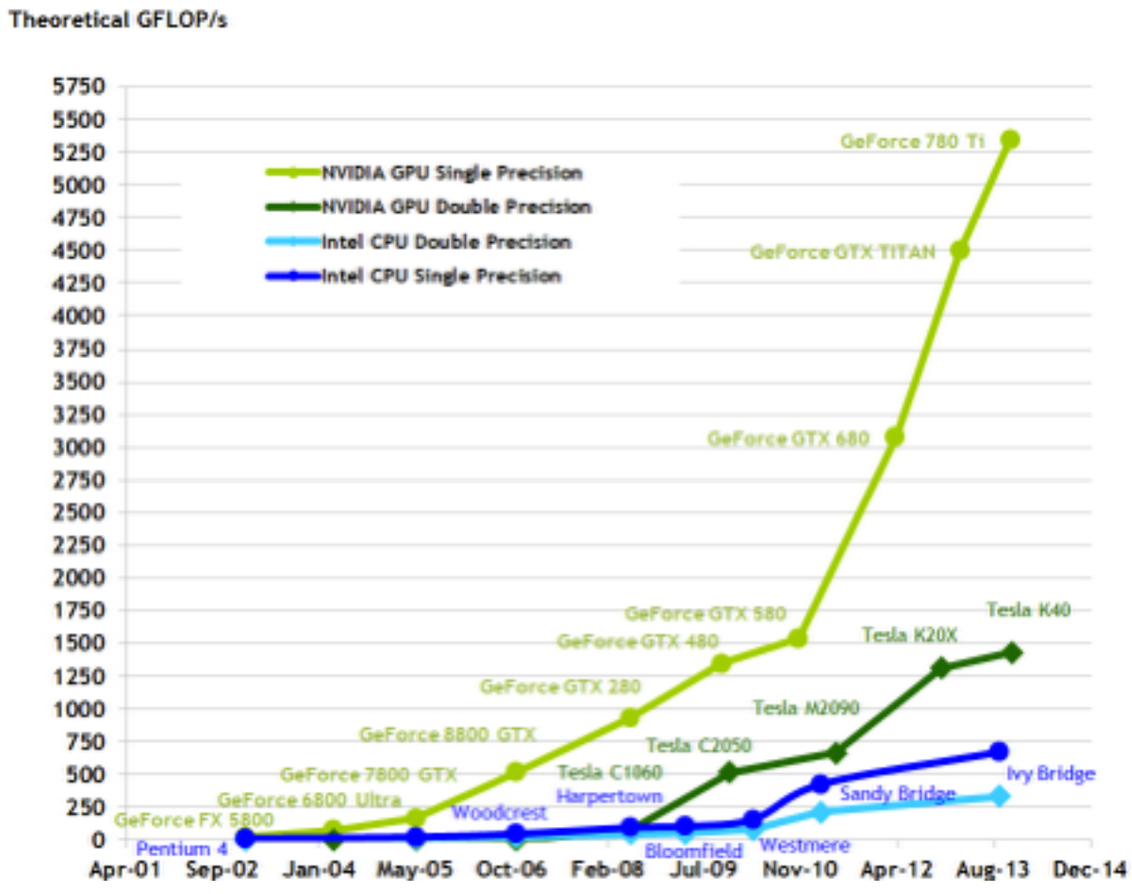
为什么GPGPU会流行?

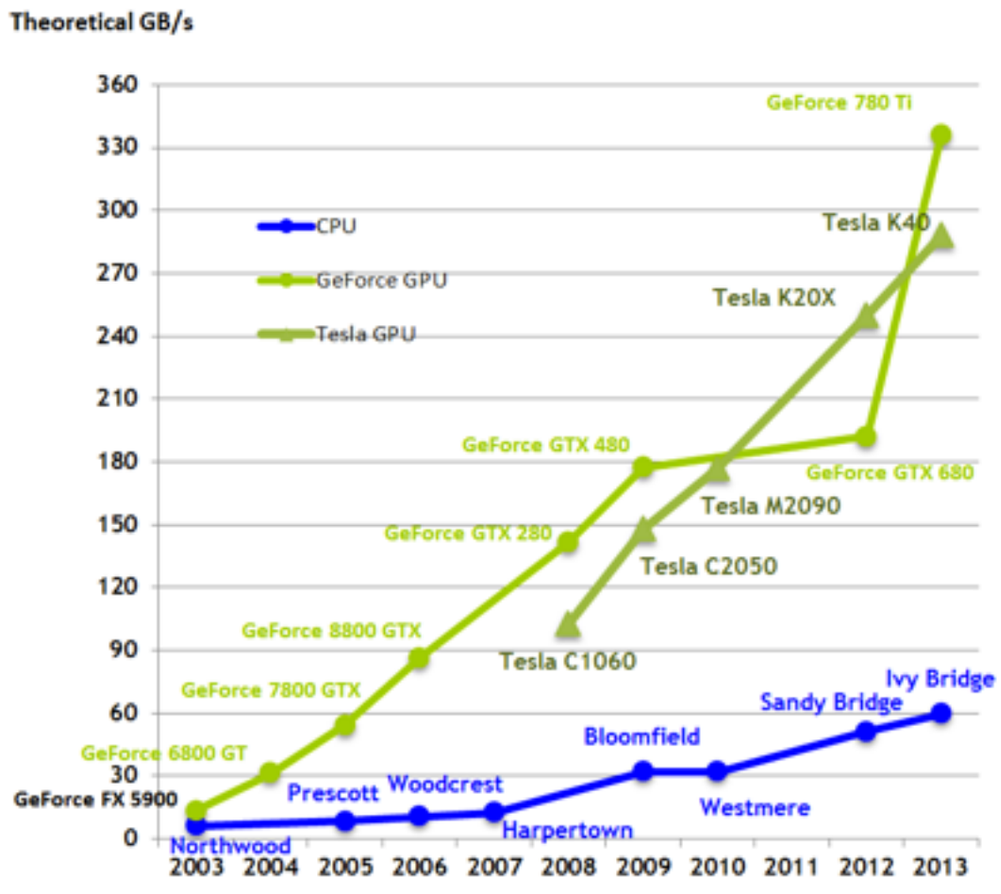
- 研发处理器开销巨大
  - Money， Time， Effort， Financial Risks
- 为什么GPGPU能快速发展?
  - 得益于广大游戏爱好者!
  - 更好的可编程性
  - 双精度支持

GPGPU 与 NVIDIA CUDA

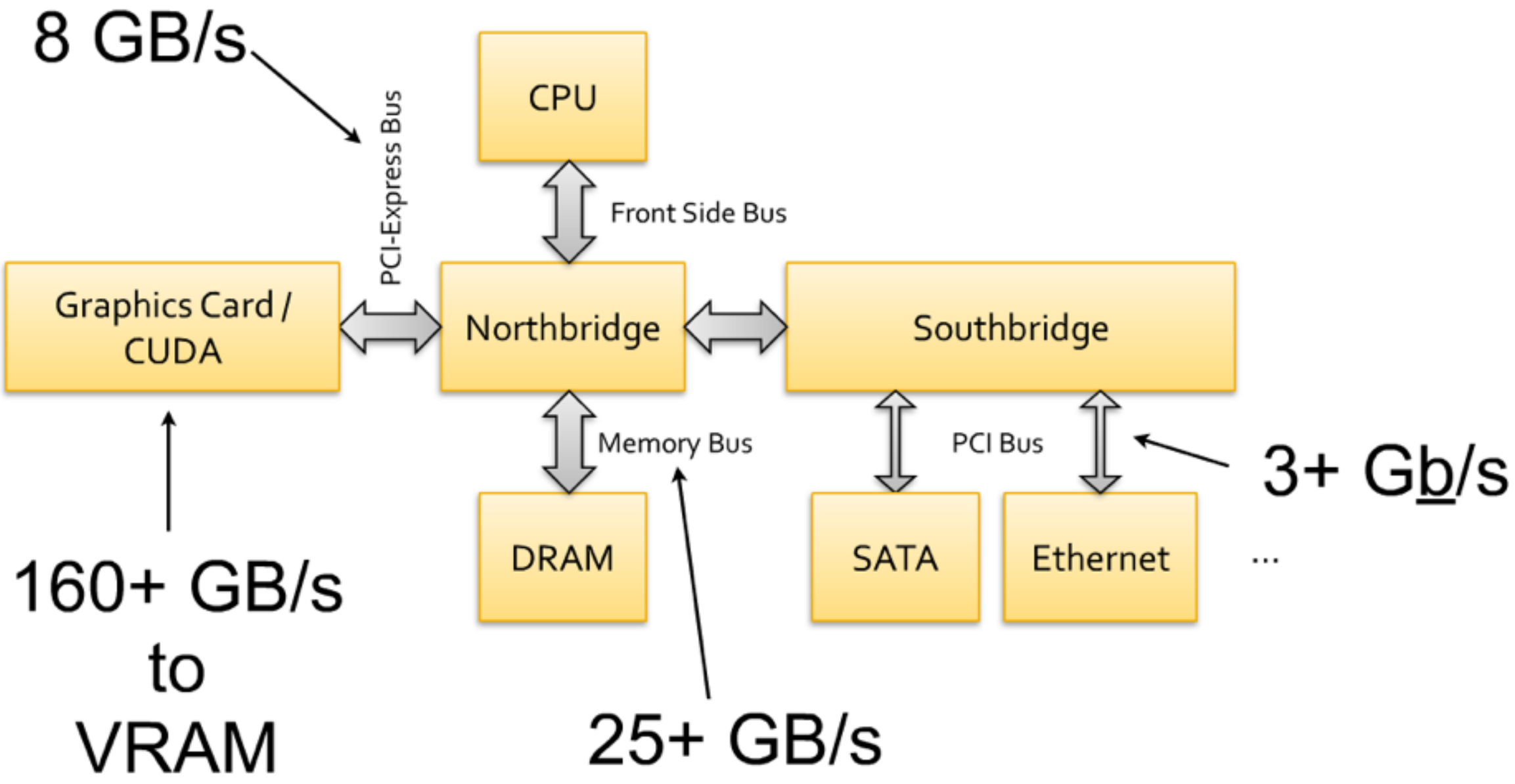
- NVIDIA CUDA 是当今最流行的 GPGPU 平台
  - CUDA：Compute Unified Device Architecture
  - 2008年5月：七千万台支持CUDA的GPU
  - 常见于：
    - TOP500排行榜（4 in top 20 @ 2013-11）
    - 个人超级计算机
  - http://www.nvidia.com/cuda

GPU 峰值速度发展

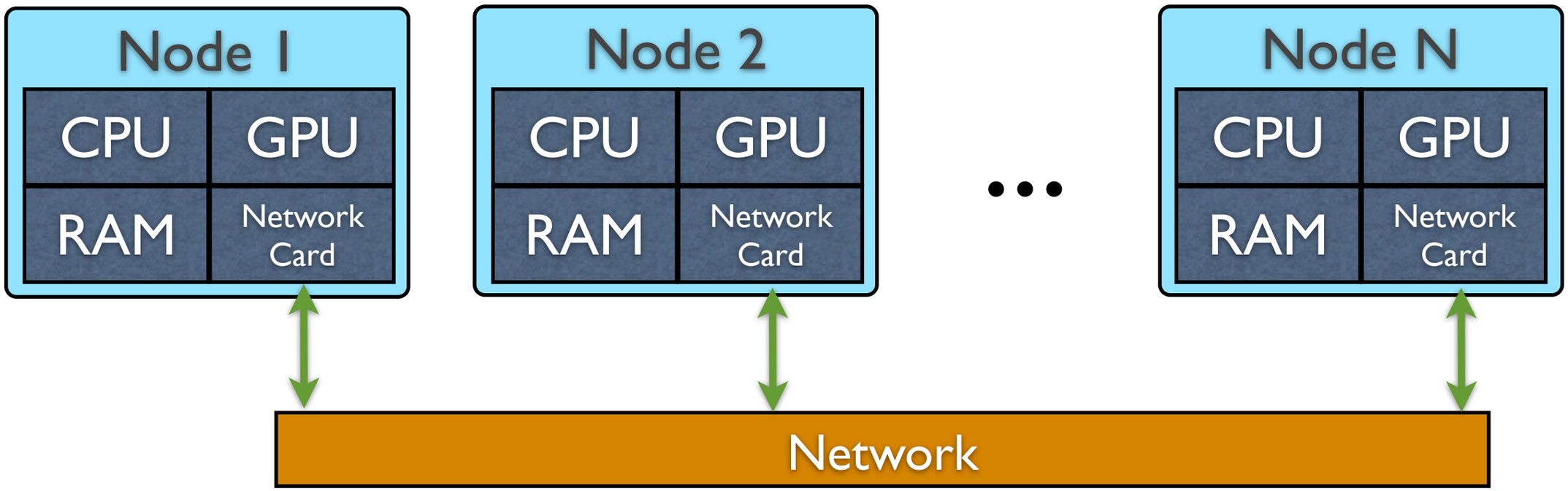

Theoretical GFLOP/s
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Double Precision
Intel CPU Single Precision
GeForce 780 Ti
GeForce GTX TITAN
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla K40
Tesla K20X
Tesla M2090
Tesla C2050
Tesla C1060
Harpertown
Woodcrest
Pentium 4
Bloomfield
Westmere
Sandy Bridge
Ivy Bridge
5750
5500
5250
5000
4750
4500
4250
4000
3750
3500
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
Apr-01
Sep-02
Jan-04
May-05
Oct-06
Feb-08
Jul-09
Nov-10
Apr-12
Aug-13
Dec-14


Theoretical GB/s
CPU
GeForce GPU
Tesla GPU
GeForce 780 Ti
Tesla K40
Tesla K20X
GeForce GTX 480
GeForce GTX 680
Tesla M2090
GeForce GTX 280
Tesla C2050
GeForce 8800 GTX
Tesla C1060
GeForce 7800 GTX
GeForce 6800 GT
GeForce FX 5900
Ivy Bridge
Sandy Bridge
Bloomfield
Prescott
Woodcrest
Westmere
Harpertown
Northwood
360
330
300
270
240
210
180
150
120
90
60
30
0
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013

GPU与计算机系统


8 GB/s
PCI-Express Bus
CPU
Front Side Bus
Graphics Card / CUDA
Northbridge
Southbridge
Memory Bus
PCI Bus
3+ Gb/s
DRAM
SATA
Ethernet
...
160+ GB/s to VRAM
25+ GB/s

基于GPU的集群



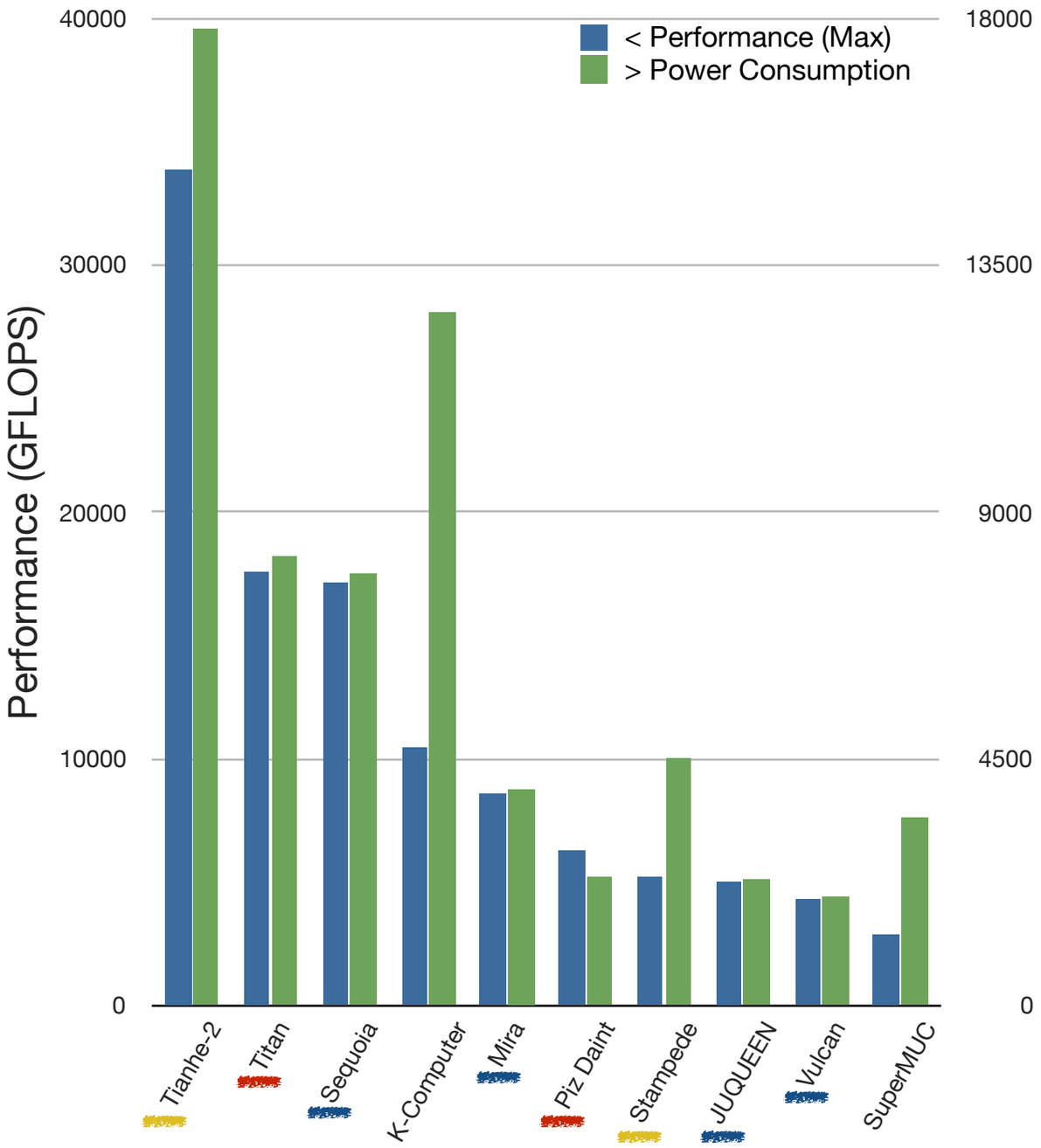
	GPU	Remote Node
Latency	<5 us	~1 us (MPI)
Peak Bandwidth	8 GB/s	12 GB/s

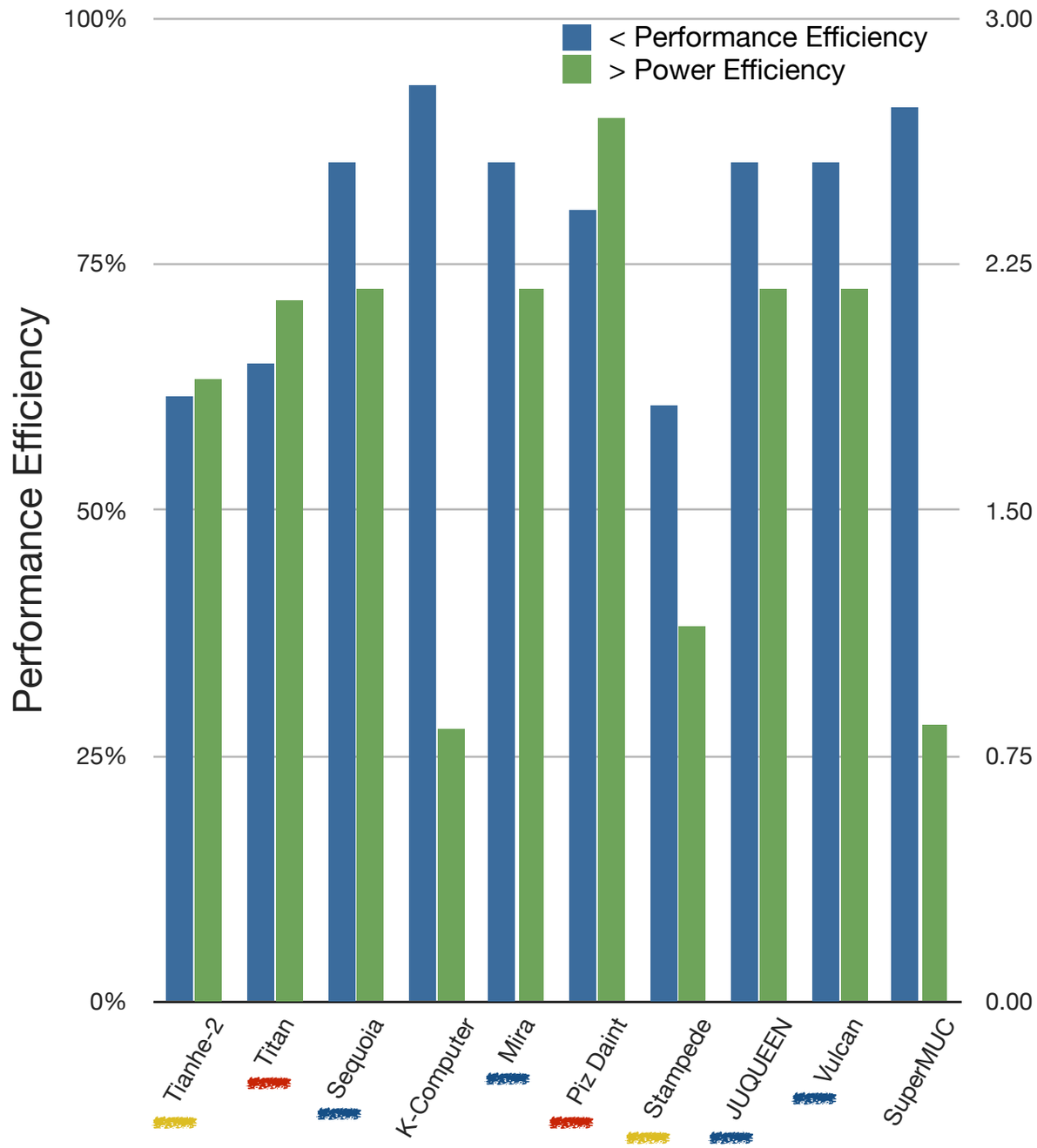
- GPU： PCI-E gen 2.0 16x. Remote Node: Infiniband QDR 12x

Top500 (2013-11)

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
9	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	5,033.2	1,972
10	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147,456	2,897.0	3,185.1	3,423

Top500 (2013-11) - 2





GPU集群的潜在性能问题

- 访问GPU与访问远程节点特性类似
  - 延时与带宽均在同一数量级
- 制约GPU集群发挥全部性能的一大瓶颈

- 解决方法：
  - 通过算法/程序设计，CPU与GPU协同计算
  - 新一代PCI-E技术（带宽翻倍、延时降低）

个人超级计算机

- 单/多GPU计算节点
  - 桌面系统
- >1TFLOPS性能
  - Tesla C2050: >500G DP FLOPS
  - K20X: ~1.3 T DP FLOPTS

大纲

- 计算机系统与加速计算
- GPU的发展与GPGPU的兴起
- NVIDA CUDA编程与优化
- GPU加速的局限性

NVIDIA CUDA简介

- 基本功能：
  - 将NVIDIA的GPU进行抽象
  - 细粒度多线程并行计算平台
  - 为编程人员提供编程接口和工具

语言绑定与API

CUDA Toolkit

Provides a comprehensive environment for C/C++ developers building GPU-accelerated applications

OpenACC

Directives for parallel computing, is a new open parallel programming standard designed to enable all scientific and technical programmers.

