CUDA / CUDNN / NCCL 最新特性介绍

David Wu(吴磊), 2018.11.21

提纲

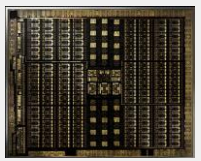
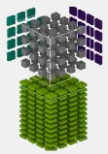
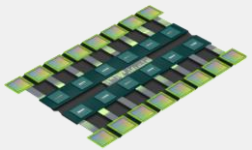
1. CUDA 10.0
    1) CUDA 加速库
    2) Nsight 开发工具家族
    3) 新的编程模型 - Graph
    4) CUDA 部署方式
    5) Turing 架构的支持
2. cuDNN 7.3
3. NCCL 2.3

CUDA 10.0

https://developer.nvidia.com/cuda-toolkit

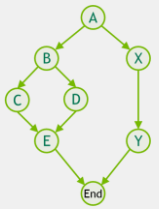
支持TURING 和最新系统

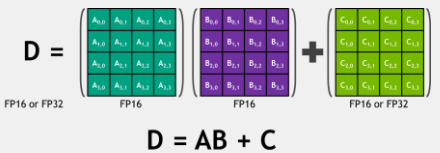
新的 GPU 架构, Tensor Cores, NVSwitch 结构


CUDA 平台

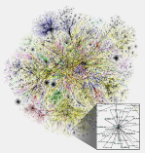
CUDA Graphs, Vulkan & DX12 中间件, Warp 矩阵计算模型





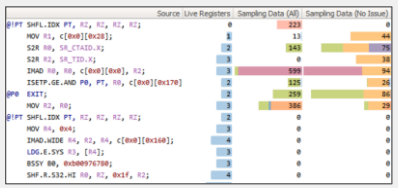
加速库

基于GPU的混合JPEG解码, 对称特征求解器, FFT 扩展计算


开发者工具

新的 Nsight 家族产品 – Nsight Systems 和 Nsight Compute


CUDA 加速库

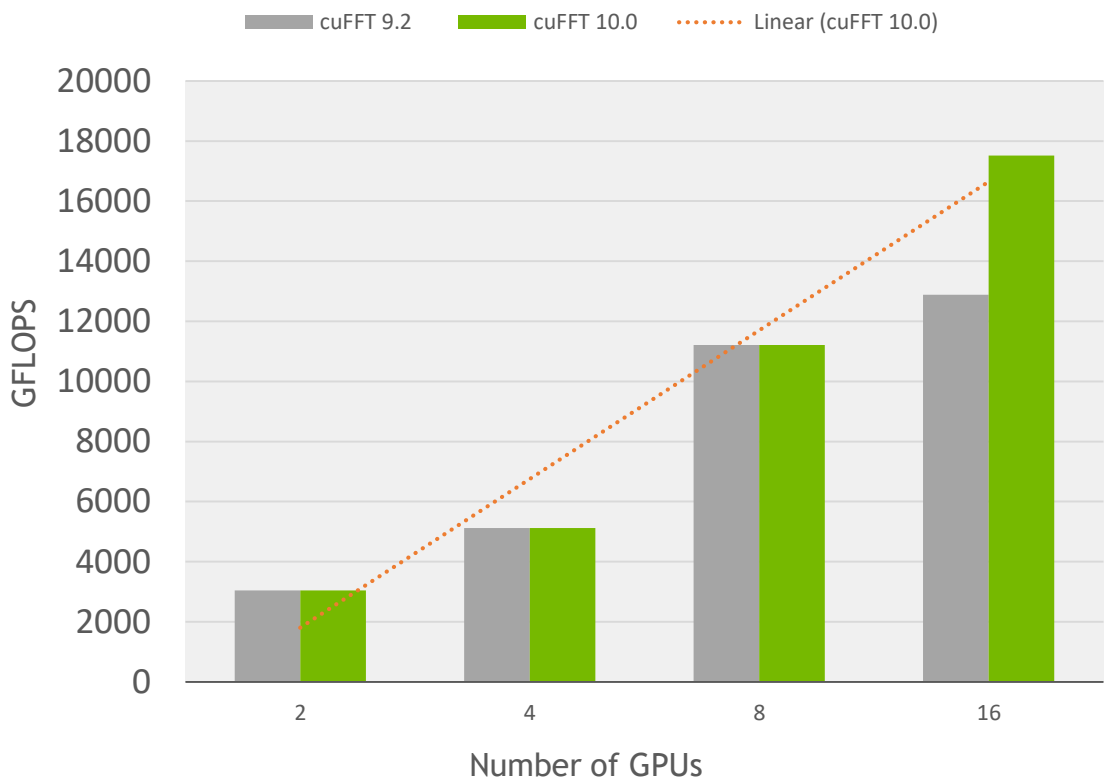
cuFFT 10.0

支持DGX-2 和 HGX-2 的多GPU扩展

- 最多支持扩展的16-GPU 系统 ：DGX-2 和 HGX-2
- 支持多GPU的 R2C 和 C2R
- 扩展到16-GPUs的超大FFT模型 – 每个GPU 32GB vs 总共512GB有效容量

https://developer.nvidia.com/cufft

Up to 17TF performance on 16-GPUs 3D 1K FFT



cuFFT (10.0 and 9.2) using 3D C2C FFT 1024 size on DGX-2 with CUDA 10 (10.0.130)