PGI Accelerator Fortran and C Compilers

Accelerate applications on GPU platforms by adding compiler directives to existing code.

The PGI CUDA C/C++ compiler for x86

Compile and optimize their CUDA applications to run on x86-based workstations, servers and clusters.

CUDA FORTRAN

Enjoy GPU acceleration directly from your Fortran program using CUDA Fortran from The Portland Group.

Anaconda Accelerate

Enables acceleration on your GPU or multi-core processor using Python.

PyCUDA

Gives you access to CUDA functionality from your Python code

Altimesh Hybridizer™

An advanced productivity tool that generates vectorized C++ (AVX) and CUDA C code from .NET assemblies (MSIL) or Java archives (bytecode)

OpenCL™

OpenCL is a low-level API for GPU computing that can run on CUDA-powered GPUs.

Alea GPU

This is a novel approach to develop GPU applications on .NET, combining the CUDA with Microsoft's F#.

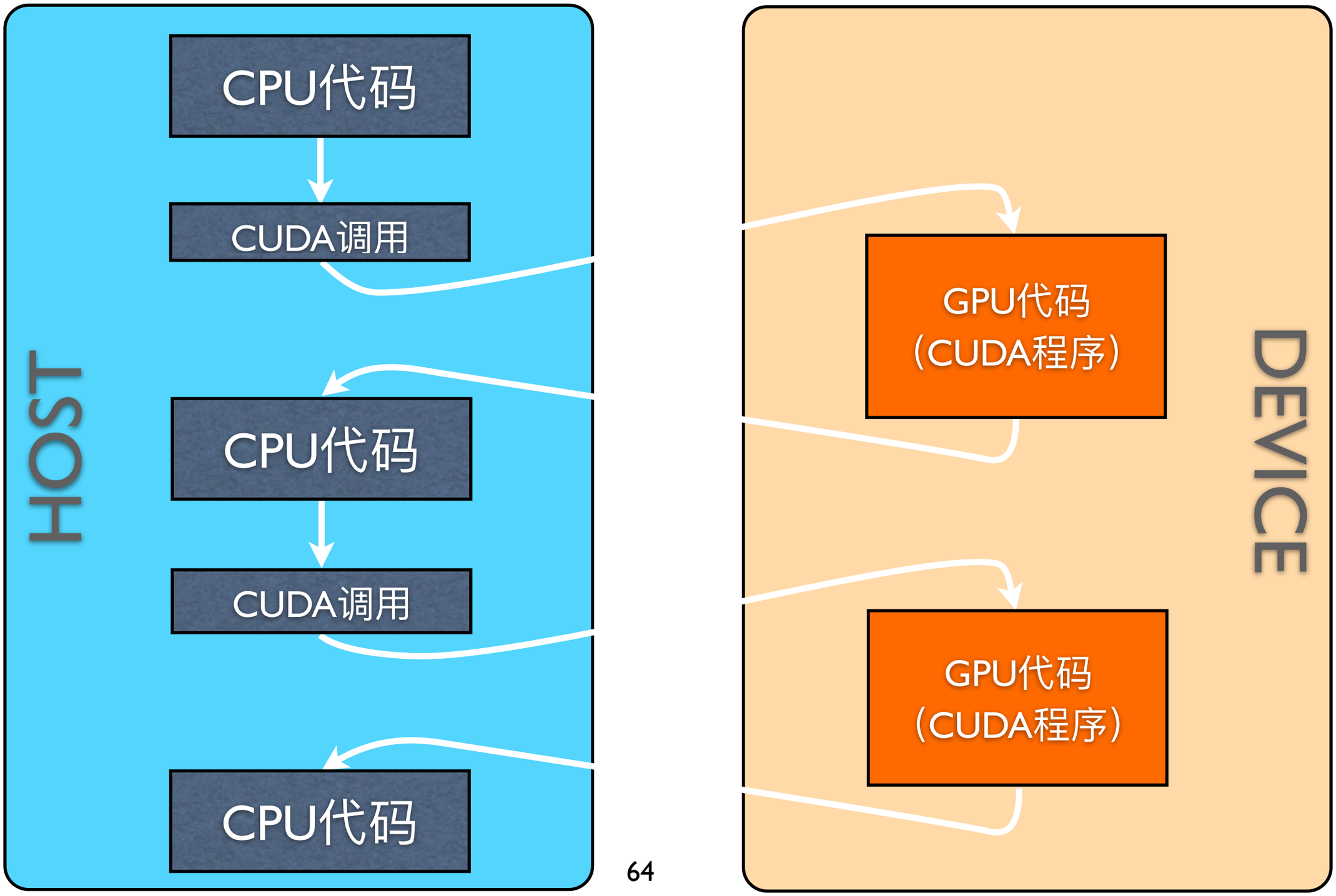
CUDA下载与安装

- 最新版本：7.5
  - https://developer.nvidia.com/cuda-toolkit
- 支持Windows，Linux，Mac
- 驱动+工具包+SDK
- 语言支持：C/C++, FORTRAN, Python

NVIDIA CUDA 编程

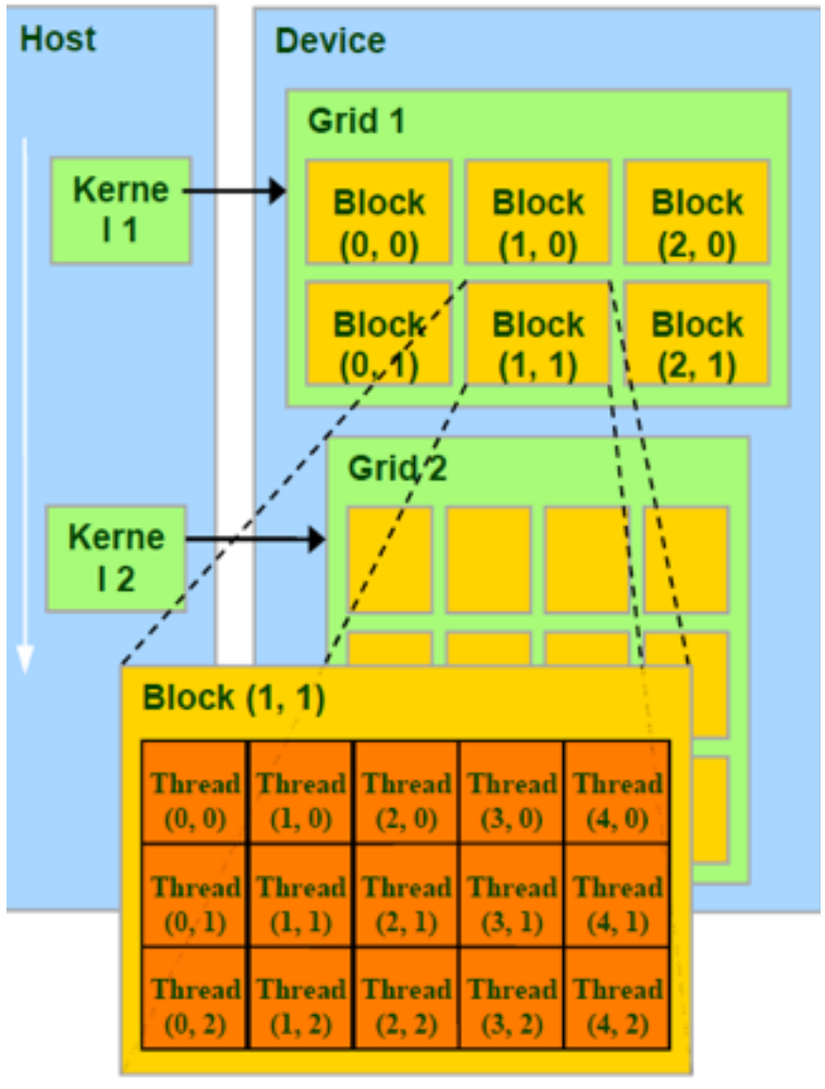
- 运行模型、存储资源、编程接口
- 性能优化
  - 明确优化目标
  - 避免潜在的性能瓶颈

CUDA运行模型


CPU代码
CUDA调用
CPU代码
CUDA调用
CPU代码
HOST
GPU代码
（CUDA程序）
GPU代码
（CUDA程序）
DEVICE

CUDA运行模型-2

- 多线程层级结构
  - Thread：
    - 最小的计算单元
  - Thread Block：
    - Thread组成的三维结构
  - Grid
    - Thread Block组成的二维结构



CUDA运行模型-3

- 用户指定内容：
  - 每个线程的工作
  - 线程、线程块、网格的组织方式
  - 线程块内部各线程间的互动（同步、等）
- 暗含语义：
  - 每个线程的工作是相似/相同的
  - 不同线程具体操作的数据不同

CUDA运行模型-4

- 线程块间无序开始，无序完成
- 线程块内各线程无序开始，无序完成
  - 除非有同步操作

线程的身份

- 内建变量：
  - gridDim 和 blockIdx：
    - 类型： dim3 和 uint3
  - blockDim 和 threadIdx：
    - 类型： dim3 和 uint3

CUDA程序举例 (1)

- SAXPY: y = alpha * x + y

```
void saxpy(       float * y,
           const float * x,
           const float alpha,
           const int n )
{
    for ( int i = 0; i < n; i += 1 )
    {
        y[i] += alpha * x[i];
    }
}
```

```
__global__
void saxpy_gpu(       float * d_y,
                  const float * d_x,
                  const float alpha,
                  const int n )
{
    const int i =
       blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        d_y[i] += alpha * d_x[i];
}
```

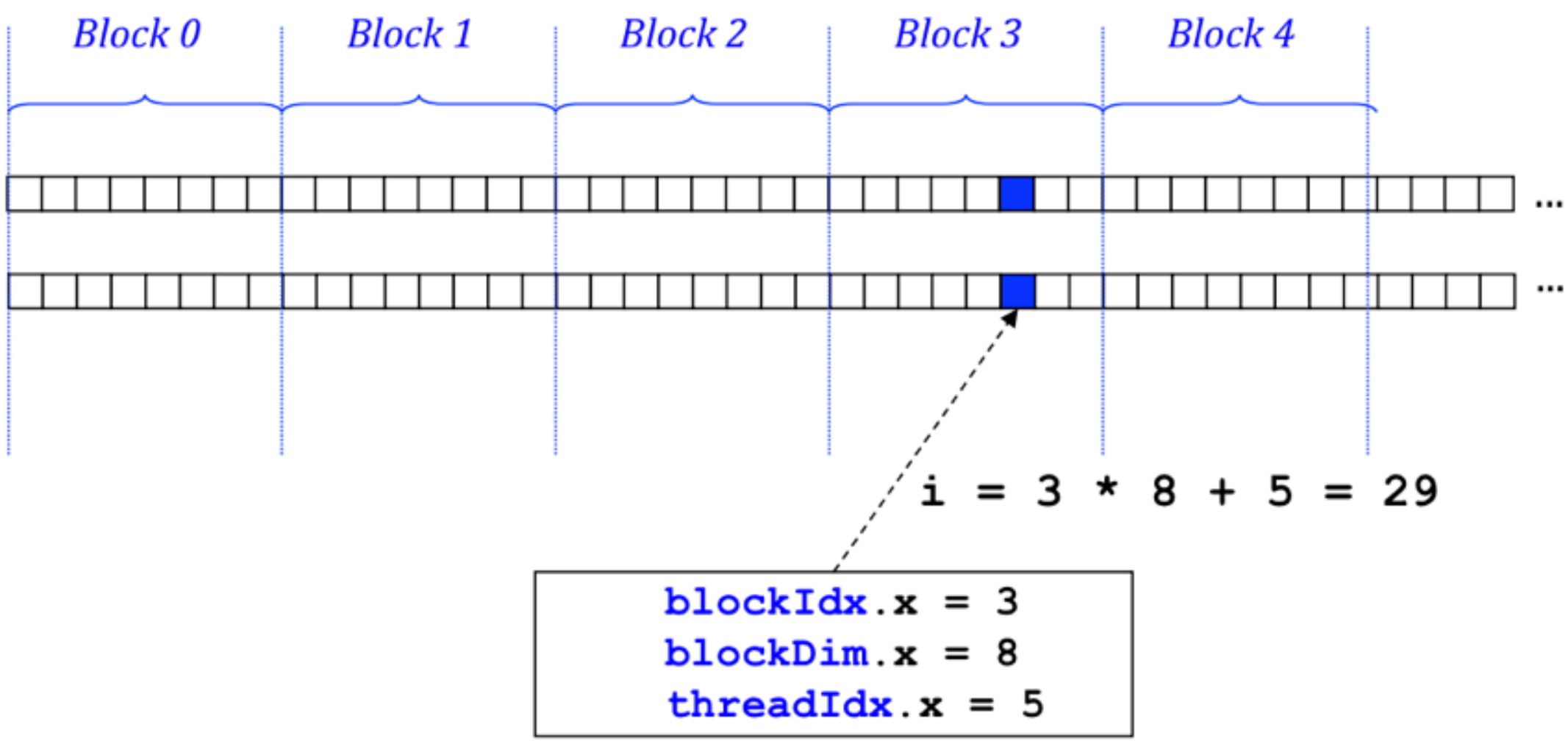
原理：一个线程负责一个元素的操作

CUDA程序举例 (1)

- 线程组织：
  - 每个Block只有一维
  - 每个Grid内只有一维
- 如何调用saxpy这个函数?

```
const int n = 10000000;
const float alpha = 0.5f;

float *d_x, *d_y;
// initialize d_x, d_y

const int blockSize = 512;
const int numBlocks = (n-1) / blockSize + 1;

saxpy_gpu <<<numBlocks, blockSize>>>(d_y, d_x, alpha, n);
```

CUDA程序举例 (I)


Block 0
Block 1
Block 2
Block 3
Block 4
...
i = 3 * 8 + 5 = 29
blockIdx.x = 3
blockDim.x = 8
threadIdx.x = 5

CUDA程序举例 (1)

- SAXPY: y = alpha * x + y

```
const int n = 10000000;
const float alpha = 0.5f;

float *d_x, *d_y;
// initialize d_x, d_y

const int blockSize = 512;
const int numBlocks = (n-1) / blockSize + 1;

saxpy_gpu <<<numBlocks, blockSize>>>(d_y, d_x, alpha, n);
```

实际产生：10000384个线程

CUDA程序举例 (1)

- SAXPY: y = alpha * x + y

```
const int n = 10000000;
const float alpha = 0.5f;

float *d_x, *d_y;
// initialize d_x,
```

```
const int blockSiz
const int numBlock

saxpy_gpu <<<numBl
```

```
__global__
void saxpy_gpu(       float * d_y,
               const float * d_x,
               const float alpha,
               const int n )
{
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        d_y[i] += alpha * d_x[i];
}
```

实际产生：10000384个线程

CUDA程序举例 (2)

- 矩阵操作： A=alpha*B+A
  - A, B均为NxN的单精度浮点矩阵
- 线程组织方式：
  - 二维格式的线程块（Block）
    - 每个线程块处理大小为MxM的块
  - 二维格式的网格（Grid）
    - 网格组织为⌈N/M⌉ x ⌈N/M⌉ 个块

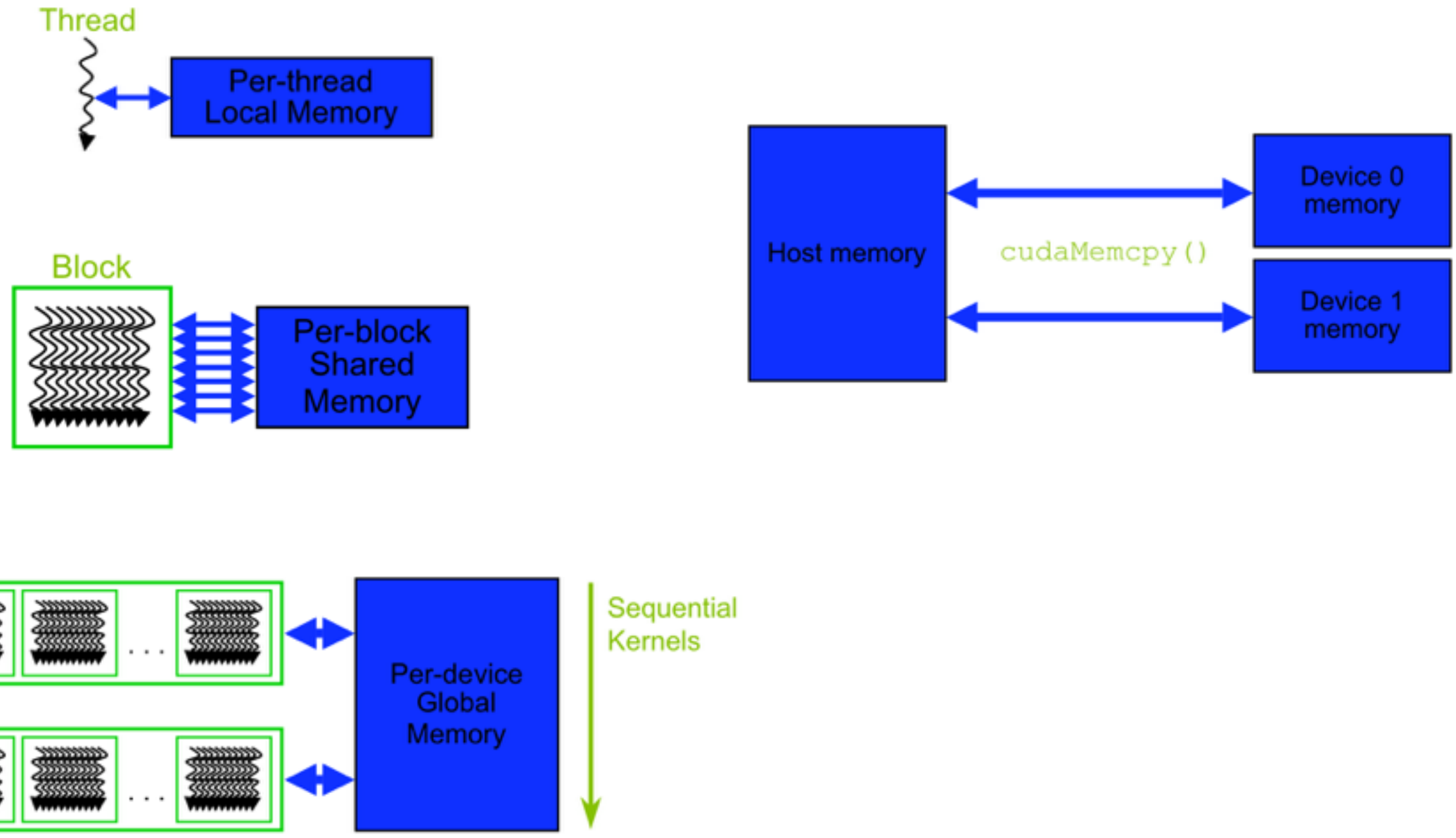
CUDA程序举例 (2)

```
__global__
void saxpy_mat_gpu(       float[N][N] d_A,
                    const float[N][N] d_B,
                    const float alpha,
                    const int n )
{
    const int row = blockIdx.x * blockDim.x + threadIdx.x;
    const int col = blockIdx.y * blockDim.y + threadIdx.y;

    if ( ( row < n ) && ( col < n ) )
        d_A[row][col] += alpha * d_B[row][col];
}
```

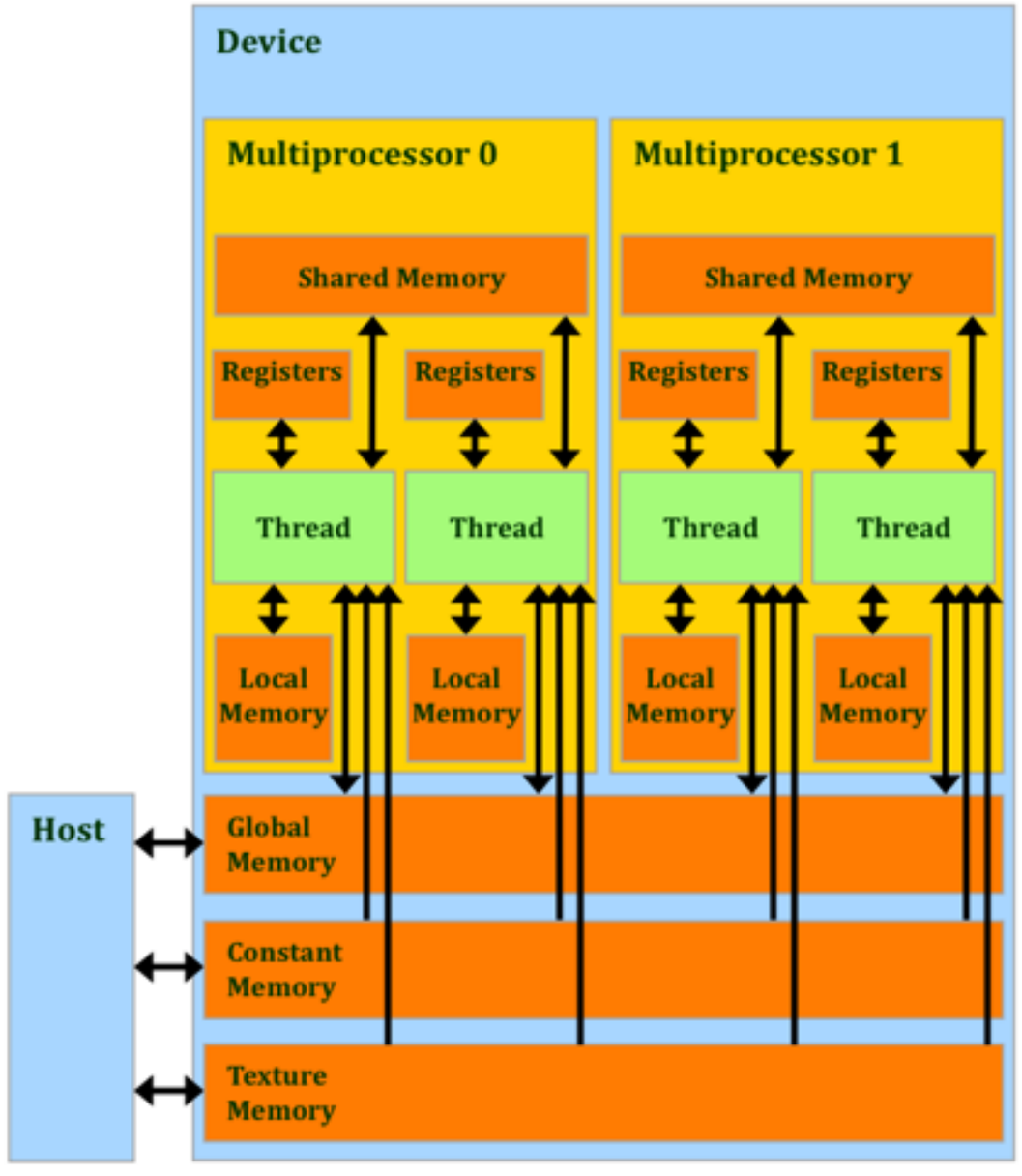
CUDA程序举例 (2)

```
#define N 4000
#define M 16

const float alpha = 0.5f;

float d_A[N][N], d_B[N][N];
// initialize d_x, d_y

dim3 blockSize( M, M );
dim3 gridSize( (N-1)/M+1, (N-1)/M+1 );

saxpy_mat_gpu <<<gridSize, blockSize>>>(d_A, d_B, alpha, N);
```

CUDA设备抽象-存储模型



CUDA存储模型


Device
Multiprocessor 0
Multiprocessor 1
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread
Thread
Thread
Thread
Local Memory
Local Memory
Local Memory
Local Memory
Host
Global Memory
Constant Memory
Texture Memory

CUDA设备抽象-存储资源

- SM级别存储资源
  - 寄存器（Register）
  - 共享内存（Shared Memory）
  - 加载/存储数据单元（LD/ST）
  - Cache（数据，材质，常数）
- GPU芯片级别存储资源
  - Cache（数据/材质）

CUDA存储模型-全局内存

- 全局内存： Global Memory
- 大小：几百MB~几个GB
- 速度：~400时钟周期（非常慢）
- 带宽：~100GB/s

全局内存分配与使用

- 分配与回收：
  - `void cudaMalloc( void ** , size_t )`
  - `void cudaFree( void * )`
- 与主存之间的交互：
  - `void cudaMemcpy( void * , void *, size_t , cudaMemcpyHostToDevice);`
  - `void cudaMemcpy( void * , void *, size_t , cudaMemcpyDeviceToHost);`

CUDA存储模型-共享内存

- 共享内存：Shared Memory （ShMem）
  - 用于同一个线程块（TB）共享使用
  - 速度快，容量小，需手动管理
  - 通过 __shared__ 来定义

```
__global__ void some_kernel( some_type some_data, ... )
{
    ...

    __shared__ float DATA[16][16];

    ....
}
```

共享内存使用模式

- 线程块内各线程协同，初始化ShMem
- 通过同步操作确保初始化完成
- 使用共享内存

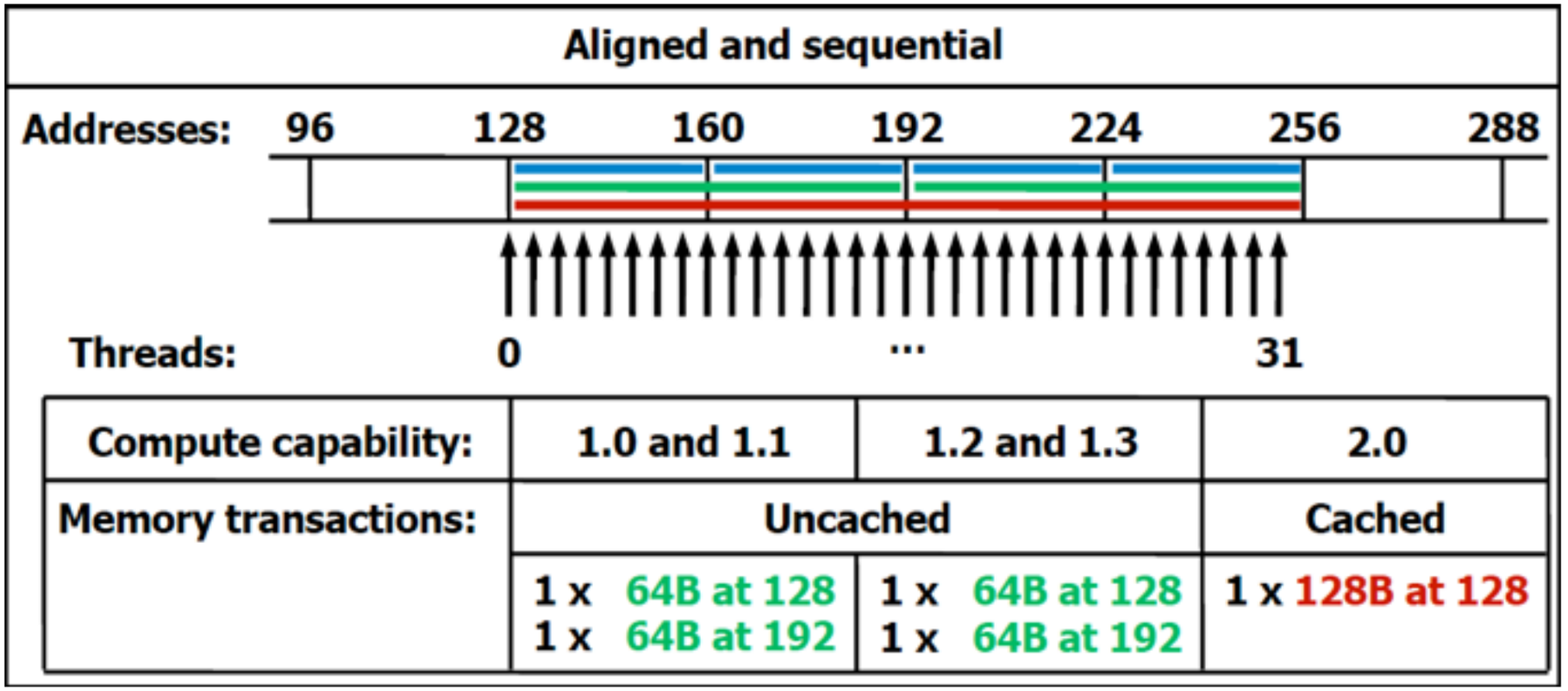
- 相当于手动管理的Cache

存储的组织方式

- 分块方式，提供较高的带宽
- 适用于：
  - 共享内存
  - 寄存器文件
  - 全局内存

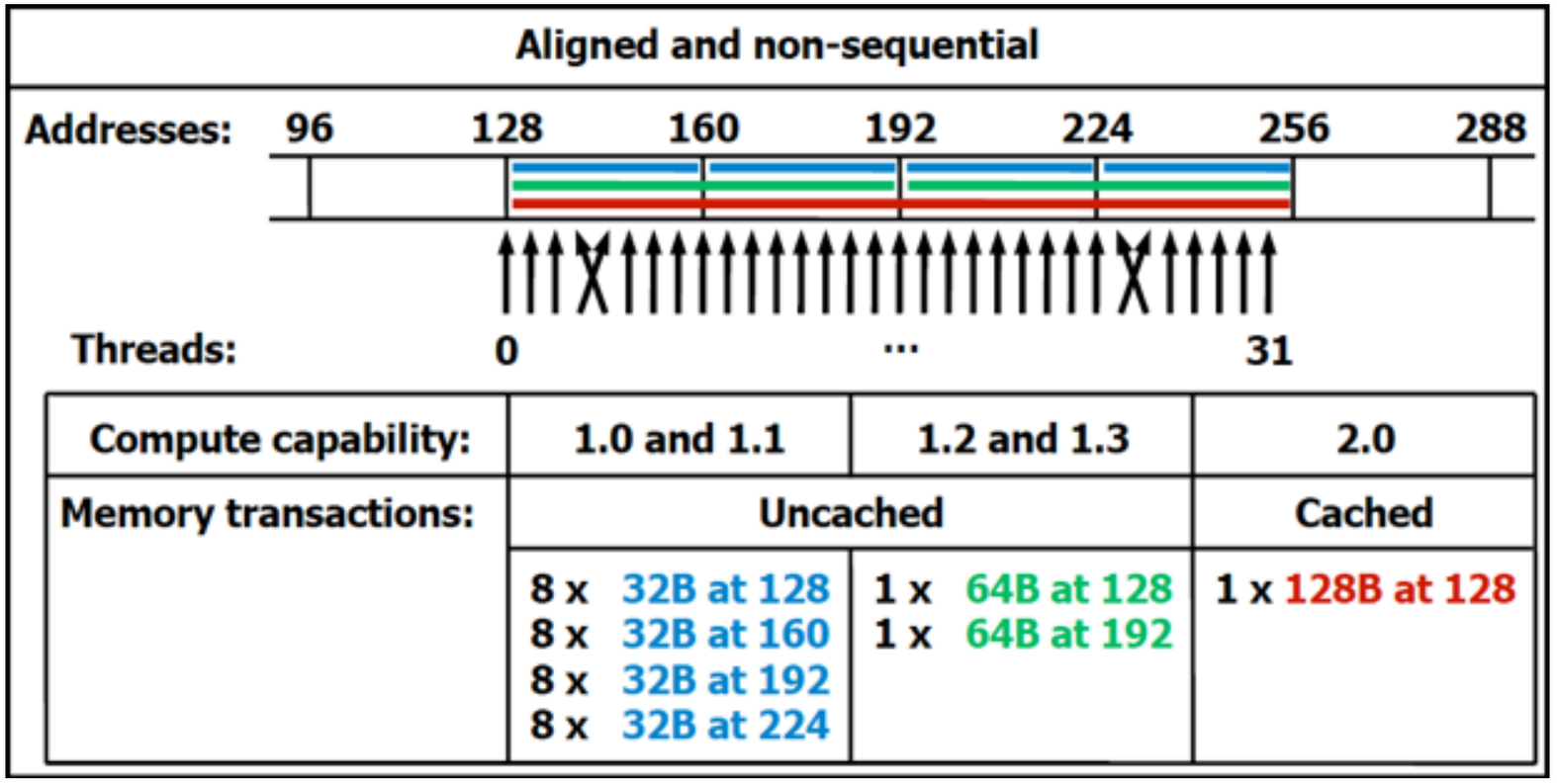
全局内存访问模式

- Coalesced Access 与 Un-Coalesced Access



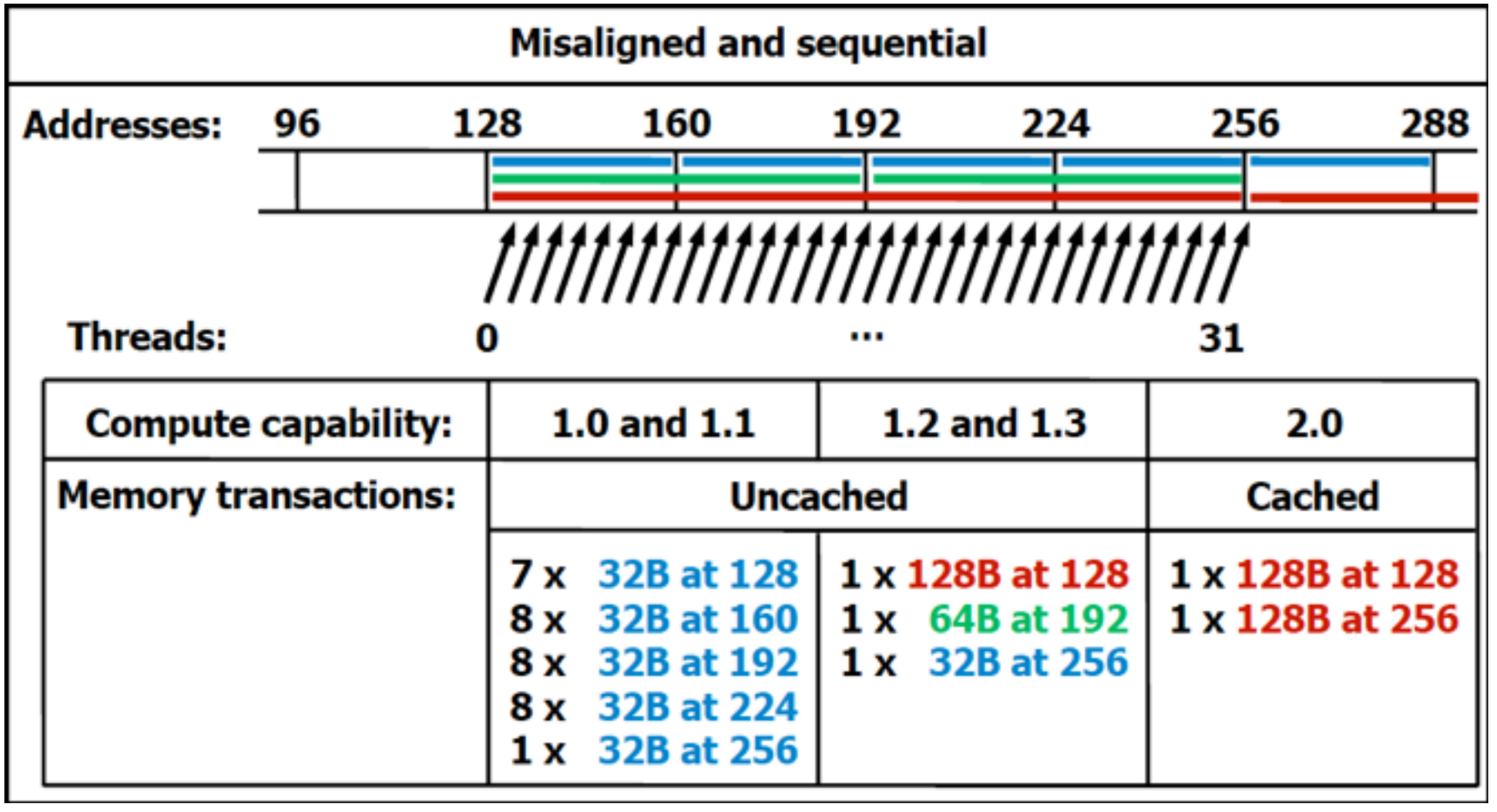
全局内存访问模式

- Coalesced Access 与 Un-Coalesced Access



全局内存访问模式

- Coalesced Access 与 Un-Coalesced Access



Coalesced Access

- 对于无Cache的GPU（GF100之前的）
  - 须严格保持对全局内存的对齐访问
  - 否则全局内存访问效率将大大下降
- 对有Cache的GPU（GF100及之后）
  - 如能保证数据的重用，则不必特别在意Coalesced Access
    - 在一定周期内能用到非对齐部分的其他数据
  - 否则仍需使用Coalesced Access

全局内存的组织

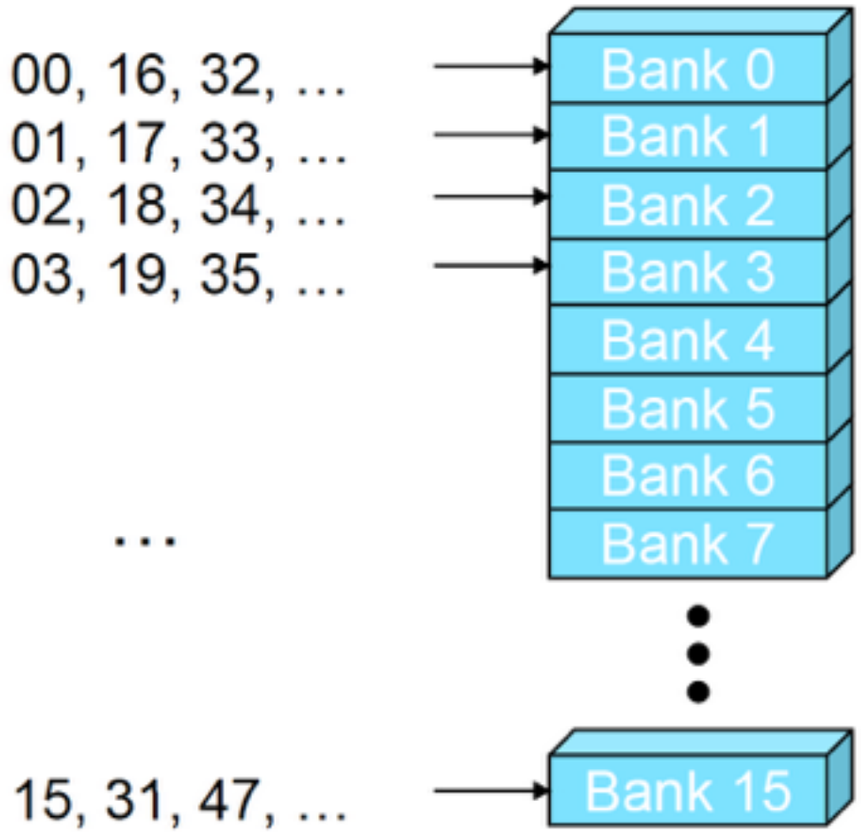
- 全局内存划分为多个Partition
  - 计算能力2.0之前（GT200等）：
    - 6个或8个partition
    - 存在潜在的Partition Camping问题
- 计算能力2.0及之后（GF100等）：
  - 采用哈希算法分配物理地址至Partition，缓解了Partition Camping的问题

共享内存组织

- 共享内存划分为多个Bank
  - GT200：16（共16KB）
  - GF100/GF104：32（共16/48KB）
- 每个Bank宽度：32bit
  - 称为一个字（Word）
- 连续16个（或32个）字存放于不同的Bank

共享内存结构-GT200

- GT200 带宽：
  - 0.65 GHz, 4 byte per Bank, 16 Banks, 30 SM‘s
  - 1.25 TB/s
- GF100 带宽：
  - 0.575 GHz, 4 byte per Bank, 32 Banks, 14 SM’s
  - 1.03 TB/s