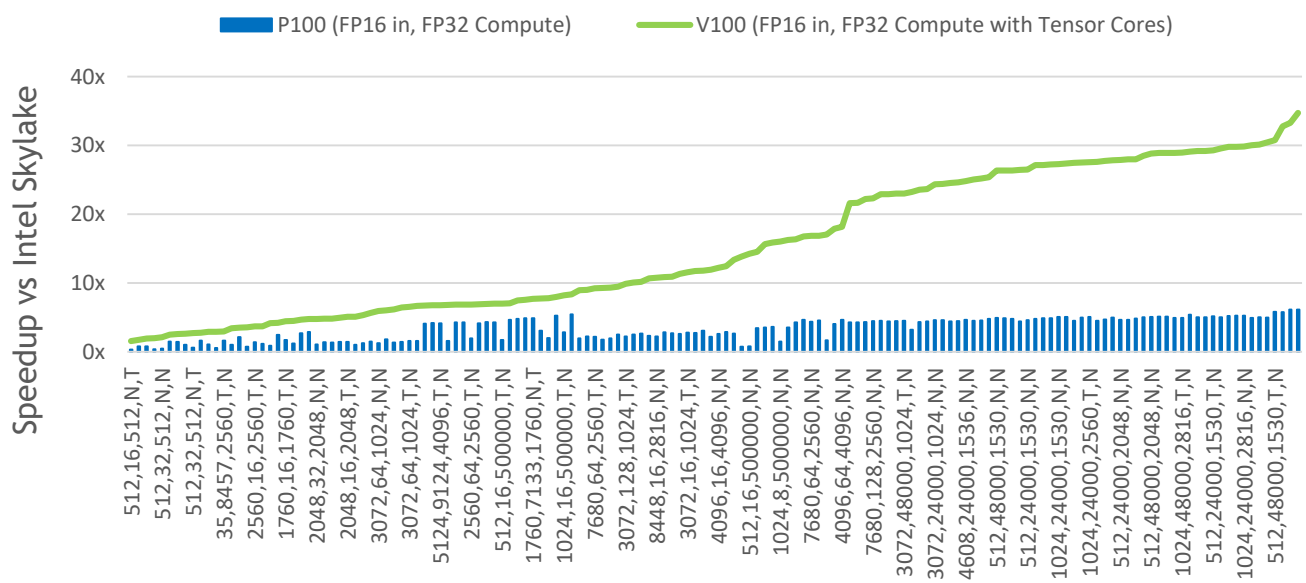
cuBLAS 10.0

专为深度学习优化的GEMM

- 基于Turing架构优化的 GEMMs 以及 GEMM 扩展，可支持 Tensor Cores
- 在不同深度学习模型中，基于不同矩阵规模的GEMM 性能微调
- 用于调试和追踪的API和错误日志

https://developer.nvidia.com/cublas

最高可达90TF Deepbench GEMM 性能



Deepbench training performance runs with Tesla P100, Tesla V100 and Intel Skylake 6140 Gold 2.3 GHz with hyperthreading off