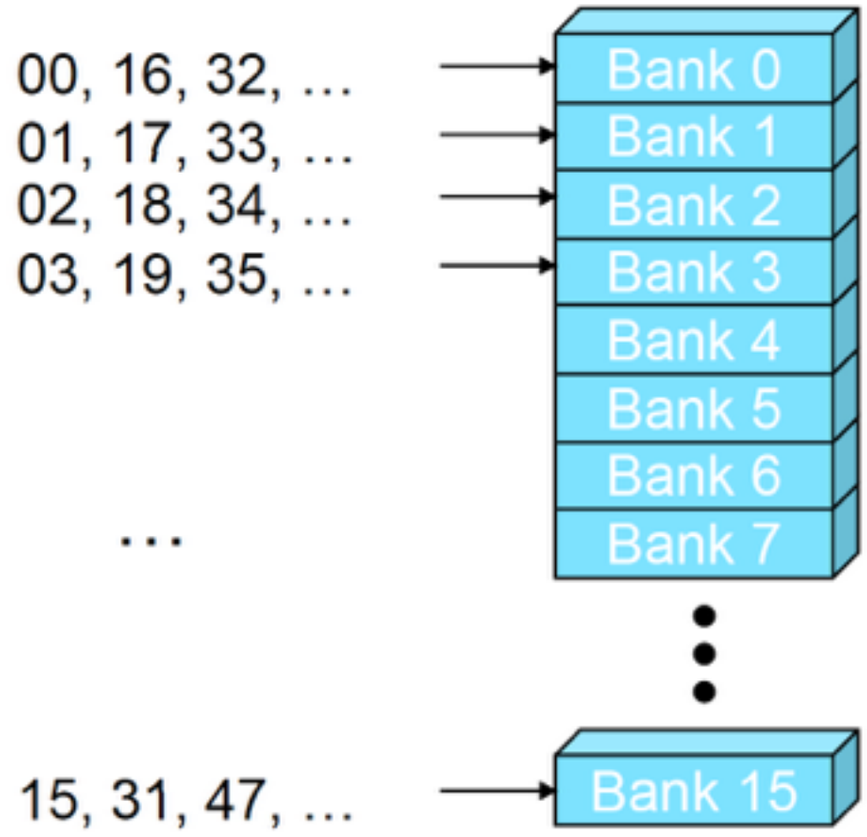


Bank Conflicts

- GT200的共享内存访问分Warp为两部分

- GF100的共享内存可直接进行

- Bank Conflict发生于：
  - 同时进行共享内存访问的进程，访问了同



以下不会产生Bank Conflict

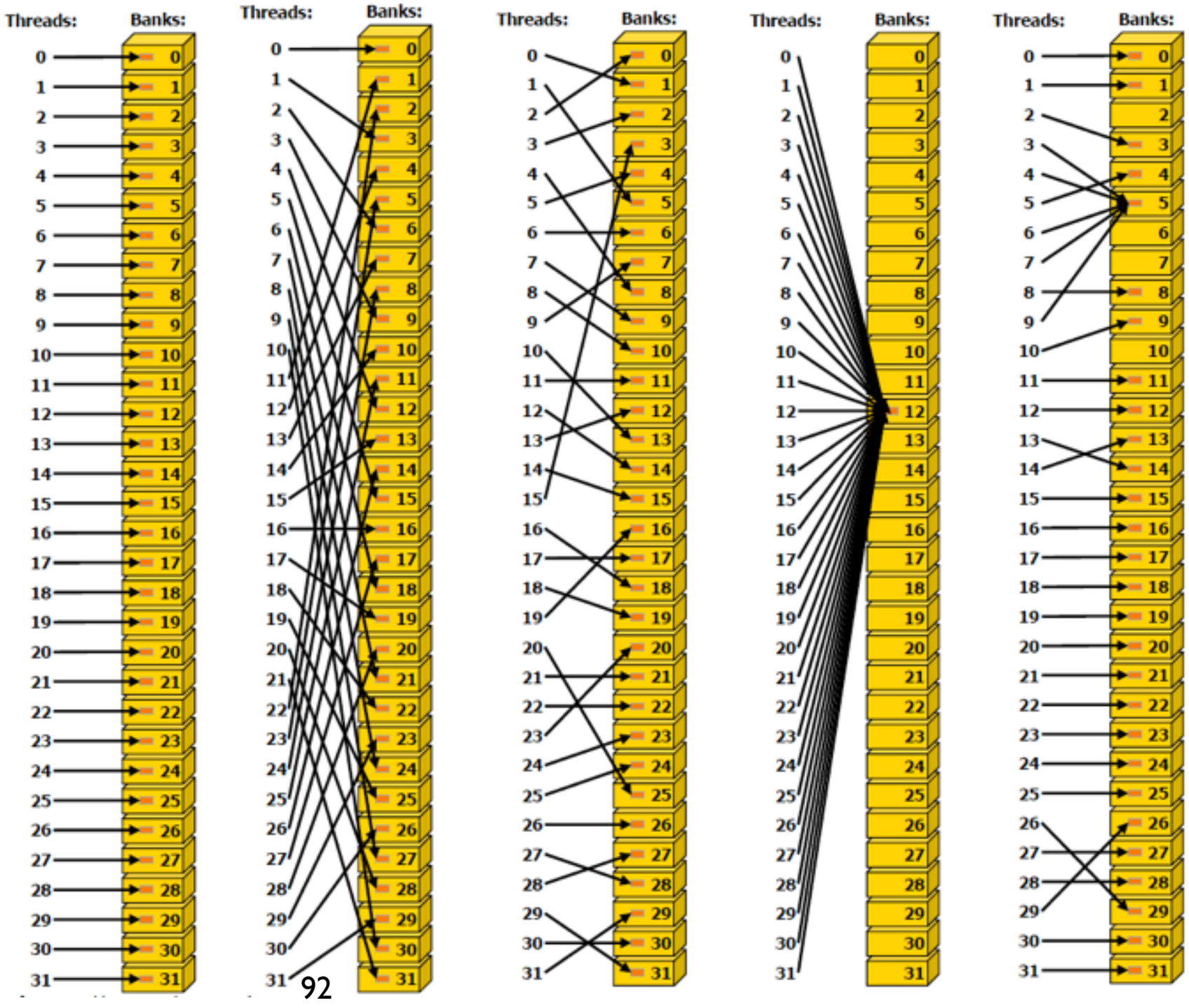
Linear Addressing

Strided Addressing*

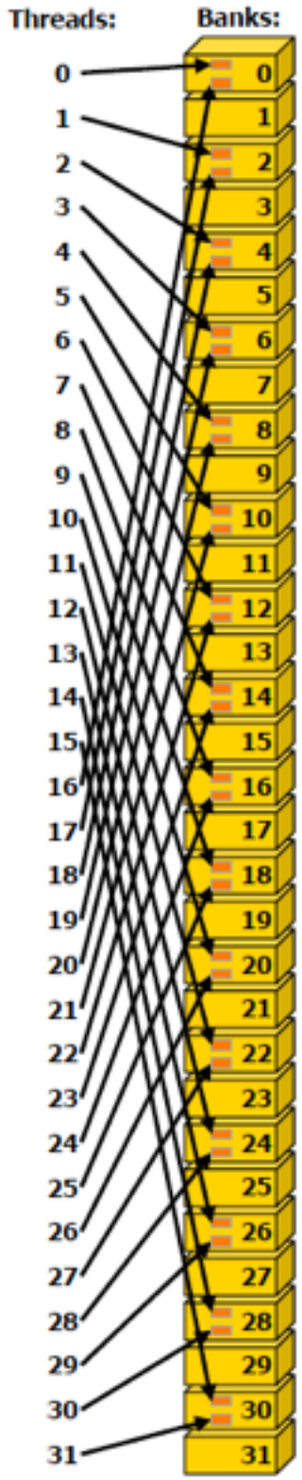
Permutated Addressing

Broadcast Access

Combinations



以下会产生Bank Conflict



Strided Addressing*

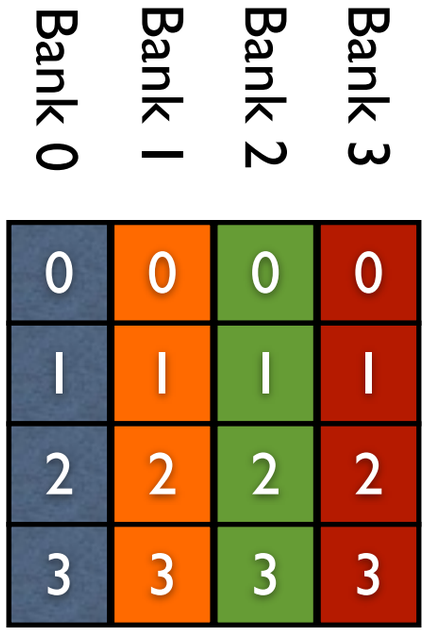
共享内存 - Shared Memory

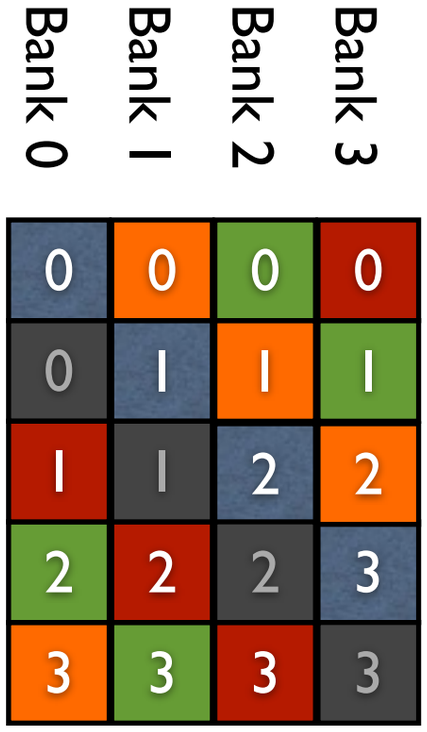
```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

- shared数组存在于共享内存连续的32个地址中
  - GT200：shared[i]和shared[i+16]存在于同一个Bank中
  - GF100：shared数组所有元素均分布在不同Bank中
- 访问过程将涉及 d 次共享内存访问
  - 其中d为s与Bank数目的最大公约数
  - 当s与Bank数目互素时，访问不产生Bank Conflict，否则将发生Bank Conflict

共享内存使用

- 避免Bank Conflict
- 加入必要的padding以避免上述情况





实例 - 矩阵转置

- 转置（Transpose）： A = A[m,n]
  - A^T=A^T[n,m]:A^T[i,j] = A[j,i]
  - 非计算密集型，访存带宽受限
- 实例：
  - 转置单精度、大小为2048x2048的矩阵
  - 基本策略：分块（32x32）处理
  - 面向平台：NVIDIA GeForce GTX280，峰值带宽：
    - 1.1GHz x 32 byte x 8 partition x 2 = 141GB/s

参考应用：矩阵拷贝

```
__global__ void copy(float *odata, float *idata,
                     int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index = xIndex + width*yIndex;

    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS) {
        odata[index+i*width] = idata[index+i*width];
    }
}
```

每个线程块负责32x32块的计算

每个线程块大小为32x8（256个线程）

每个线程处理4个数据

性能评估

- 进行nrep次核心函数调用、求时间平均
- 将核心函数中的计算部分进行nrep次调用并求时间平均
  - 平摊核心调用开销，评价访存特征

```
__global__ void copy(float *odata, float* idata,
                     int width, int height, int nreps)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index = xIndex + width*yIndex;

    for (int r = 0; r < nreps; r++) {
        for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS) {
            odata[index+i*width] = idata[index+i*width];
        }
    }
}
```

矩阵转置-基本版本

```
__global__ void transposeNaive( float *odata, float* idata,
                                int width, int height,
                                int nreps )
{
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

    int index_in  = xIndex +  width * yIndex;
    int index_out = yIndex + height * xIndex;

    for (int r=0; r < nreps; r+=1) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i] = idata[index_in+i*width];
        }
    }
}
```

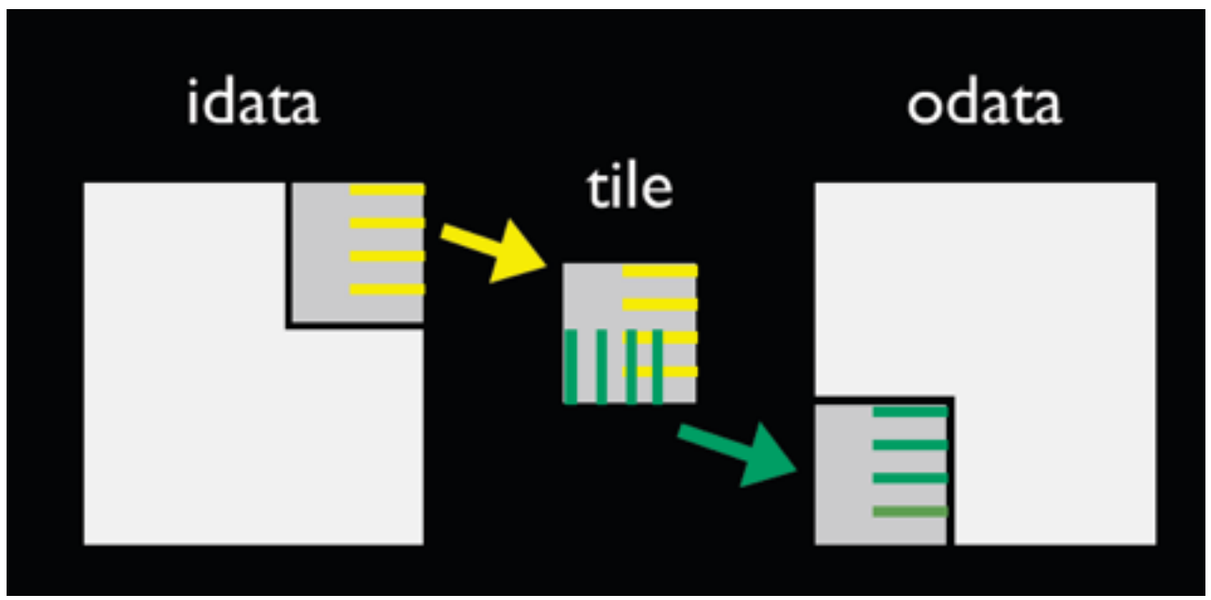
矩阵转置 - 基本版本

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2

问题？全局内存访问特征差！

矩阵转置 - 使用ShMem

- 流程：
  - 读入A的小块，进行转置，写回相应的内存位置
- 为什么使用ShMem?
  - 可以使写出过程变为Coalesced操作



矩阵转置 - ShMem版本

```
__global__ void transposeCoalesced( float *odata,float *idata,
                                    int width, int height, int nreps)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;
    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
        }
        __syncthreads();
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
        }
    }
}
```

矩阵转置 - ShMem版本

- 性能评价

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

矩阵转置 - ShMem+版本

- ShMem版本的一个问题
  - ShMem读取访问存在Bank Conflict

```
for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
        }
        __syncthreads();
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
        }
    }
}
```

- ShMem+版本：通过padding消除Bank Conflict

矩阵转置 - ShMem+版本

- 性能评估

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2

矩阵转置 - ShMem+版本

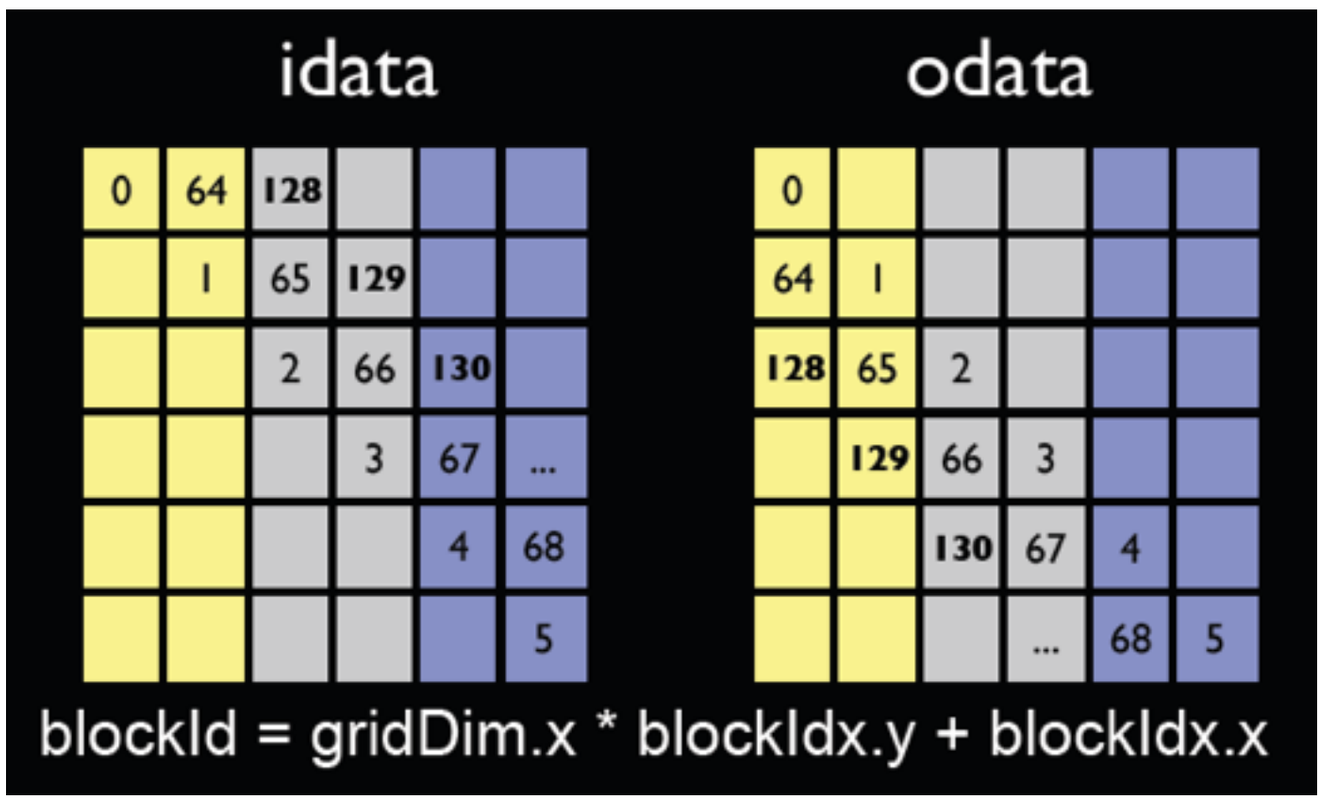
- 仍未解决的性能问题：
  - 与理论带宽值相去甚远
  - 问题？GTX280 的 Partition Camping
    - 类似Bank Conflict
    - 8个Partition，每256Byte存在于一个Partition上
    - 相距2KB的数据则同样位于同一Partition

矩阵转置 - Partition Camping

- 2048x2048的单精度矩阵
  - 每行大小整除 2KB
  - 转置后同一行各tile的写回过程引发 Partition Camping 问题

解决办法?

- 设计另一种解读线程块身份的方式



矩阵转置 - 最终版本

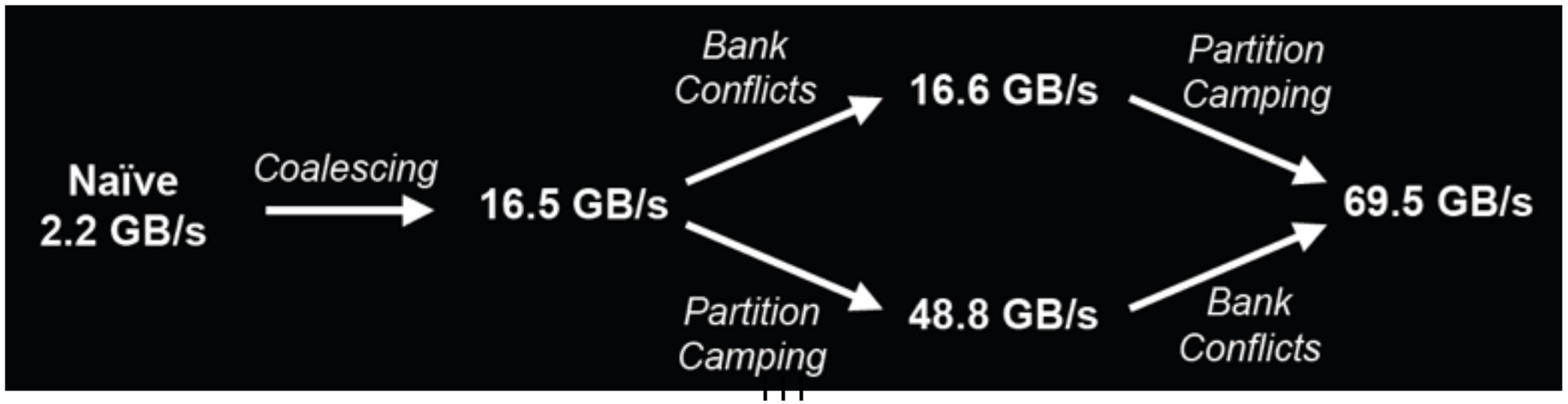
```
__global__ void transposeDiagonal(float *odata, float *idata,
                                  int width, int height, int nreps)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];

    int blockIdx_y = blockIdx.x;
    int blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;

    int xIndex = blockIdx_x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx_y * TILE_DIM + threadIdx.y;

    int index_in = xIndex + (yIndex)*width;
    xIndex = blockIdx_y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx_x * TILE_DIM + threadIdx.y;

    int index_out = xIndex + (yIndex)*height;
    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
        }
        __syncthreads();
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
        }
    }
}
```

矩阵转置 - 最终版本

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Diagonal	69.5	78.3

矩阵转置 - 优化过程回顾

- 观察反常的性能特性，进行分析
- 针对影响性能的特性做相应的处理
- 改变访存方式
- 调整算法设计



实例 - 矩阵转置

- Coalesced Access
- 有效利用共享内存
  - 避免Bank Conflicts
- 根据性能需求，改善算法设计

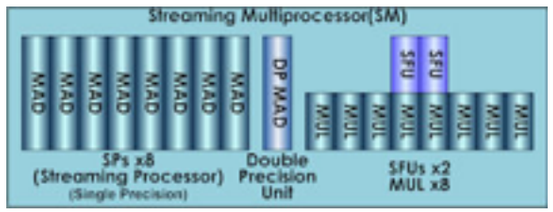
CUDA设备计算资源

- GPU由几个流处理器集合（Stream Multiprocessor，即SM）构成
- 每个SM由多个流处理器（Stream Processor，即SP）、存储部件、以及其他一些功能部件（SFU）构成

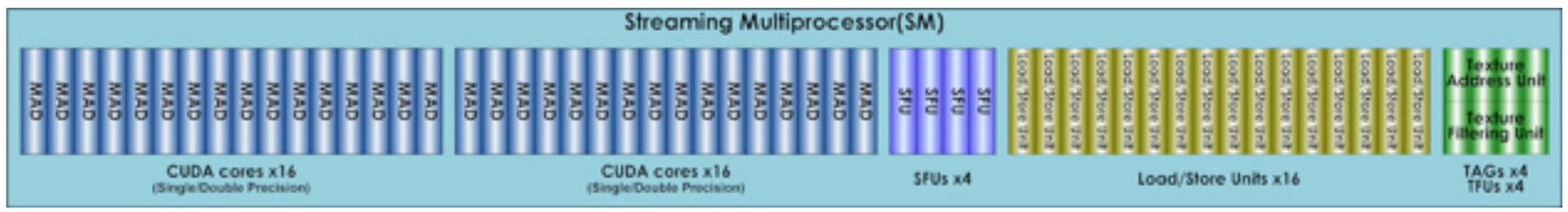
	年份	SM个数	SP个数	SFU个数
GT200	2008	16	8	2
GF100	2010	14/16	16x2	4
GF104	2010	7	16x3	4
GK104	2012	8	16x12	16x2
GK110	2013	14	16x16	

CUDA设备抽象-计算资源

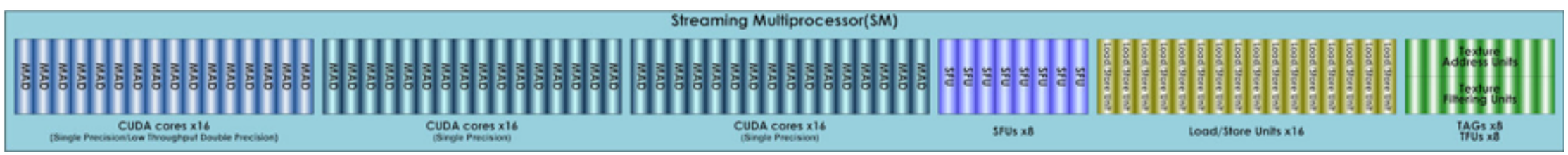
GT200



GF100



GF104



CUDA设备抽象-SM资源

- 每个资源都是一条流水线

	SP		SFU		Tex		LD/ST		总数
	宽度	个数	宽度	个数	宽度	个数	宽度	个数	
GT200	8	1	4	1	4	1/3	8	1/3	2 2/3
GF100	16	2	4	1	4	1	16	1	5
GF104	16	3	8	1	8	1	16	1	6
GK104	16	12	16	2	8	2	16	2	18
GK110	16	16	16	2	8	2	16	2	22

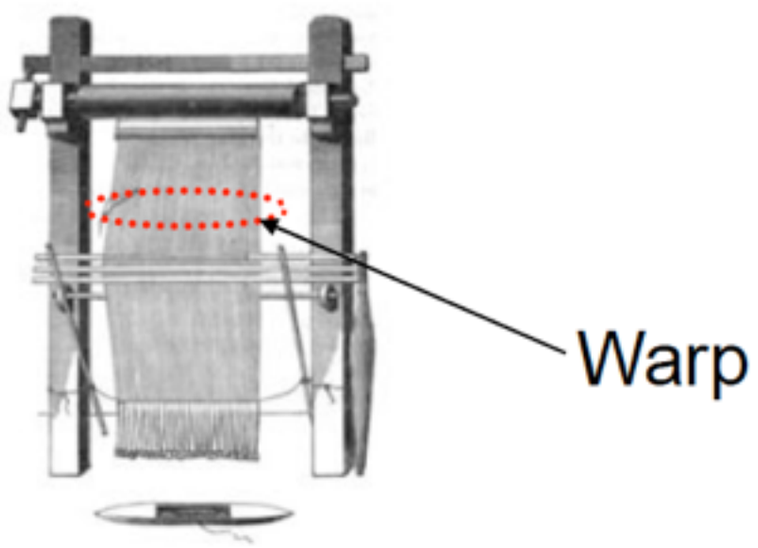
线程调度

- 编译期：
  - 根据硬件，决定每个SM上可运行的线程块个数
- 运行期：
  - 以线程块为粒度调度到SM上
    - 用户透明，简单的Round-Robin方式
  - 以Warp（32个线程）为粒度调度到流水线上
    - 可同时调度多个Warp到多个流水线

Warp：最基本的调度单元

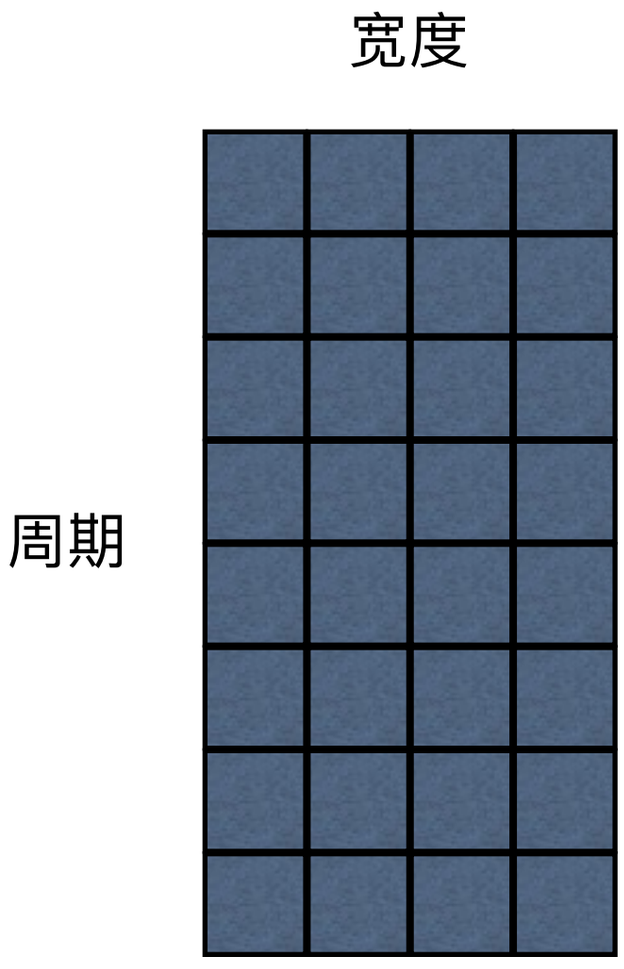
- Warp的稳定性：始终是32
  - 以SIMD方式运行

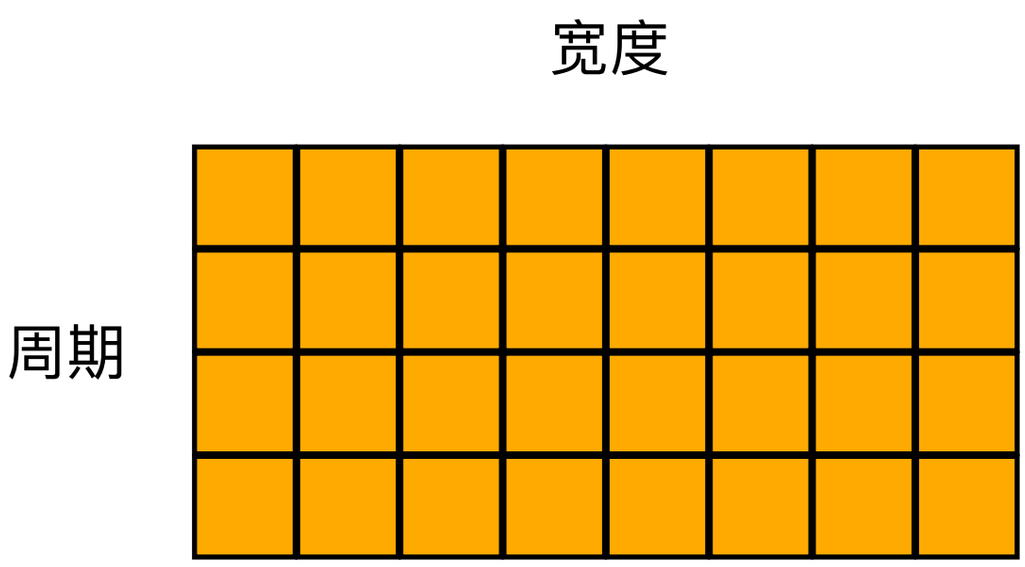
- Warp级别问题：
  - Warp在流水线上的执行
  - Warp调度



Warp在流水线上的执行

- 宽度n：每个周期都能发射n条指令
- 延时m：某个数据开始执行到执行结束所需周期

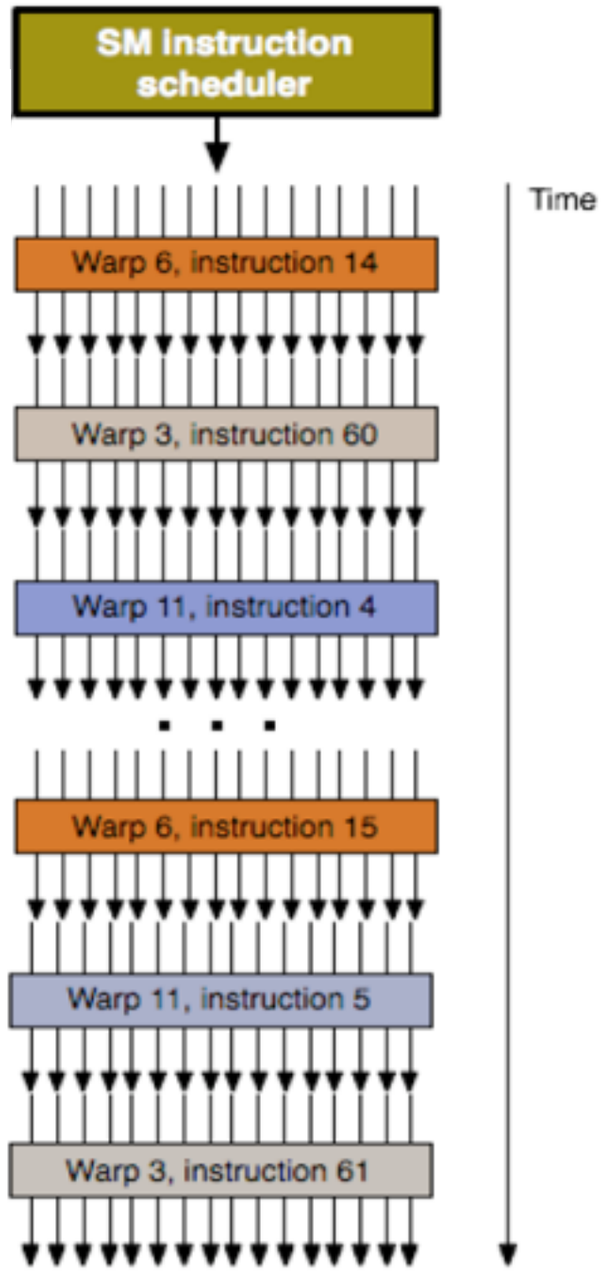




大小为w的Warp在宽度为n、延时为m的流水线上执行，将在(w/n+m-1)个周期后得到结果

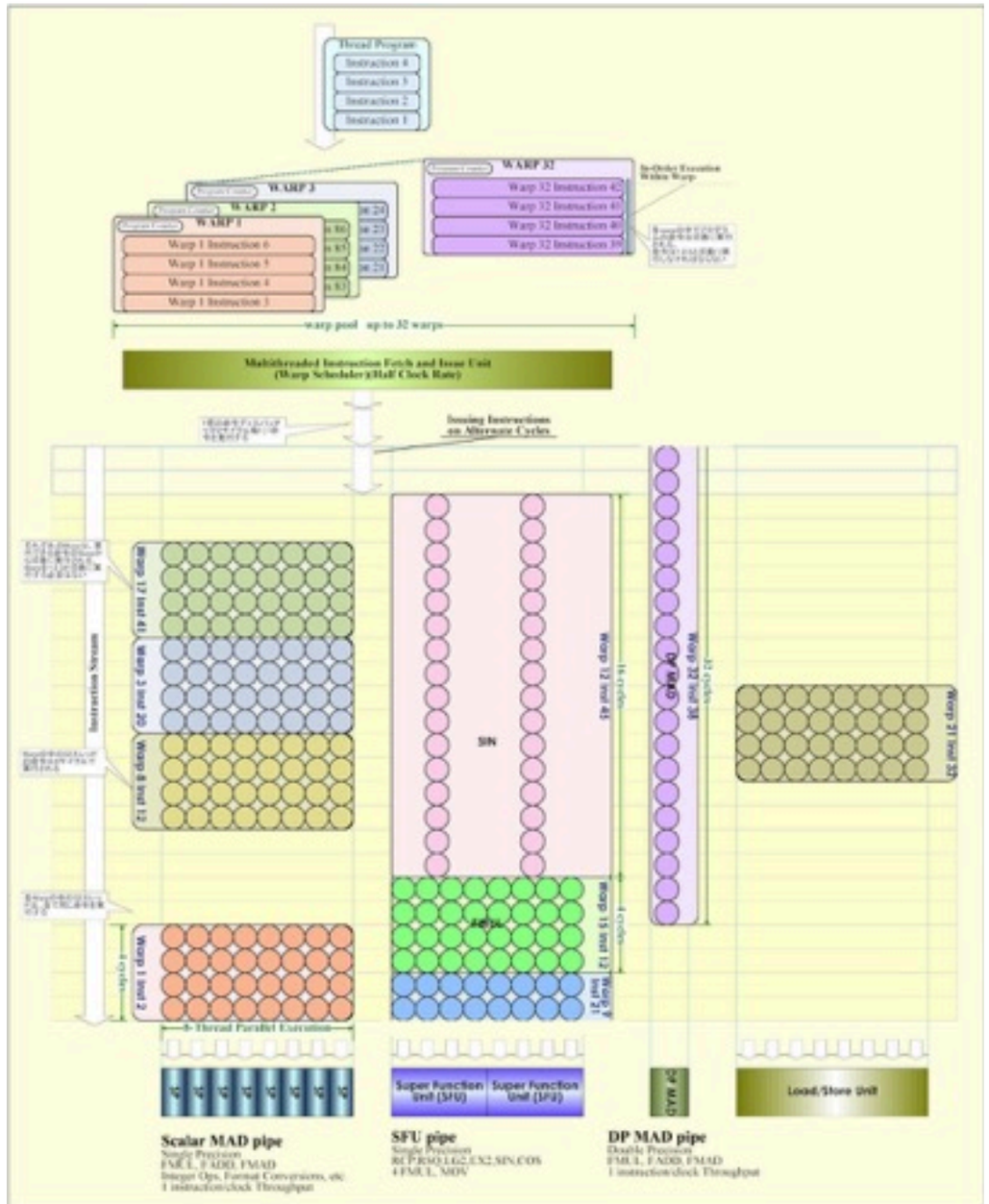
Warp的调度

- 当某个Warp所有输入均有效时，Warp可以被执行
- 当Warp进入高延时流水线中，硬件将调用其他Warp以保证整体吞吐率
  - 高延时操作包括数据运算和访存
  - 其他Warp可能来自其他线程块



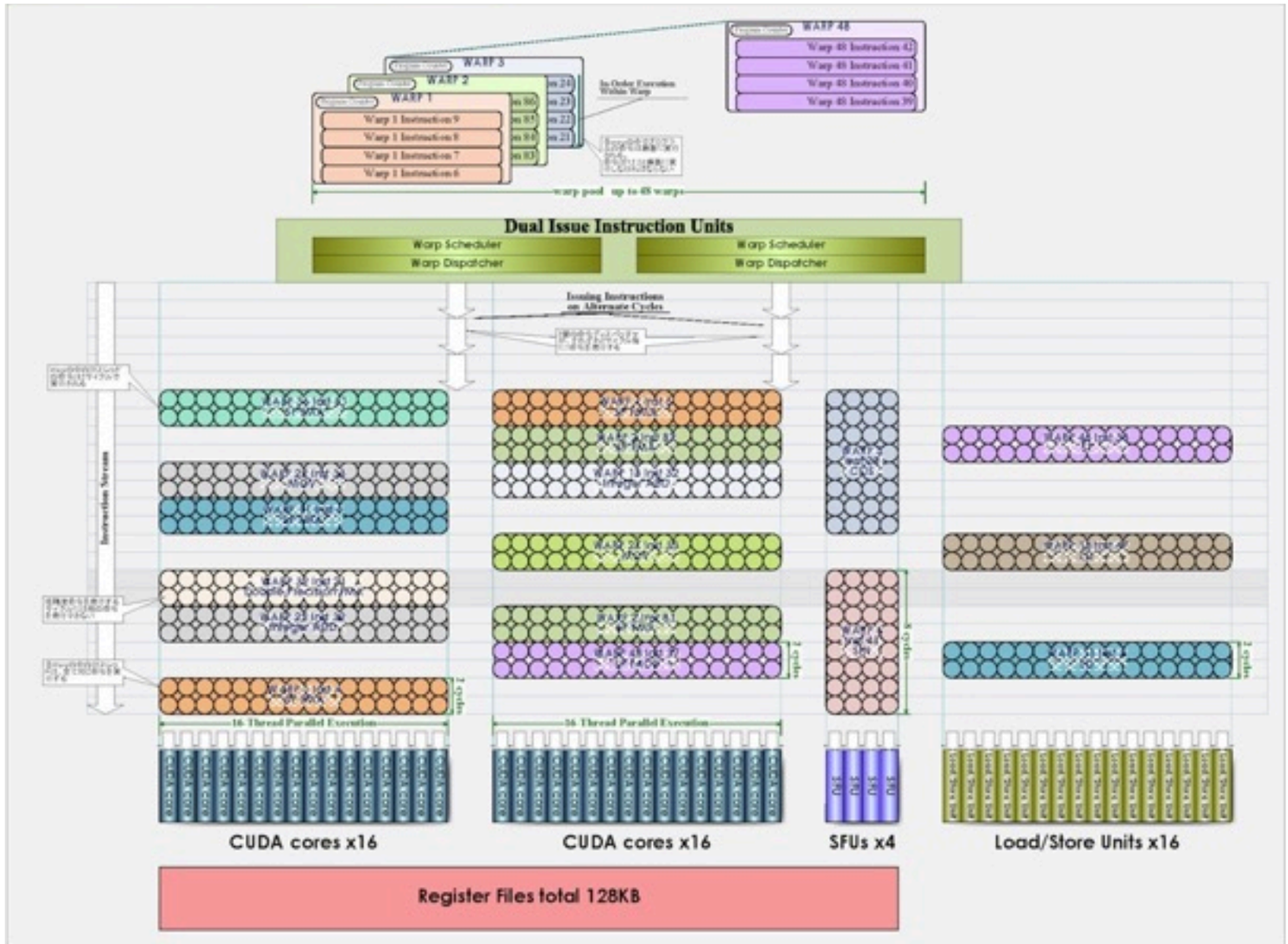
GT200上的Warp调度

- 最多可有32个Warp
  - 可来自于不同的线程块
- 每周期可发射来自于一个Warp的指令



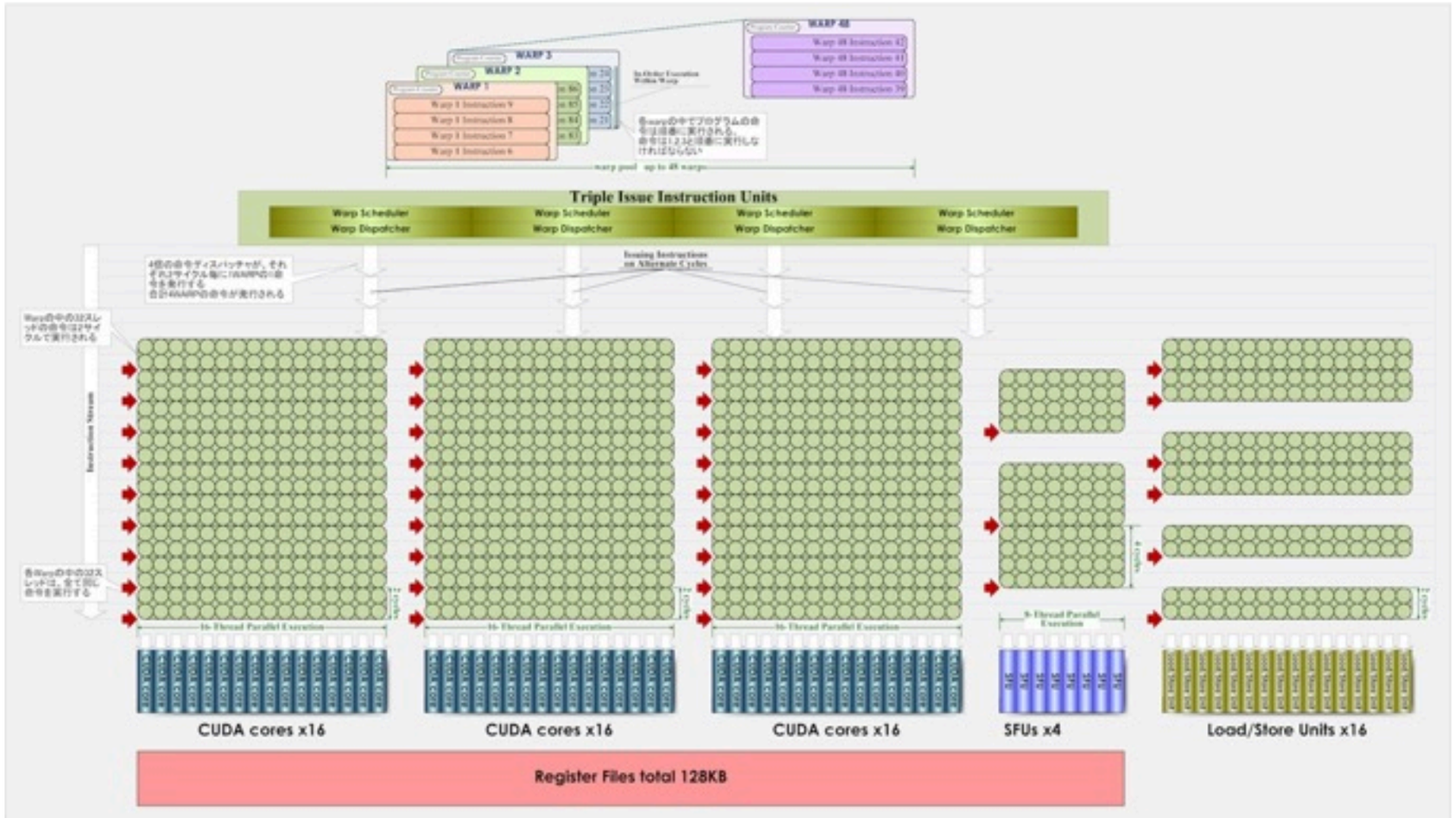
GF100上的Warp调度

- 每周期发射两组来自不同Warp的指令



GF104上的Warp调度

- 每周期4组指令
- 可能来自于同一Warp的两条指令



分支与Warp

- 分支与SIMD直觉相抵触
- 当不同线程执行if-else语句，怎么办?
- 不同Warp之间互不影响
- 同一Warp内的各线程分支判断结果不同时，分支两侧均被执行，部分结果被相应屏蔽

线程块调度与生命周期

- 线程块调度至某个SM
  - 将始终在该SM上执行
- 启动各个Warp
- 直至最后一个Warp退出，线程块方退出

CUDA 优化

- 基本优化原则
  - 提供足够的并行度以增加吞吐率
  - 避免潜在的性能陷阱
- 理解性能行为
  - 了解硬件特征
  - 了解应用本身的性能特性与瓶颈

峰值速度的计算

- 计算特性
  - MAD操作数 x SP个数 x SM个数 x 频率
- 访存特性
  - 宽度 x 频率

计算能力举例

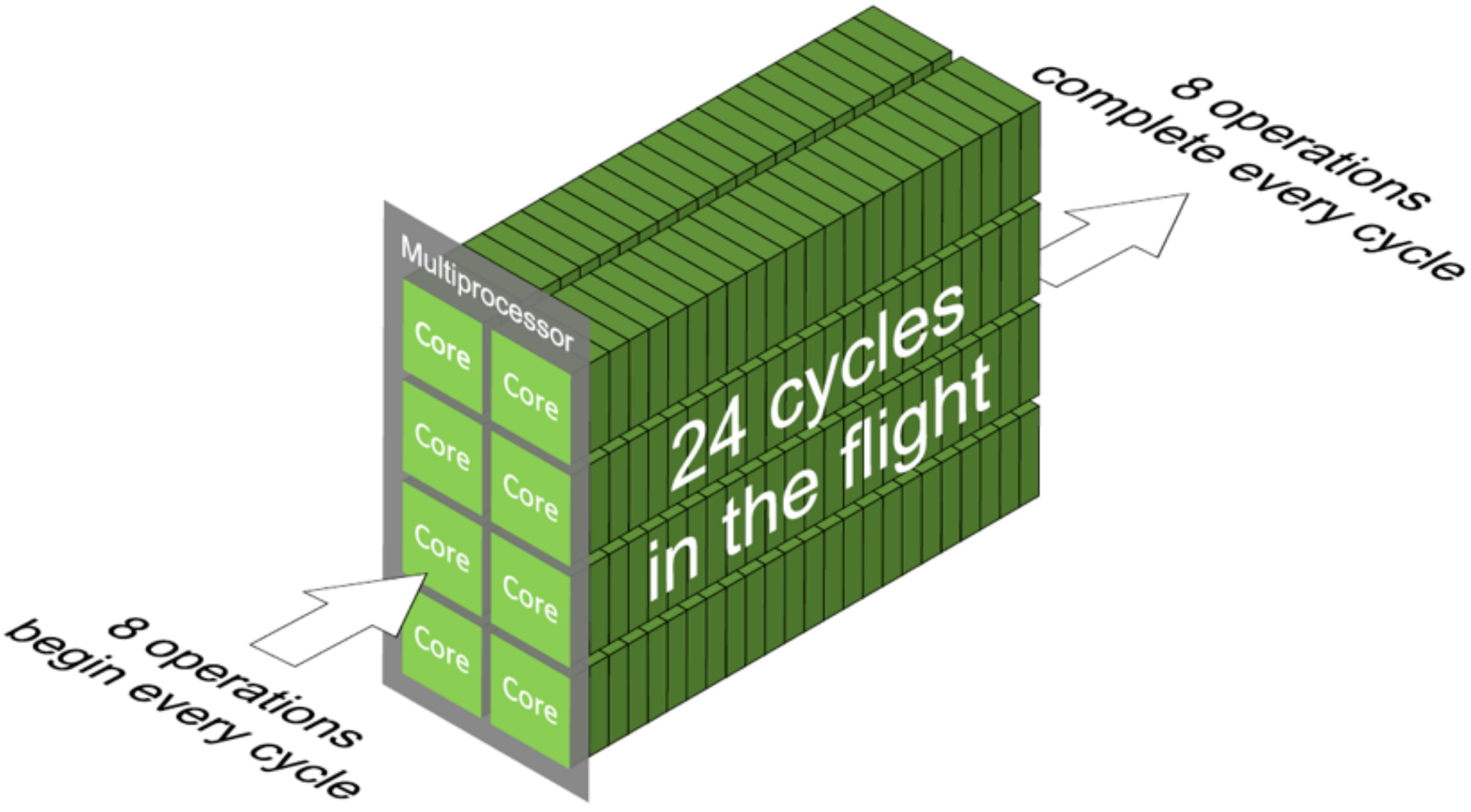
	SM 配置			运算频率	峰值速度	
	单精度	双精度	个数		单精度	双精度
Tesla C1060	8	1	30	1.3 GHz	624 GFLOPS	78 GFLOPS
Tesla C2070	32	16	14	1.15 GHz	1.03 TFLOPS	515 GFLOPS

	Partition 宽度	Partition 个数	内存类型	内存频率	峰值带宽
Tesla C1060	64 bit	8	GDDR3	800 MHz	102 GB/s
Tesla C2070	64 bit	6	GDDR5	1.5 GHz	144 GB/s

足够的并行度

- 如何充分利用流水线?
  - 访存系统、计算流水线
- 原则： Little's Law
  - 宽度为W，延时为T的流水线
  - 需提供W*T个并行的操作方能保证流水线被充分利用

GT200的计算单元


8 operations
complete every cycle
Multiprocessor
Core
Core
Core
Core
Core
Core
Core
Core
24 cycles
in the flight
8 operations
begin every cycle

举例-计算单元

GPU model	Latency (cycles)	Throughput (cores/SM)	Parallelism (operations/SM)
G80-GT200	≈24	8	≈192
GF100	≈18	32	≈576
GF104	≈18	48	≈864

举例-访存

	Latency	Throughput	Parallelism
Arithmetic	≈18 cycles	32 ops/SM/cycle	576 ops/SM
Memory	< 800 cycles (?)	< 177 GB/s	< 100 KB

基本原则

- 计算受限：
  - 提供足够的计算并行度以保证流水线的高吞吐率
- 访存受限：
  - 提供足够的并发访存请求
    - 尽可能地使用缓存
      - Cache 与 共享内存

如何有效利用寄存器

- 寄存器是速度最快的存储部件
  - 带宽高，速度快
- 主要问题：
  - 只能通过声明局部变量的方式以使用
  - 线程间不能通过寄存器通信

寄存器文件的性能

- 考虑GF100上的MAD操作（单精度）
  - GF100： 14个SM、32个单精度MAD单元
  - 每个单精度MAD操作需要：
    - 12 byte输入，4 byte输出
- 因此，达到峰值计算速度：
  - 14 x 32mad/cycle x 2 flop/mad x1.15 GHz = 1.03 TFLOPS
- 需提供的存储带宽为：
  - 14 x 32mad/cycle x (12+4)byte/mad x 1.15GHz = 8.24 TB/s

寄存器文件的组织

- 寄存器划分为多个Bank
  - 多读端口，Bank数远多于共享内存
    - GT200：64
      - 通过测试得到
    - GF100：>64

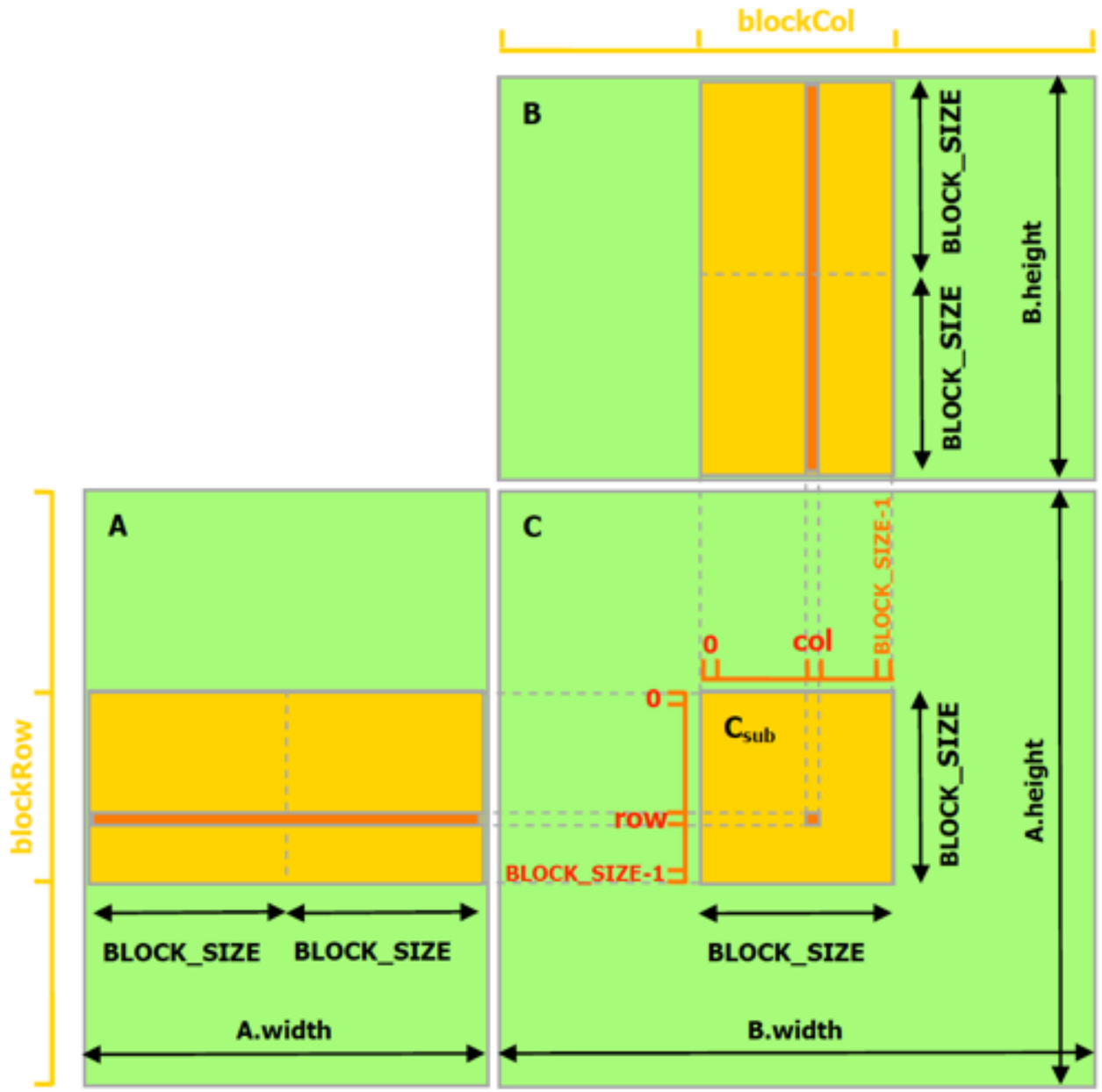
寄存器 v.s. 共享内存

- 寄存器可提供比共享内存高得多的读写带宽
  - >8 TB/s v.s. ~1.25 TB/s
- 有效使用寄存器是能达到峰值计算性能的关键
  - 共享内存并不能提供足够的带宽

如何有效利用寄存器?

- 首先，确定必要性
  - 适用于计算密集型问题（而非访存密集型问题）
- 利用方法
  - 每个线程通过局部变量读写更多的数据，进行更多的运算

举例 - 稠密矩阵相乘


blockCol
B
BLOCK_SIZE
BLOCK_SIZE
B.height
A
C
BLOCK_SIZE-1
0
col
0
C_{sub}
row
BLOCK_SIZE-1
BLOCK_SIZE
A.height
blockRow
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
A.width
B.width

使用共享内存

- 重用A和B矩阵的每个子矩阵
  - MxM的子块
- 降低全局内存访问量
- 改善对B矩阵的访问模式

CUDA程序举例 (I)

```
#define M 16

// Matrices are stored in row-major order with padding:
// A(row, col) = *(A.elements + row * A.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value)
{
    A.elements[row * A.stride + col] = value;
}

// Get the MxM sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;

    Asub.width = M;
    Asub.height = M;
    Asub.stride = A.stride;
    Asub.elements = &A.elements[A.stride * M * row + M * col];
    return Asub;
}
```

CUDA程序举例 (2)

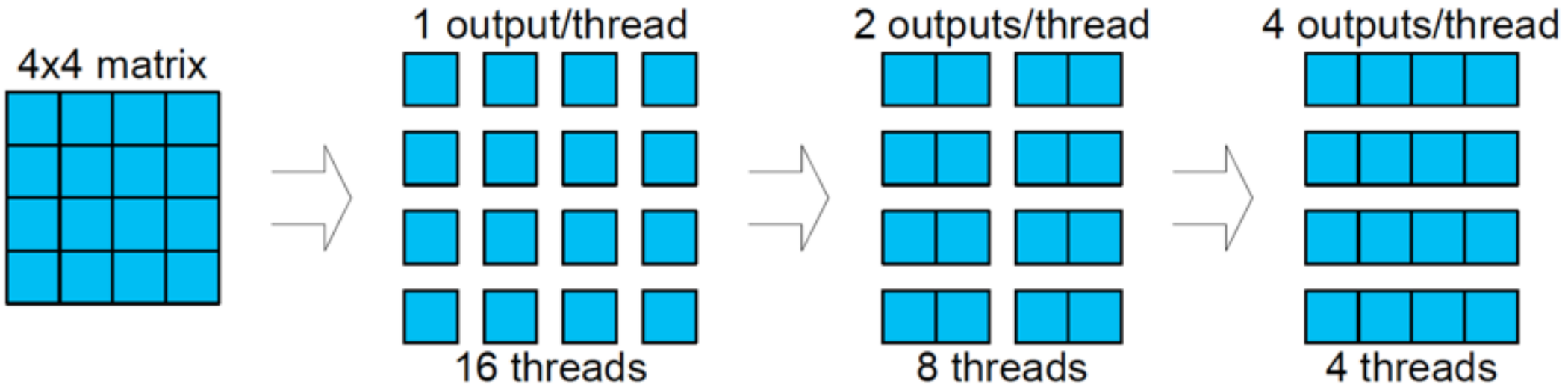
```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are required to compute Csub
    for (int i = 0; i < (A.width / M); i += 1) {
        // Get sub-matrices of A and B
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[M][M];
        __shared__ float Bs[M][M];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded before computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int j = 0; j < M; j += 1)
            Cvalue += As[row][j] * Bs[j][col];

        // Synchronize to make sure that the preceding computation is done
        __syncthreads();
    }

    // Write Csub to device memory, with each thread writing one element
    SetElement(Csub, row, col, Cvalue);
}
```

CUDA程序举例 (3)

```
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(M, M);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

举例 - 矩阵乘法

- 每线程处理目标矩阵中的单个元素 ->
- 每线程计算目标矩阵中的多个元素



矩阵乘法 - 基线

- 平台：
  - NVIDIA GeForce GTX480
- 基于分块版本的矩阵乘法
  - 矩阵大小： 1024 x 1024
  - 增加Block Size为32
  - 进行循环展开
- 性能：242 GLOPS

矩阵乘法 - 基线

```
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

21 register per thread, 242 GLOPS

矩阵乘法 - 每线程2输出

- 调整线程组织方式

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE/2); //32x16
dim3 grid(uiWC / BLOCK_SIZE, uiHC / BLOCK_SIZE);
```

- 核心计算(I)：

```
float Csub[2] = {0,0};//array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
                    a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
```

矩阵乘法 - 每线程2输出

- 核心计算(2):

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+16, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```

28 register per thread, 341 GLOPS (x1.4)

矩阵乘法 - 每线程2输出

- 对共享内存访问的变化：
  - BS 矩阵被更大程度地重用

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+16, k) * BS(k, tx);
}
```

矩阵乘法 - 每线程4输出

- 进一步调整线程组织方式

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE/4); //32x8
dim3 grid(uiWC / BLOCK_SIZE, uiHC / BLOCK_SIZE);
```

- 核心计算(I)：

```
float Csub[4] = {0,0,0,0};//array is in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
                    a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
```

矩阵乘法 - 每线程4输出

- 核心计算(2):

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+8, k) * BS(k, tx);
        Csub[2] += AS(ty+16, k) * BS(k, tx);
        Csub[3] += AS(ty+24, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```

41 register per thread, 427 GLOPS (x1.76)

矩阵乘法 - 每线程8输出

- 每线程使用63个寄存器
- 每线程:
  - 读取8个A或B矩阵的值
  - 计算并写回8个C矩阵的值
- 更大程度地重用BS矩阵

MAGMA 中的 DGEMM

- MAGMA：
  - LAPACK的高性能GPU版本
  - 每线程处理36个元素
  - 性能： 838 GLOPS
- 原因？更好地利用了寄存器！

如何通过CUDA加速科学应用?

- 整体移植
  - 用CUDA的原生语言重写
  - 更改FORTRAN/C代码
  - 利用第三方库提升可编程性
- 定位hotspot并针对性地用GPU进行实现
- 定位hotspot并利用已有高性能软件库

OpenACC 举例

Fortran Version

```
program picalc
   implicit none
   integer, parameter :: n=1000000
   integer :: i
   real(kind=8) :: t, pi
   pi = 0.0
!$acc parallel loop
   do i=0, n-1
      t = (i+0.5)/n
      pi = pi + 4.0/(1.0 + t*t)
   end do
!$acc end parallel loop
   print *, 'pi=', pi/n
end program picalc
```

C Version

```
#include <stdio.h>
#define N 1000000

int main(void) {
double pi = 0.0f; long i;
#pragma acc parallel loop
for (i=0; i<N; i++) {
   double t= (double)((i+0.5)/N);
   pi +=4.0/(1.0+t*t);
}
printf("pi=%16.15f\n",pi/N);
return 0;
}
```

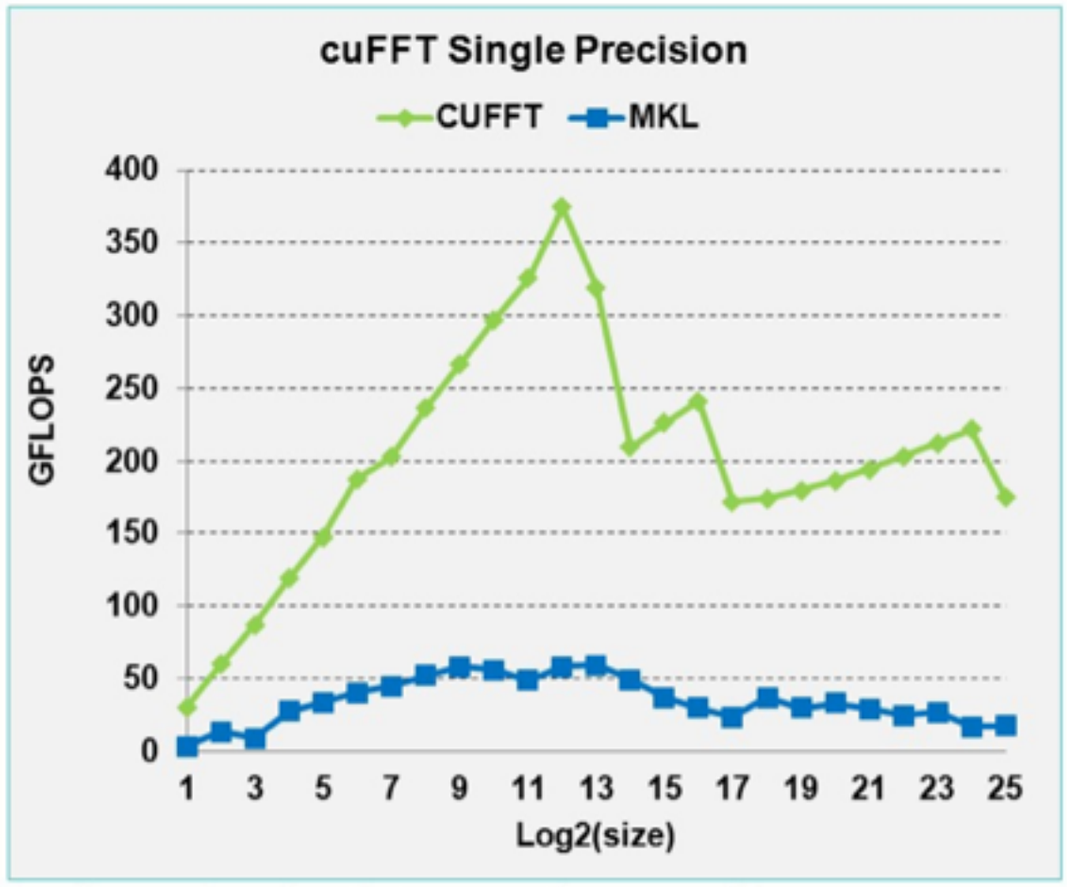
CUDA软件库-官方

软件包	范畴	功能
cuBLAS	线性代数（稠密）	BLAS 1/2/3 操作
cuFFT	快速傅里叶变换	快速傅里叶变换
cuSPARSE	线性代数（稀疏）	前代回代，矩阵乘等
CUSP	线性代数（稀疏）	迭代法+预处理器
Performance Primitives	信号与图像处理	信号与图像处理

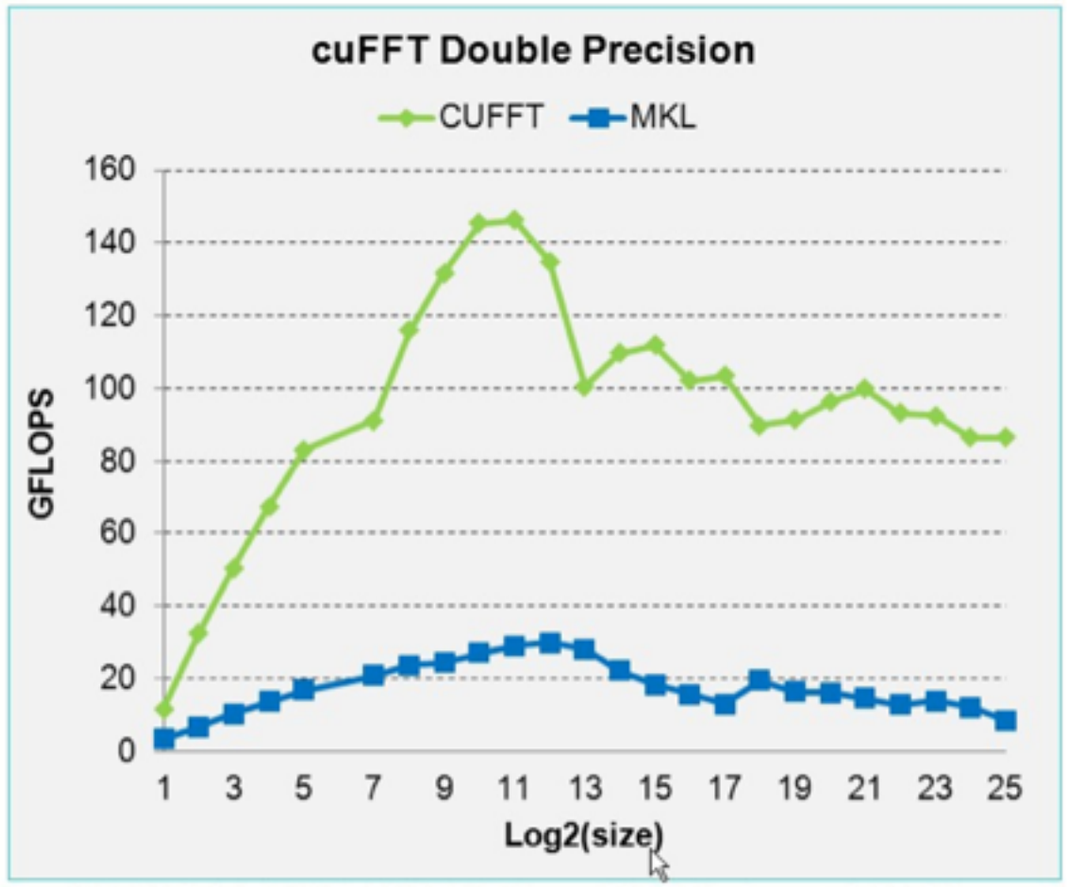
CUDA软件库-第三方

软件包	范畴	功能
Thrust	基本算法	排序，规约，查找，变换
MAGMA	线性代数（稠密）	BLAS+LAPACK
CULA Tools	线性代数	稠密与稀疏线性代数
IMSL Fortran Numerical Library	语言绑定	直接加速FORTRAN程序

CUDA软件库-cuFFT

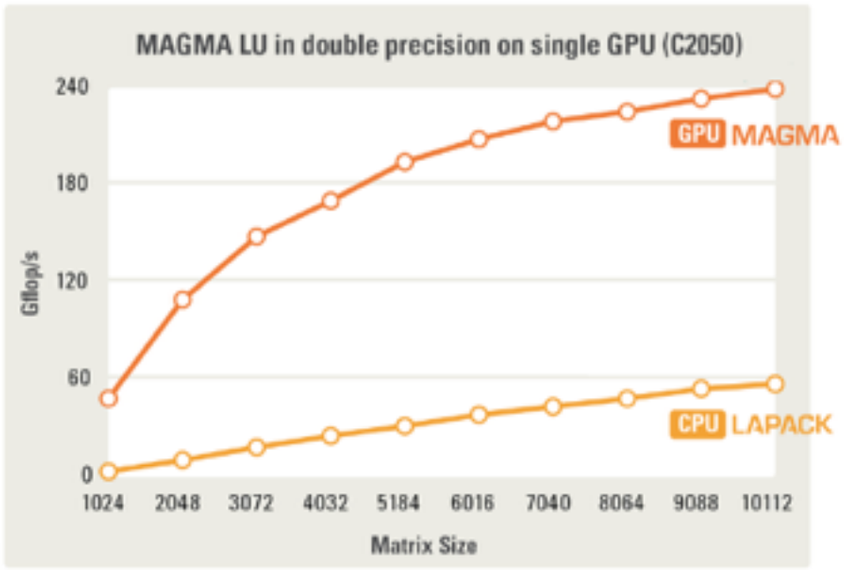


- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz



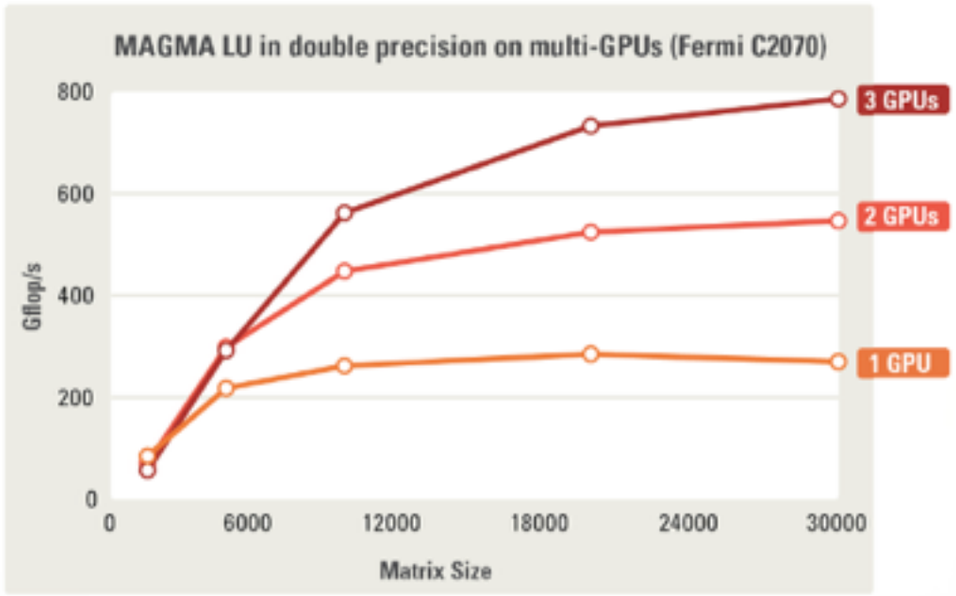
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
- Performance may vary based on OS version and motherboard configuration

CUDA软件库-MAGMA



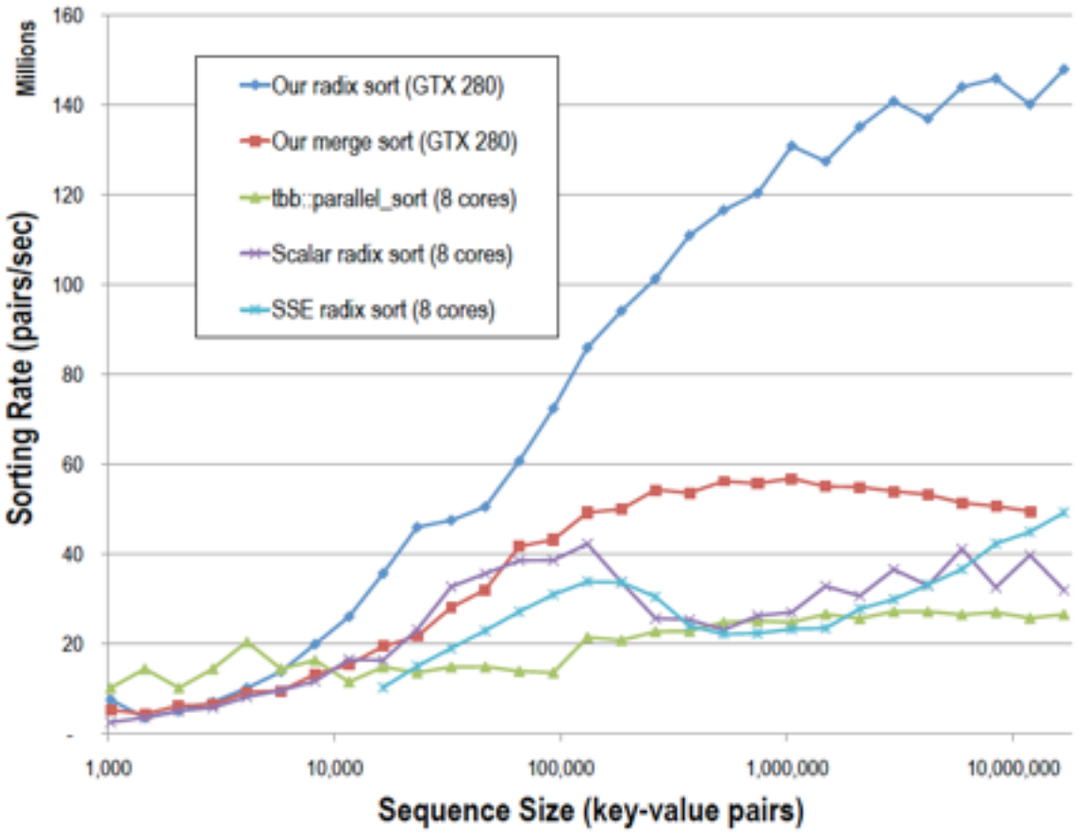
GPU Fermi C2050 (448 CUDA Cores @ 1.15 GHz)
+ Intel Q9300 (4 cores @ 2.50 GHz)
DP peak **515** + **40** GFlop/s
Power * ~220 W

CPU AMD Istanbul
[8 sockets x 6 cores (48 cores) @2.8GHz]
DP peak **538** GFlop/s
Power * ~1,022 W

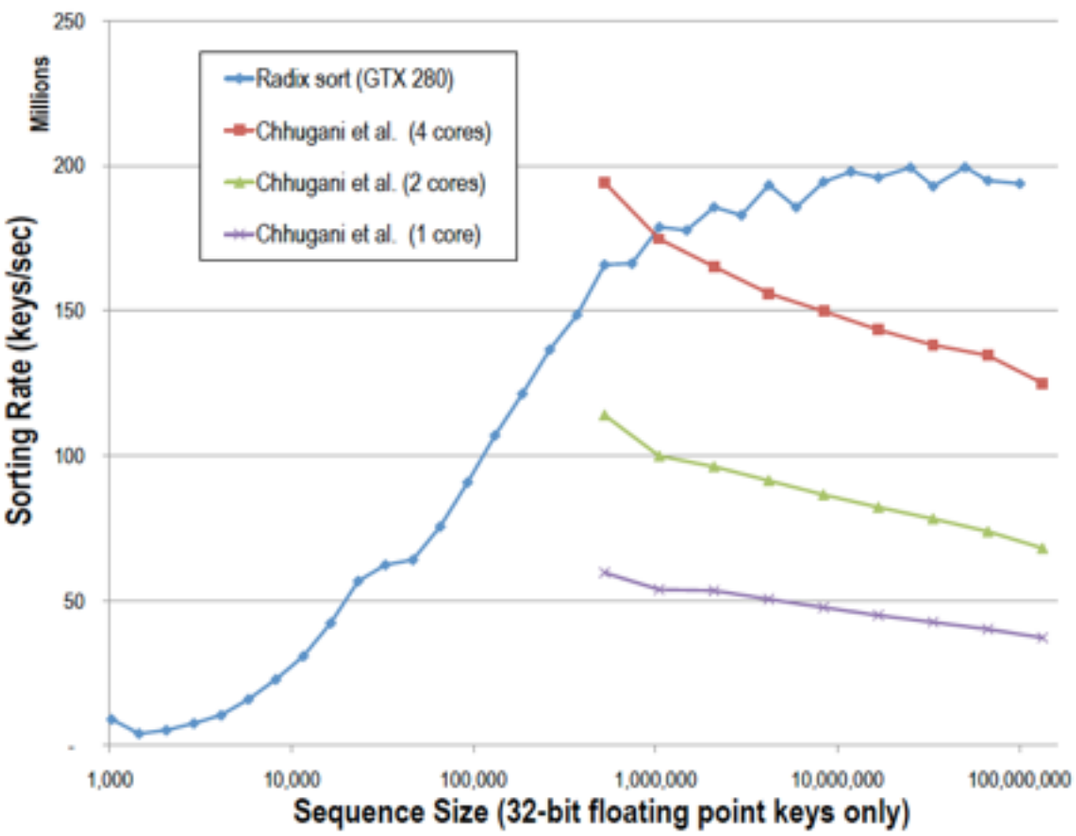


Keeneland system, using one node
3 Nvidia GPUs (M2070 @ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

CUDA软件库-Thrust

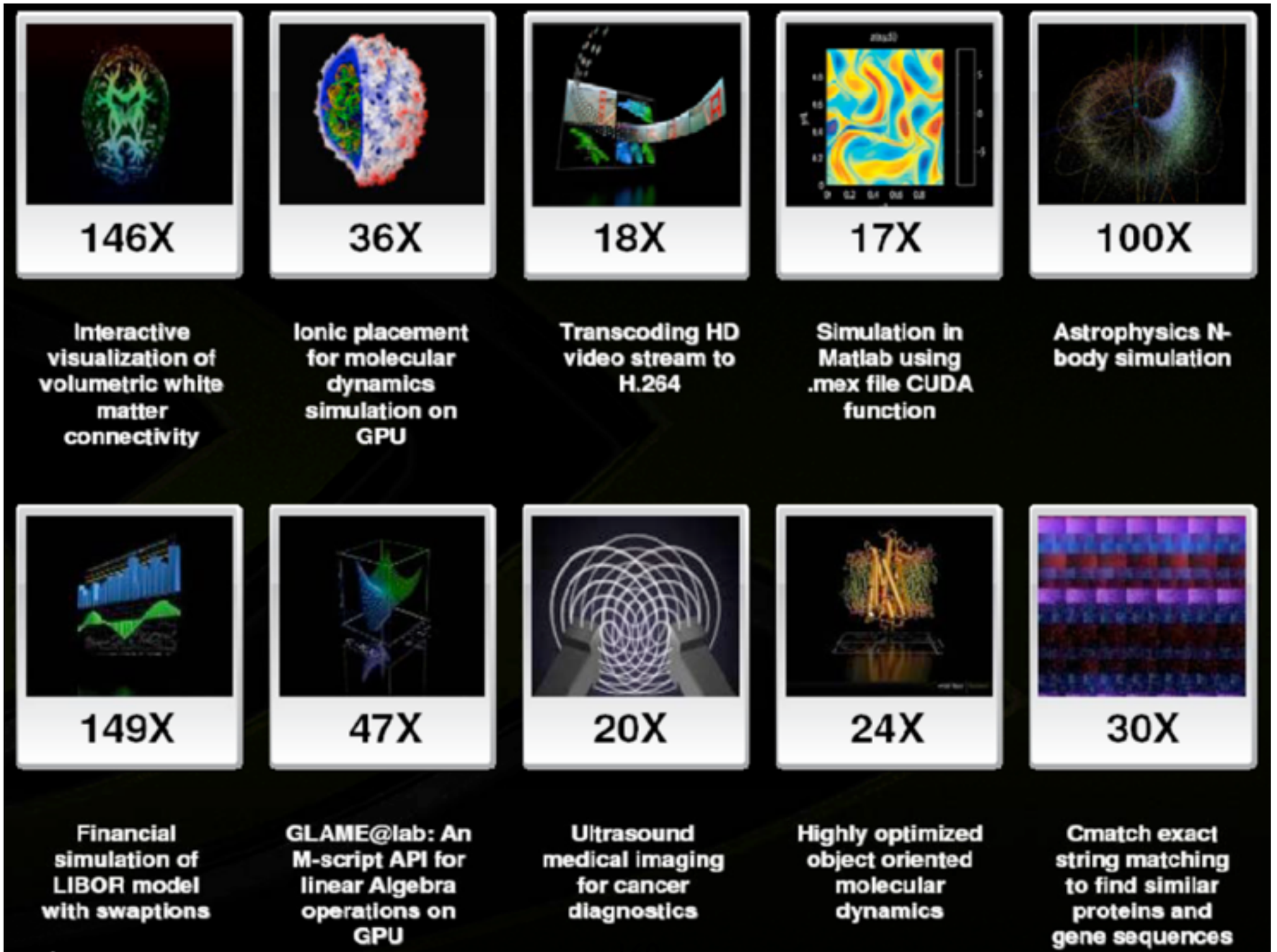


(a) 8-core Clovertown (key-value pairs)



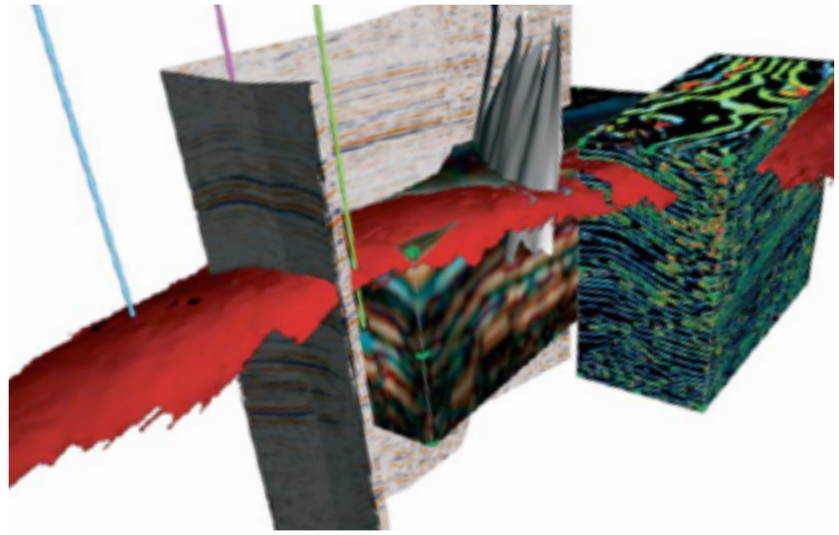
(b) 4-core Yorkfield (float keys only)

CUDA加速计算的广泛应用



GPU应用案例-1

- 三维地质数据分析： ffA
  - 地震数据反演

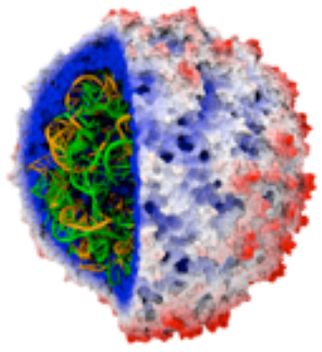


- 37倍加速（CPU+GPU vs CPU）

	配置	功率	性能	价格
系统1	1429台4处理器集群	571 KW	~1T FLOPS	$ 3.1M
系统2	25台4路GPU加速集群	27 KW		$ 310 K

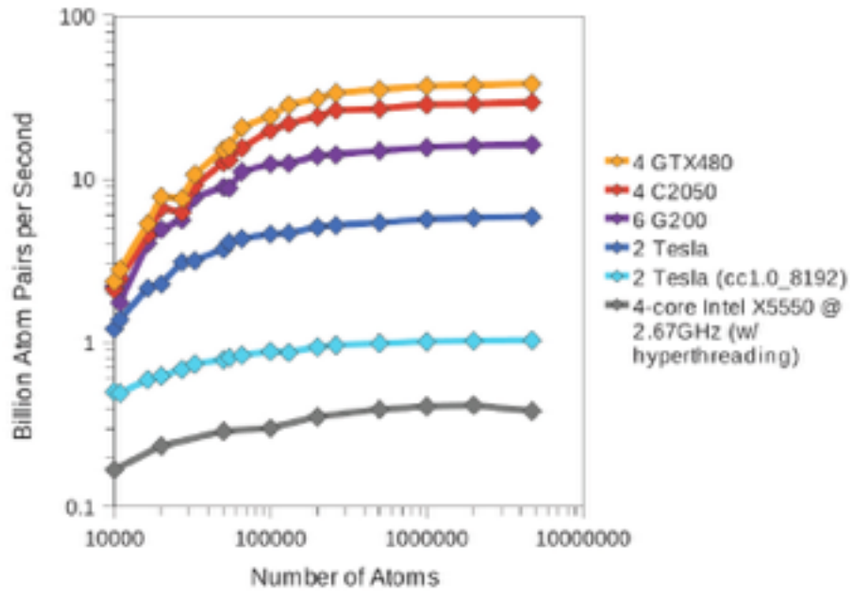
GPU应用案例-2

- 分子动力学模拟（molecular dynamics）



  - RDF：30~92倍加速
  - VMD： 30倍加速
  - http://www.ks.uiuc.edu/Research/gpu/

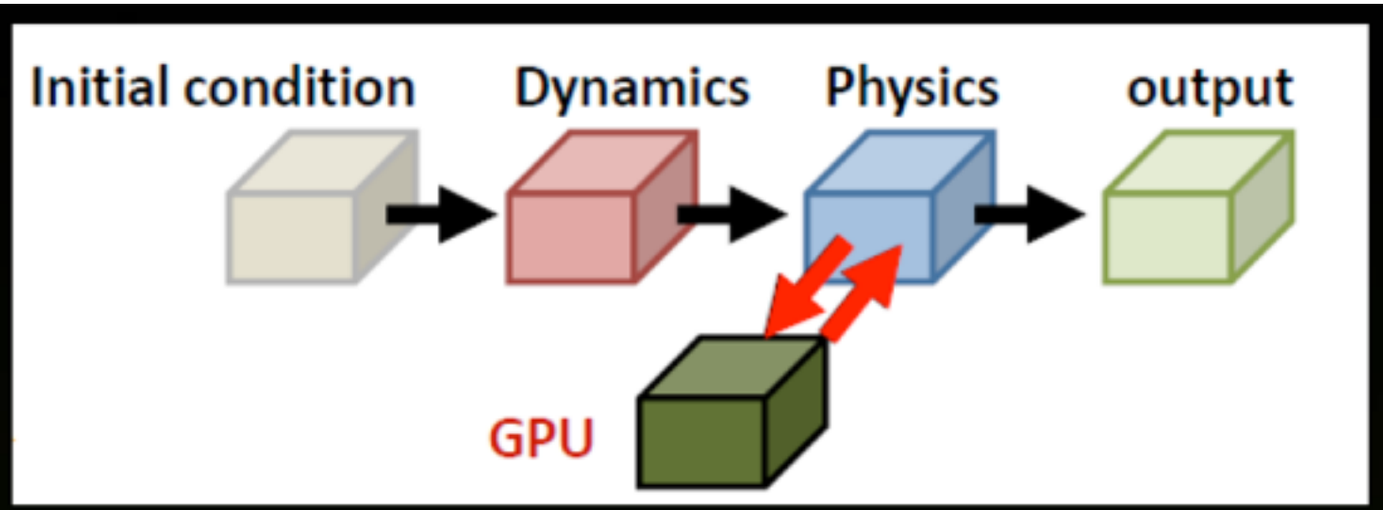
Kernel	Cores/GPUs	Runtime (s)	Speedup
Intel X5550-SSE	1	30.64	1.0
Intel X5550-SSE	8	4.13	7.4
GeForce GTX 480	1	0.255	120
GeForce GTX 480	2	0.136	225
GeForce GTX 480	3	0.098	312
GeForce GTX 480	4	0.081	378



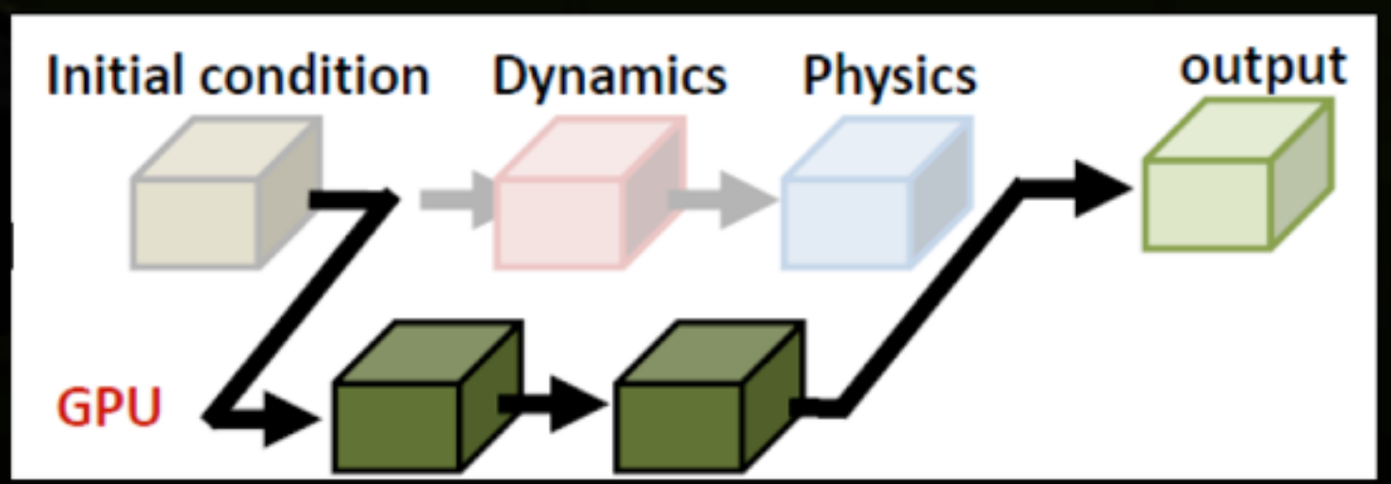
GPU应用案例-3

Physics Only
- WRF

Dynamics Only
- HIRLAM
- HOMME



Full GPU Approach
- ASUCA
- CAM5
- COSMO
- GEOS-5
- GRAPE
- ICON
- NIM

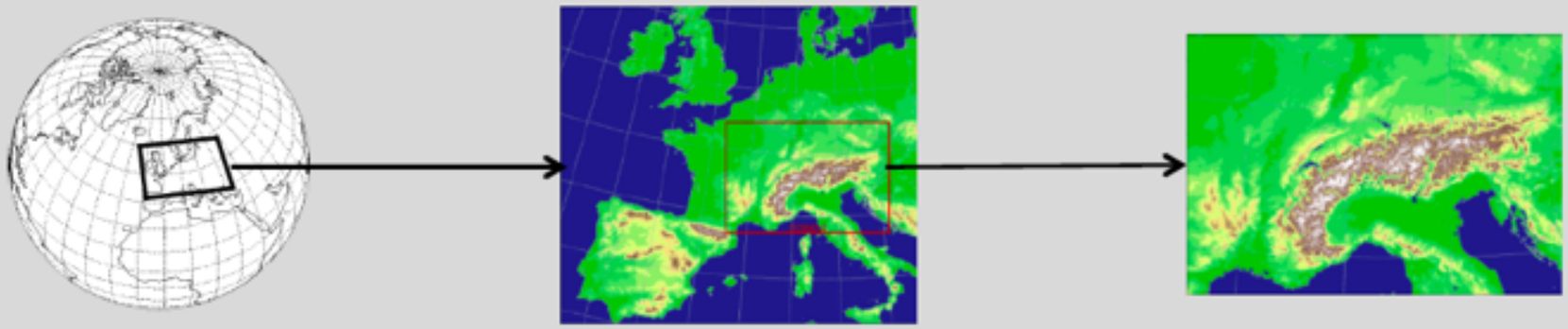


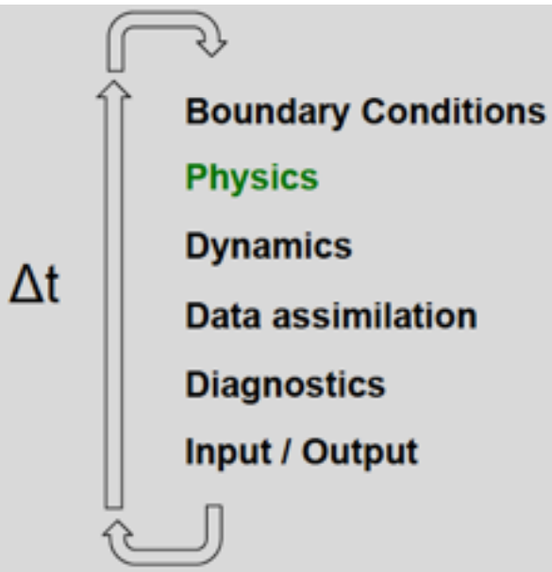
GPU & COSMO (I)

EZMWF-Modell
with 16 km grid
2 x per day
10 day forecast

COSMO-7
with 6.6 km grid
3 x per dat
72 h forecast

COSMO-2
with 2.2 km grid
7 x per day 33 h forecast
1 x per day 45 h forecast





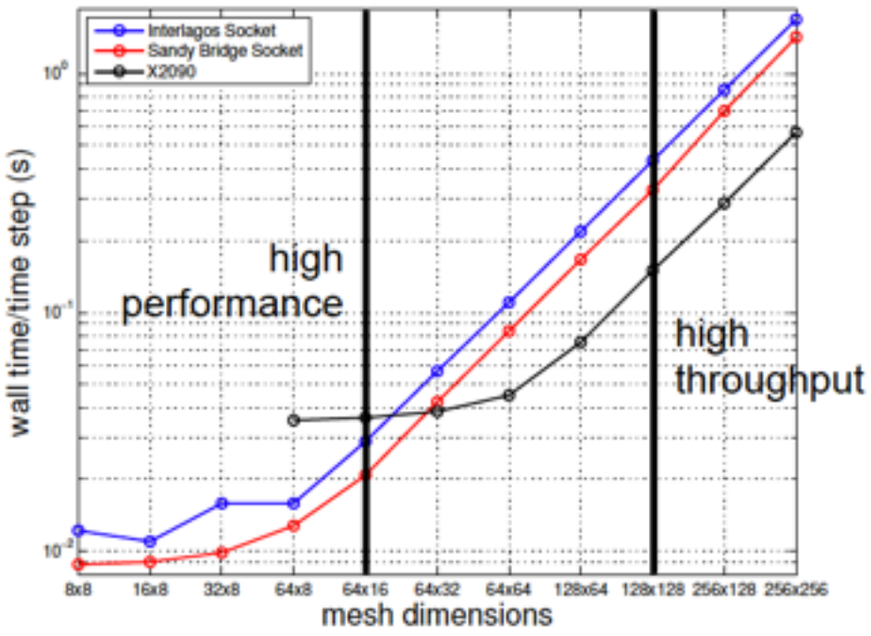
- Large group of developers
- Code may be shared with other models
- Less memory bandwidth bound
- Large part of code (50% of the lines)
- 25% of runtime

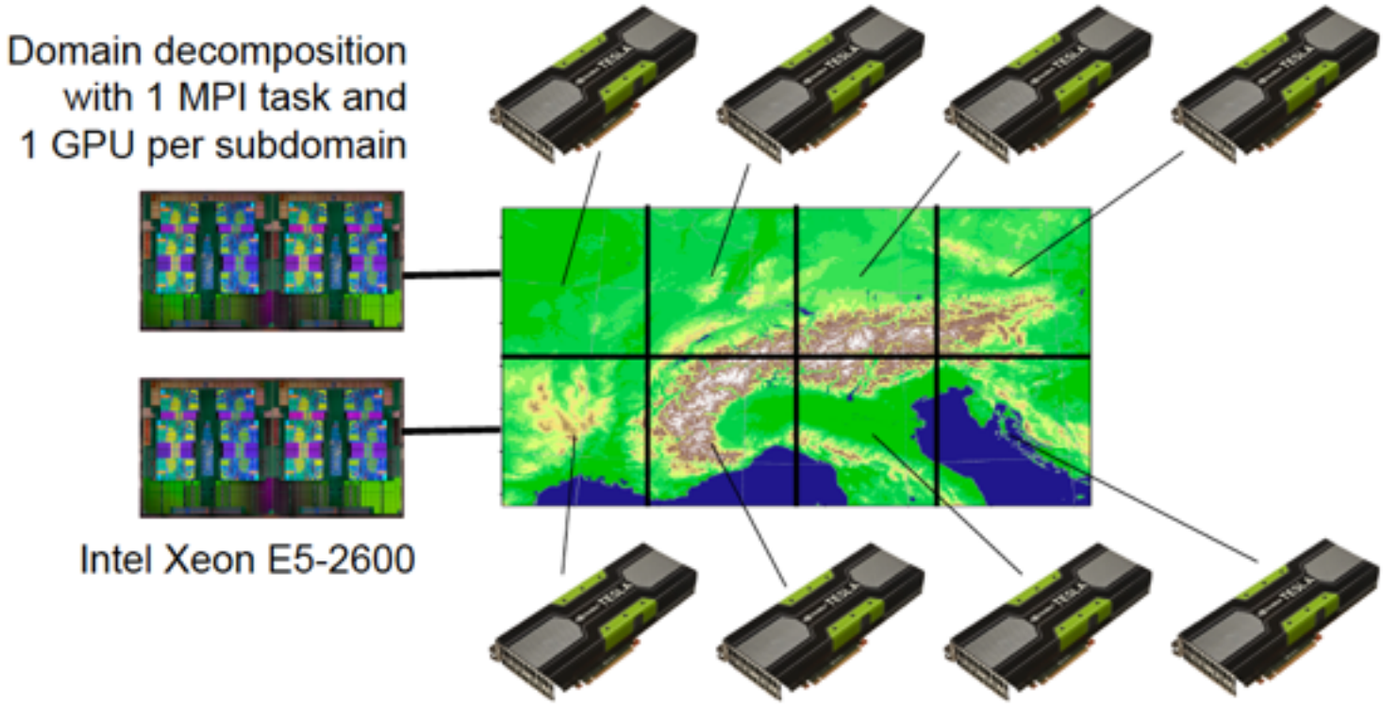
→ **GPU port with compiler directives (OpenACC)**
→ Code optimization
→ Critical routines currently have CPU and GPU version

- Small group of developers
- Memory bandwidth bound
- Complex stencils (3D)
- 60% of runtime

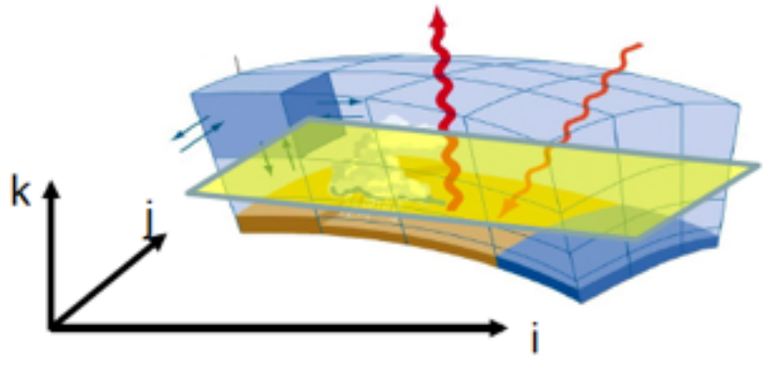
→ **Complete rewrite in C++/CUDA**
→ Development of a stencil library (DSL)
→ Target architectures CPU (x86) and GPU.
→ Extendable to other architectures
→ Long term adaptation of the model

GPU & COSMO (2)

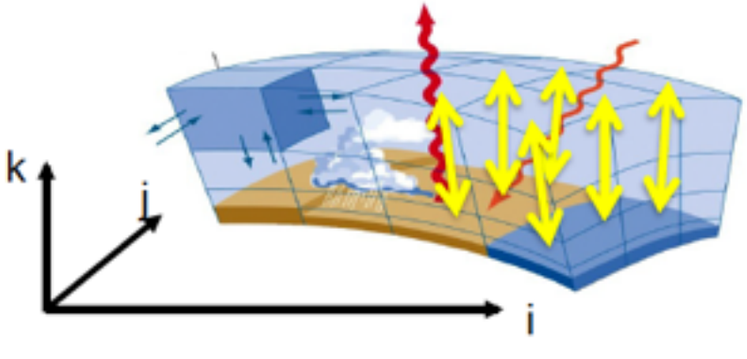




- **Stencils** (finite diferences)
  - Horizontal dependencies
  - No loop carried dependencies



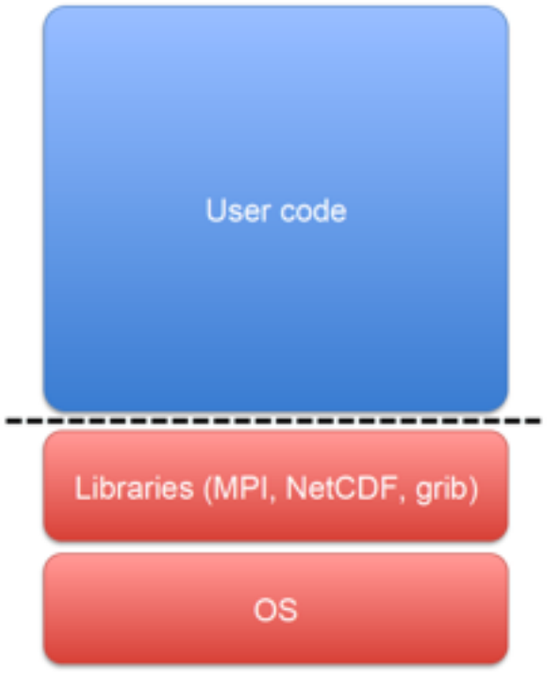
- **Tridiagonal linear solves**
  - Vertical dependencies
  - Loop carried dependencies
  - Parallelizable in horiztonal

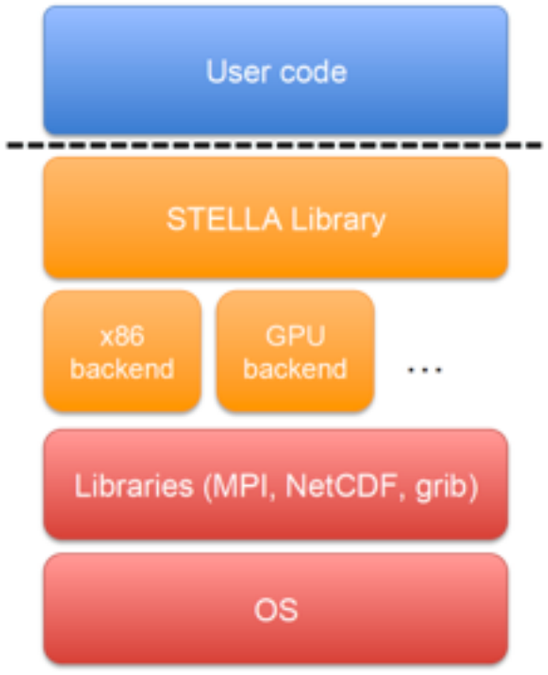


GPU & COSMO (3)

```
DO k = 1, ke
  DO   j  = jstart, jend
    DO i  = istart, iend
      data_out(i,j,k) = -4.0_wp * data_in(i,j,k)              &
                      + data_in(i+1,j,k) + data_in(i-1,j,k)  &
                      + data_in(i,j+1,k) + data_in(i,j-1,k)
    ENDDO
  ENDDO
ENDDO
```

```
// Laplacian stencil
template<typename TEnv>
struct Laplacian
{
    static T Do(Context ctx)
    {
        ctx[data_out::Center()] =
            - (T)4.0 * ctx[data_in::Center()]
            + ctx[data_in::At(iplus1)] + ctx[data_in::At(iminus1)]
            + ctx[data_in::At(jplus1)] + ctx[data_in::At(jminus1)]
    }
};
```

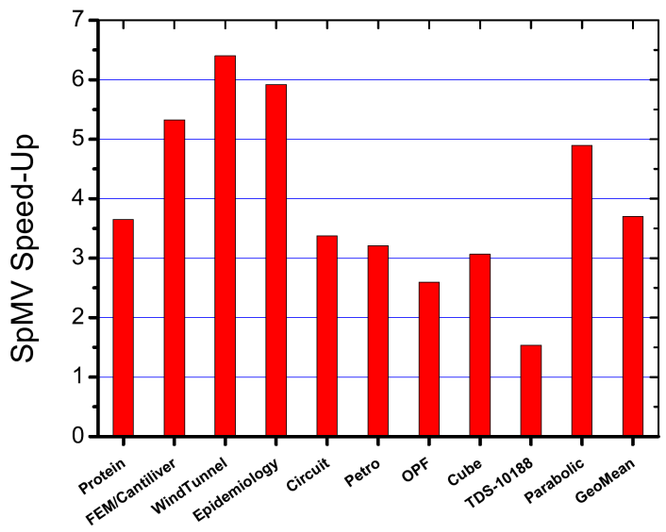


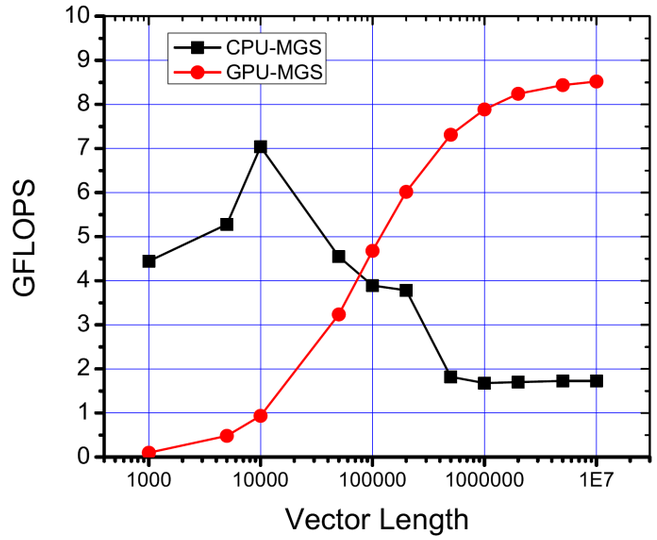



```
// Apply the Laplacian stencil to domain
StencilCompiler::Build(
    stencil_,
    "Laplacian",
    calculationDomain,
    StencilConfiguration<Real, BlockSize<8,8>>(),
    define_loops(
        define_sweep<cKIncrement>(
            define_stages(
                StencilStage<Laplacian,
                             IJRange<cComplete,0,0,0,0>,
                             KRange<FullDomain,0,0> >(),
            )
        )
    )
);
```

GPU应用实例-4

- Krylov迭代法与预处理器
  - 设计适合GPU的预处理器： ML-AINV
  - 迭代法核心操作加速：~6倍





大纲

- 计算机系统与加速计算
- GPU的发展与GPGPU的兴起
- NVIDA CUDA编程与优化
- GPU加速的局限性

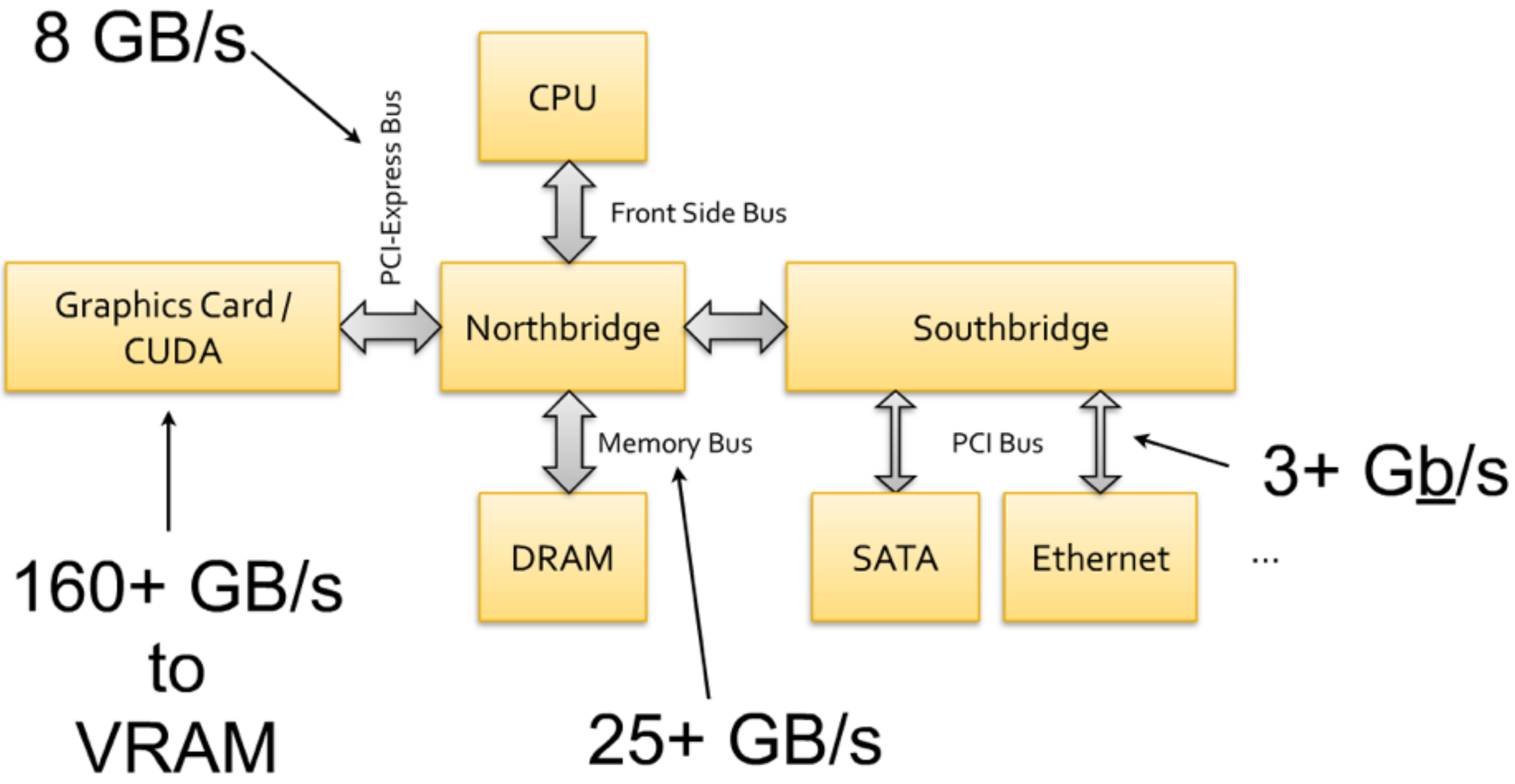
GPU加速的局限

- 性能分析通用原则： Amdahl's Law
- 数据传输瓶颈：
  - 原理与实例
  - 对应用模式的限制
    - MPI+GPU的加速模式

Amdahl's Law与GPU

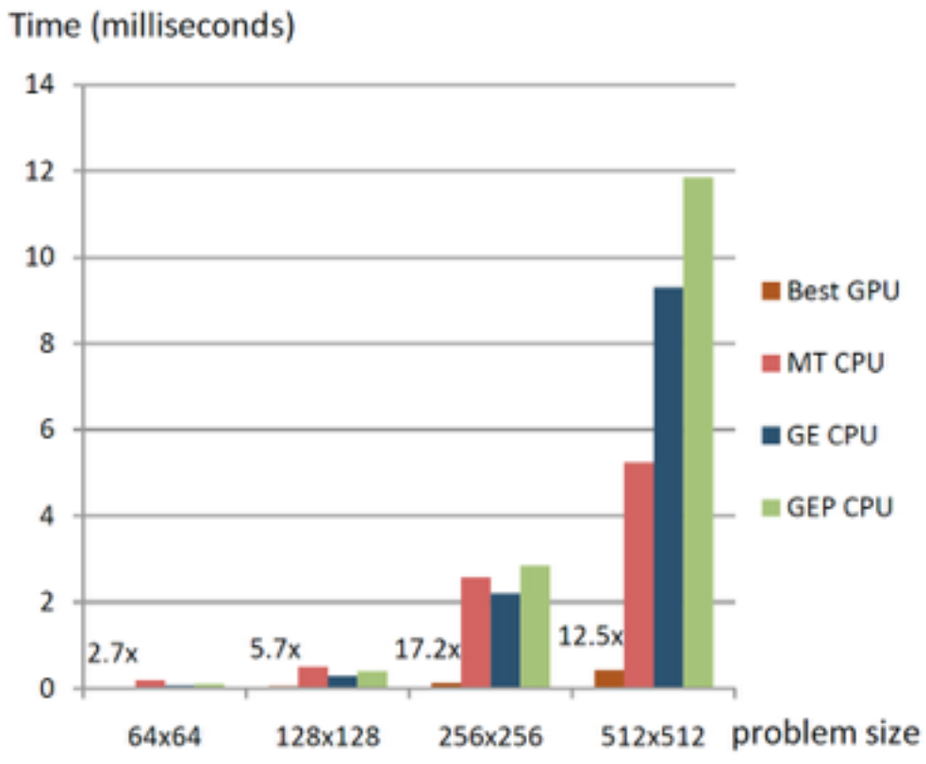
- Amdahl's Law
  - 固有串行部分占整体时间的比例为a
    - 能达到的加速比上限为1/a
- 适用于任何并行化/加速/性能优化工作
  - 程序热点的性能提升反应在整体性能提升上会打折扣

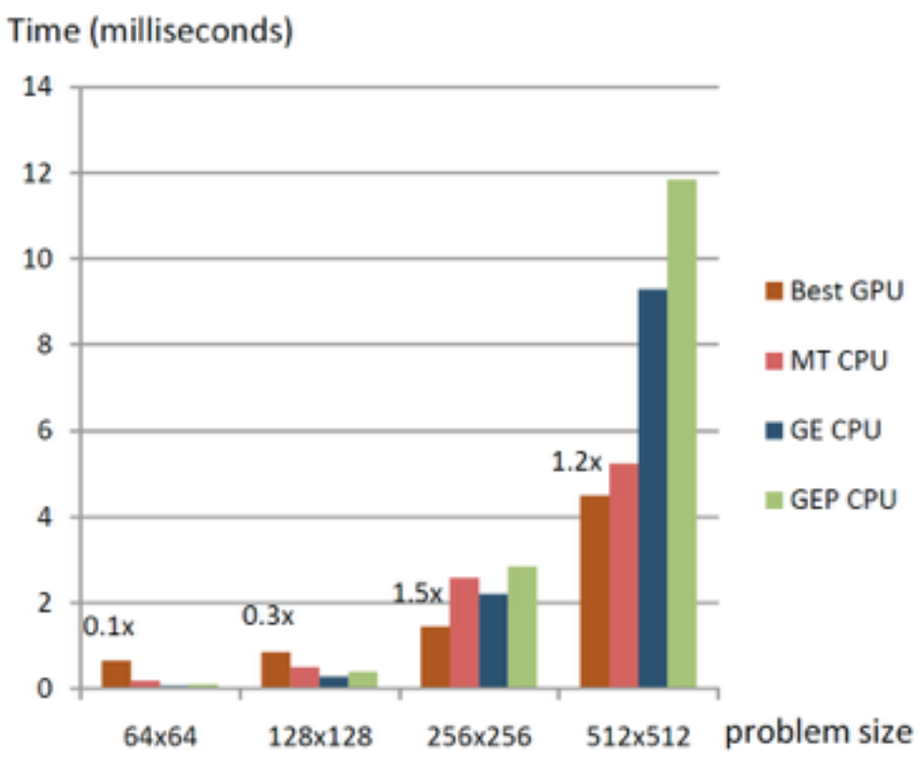
GPU与计算机系统


8 GB/s
PCI-Express Bus
CPU
Front Side Bus
Graphics Card / CUDA
Northbridge
Southbridge
Memory Bus
PCI Bus
3+ Gb/s
DRAM
SATA
Ethernet
...
160+ GB/s to VRAM
25+ GB/s

问题?

- 三对角矩阵分解/求解问题
  - 分解复杂度为O(n)（大致为5n/3）
- 将矩阵传递至GPU时，CPU即可基本完成求解





影响加速计算的软件模式

- 当PCI-E成为瓶颈时，考虑将更多的计算部分、甚至整体应用移植至GPU
  - 避免数据拷贝
  - CPU仅负责IO与通信
- 例子：
  - 地球模式：ASUCA等

加速计算与集群

- 与GPU直接交换数据，其延时与带宽均与MPI访问远程节点相当
- 数据路径：
  - GPU <=> CPU <=> 网络
- 天河-2 & Titan： 实际效率<65%

总结

- GPU加速计算方兴未艾
- NVIDIA CUDA 的编程与运行模型
- 了解应用的性能行为、明确优化目标
- 科学看待加速计算、扬长避短

谢谢大家！