cuSOLVER 10.0

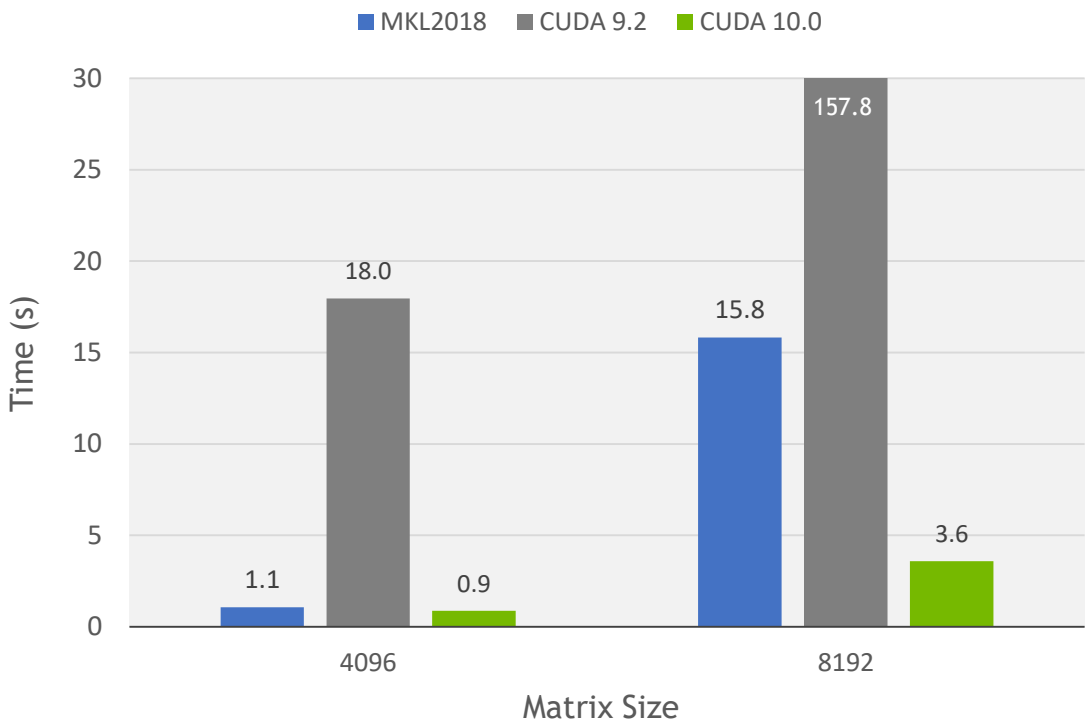
稠密线性矩阵

改进了性能的算法：

- Cholesky 分解
- 对称和广义对称特征解算器
- QR 分解

https://developer.nvidia.com/cusparse

对称特征求解器算法最高可达44x加速 (DSYEVD)



Benchmarks use 2 x Intel Gold 6140 (Skylake) processors with Intel MKL 2018 and NVIDIA Tesla V100 (Volta) GPUs

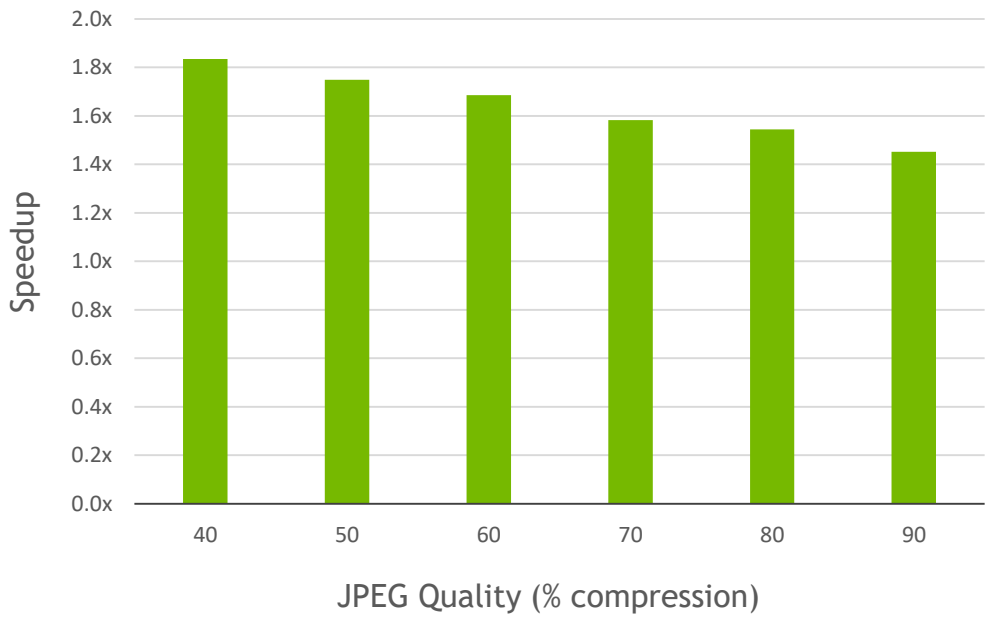
nvJPEG 10.0

GPU加速混合JPEG解码器

- 利用CPU和GPU实现低延迟的混合解码
- 支持单个或批量图像解码，获得最佳延迟或吞吐性能
- 支持多种分辨率和二次采样模式
- 支持RGB, BGR, RGBI, BGRI, YUV等颜色空间转换

https://developer.nvidia.com/nvjpeg

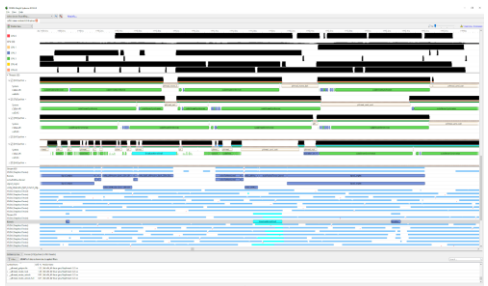
JPEG解码最高可达1.8x加速



JPEG decoding performance (images/sec) on Tesla V100 vs. libjpeg-turbo on Intel Skylake CPU 6140 @ 2.3GHz hyperthreading off. Image size: 640x480. Decoding performed on various JPEG compression/quality levels as described by imagemagick library's quality parameter.

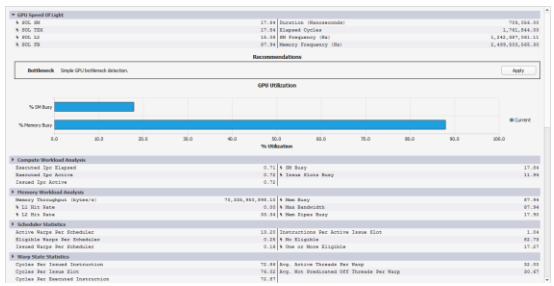

NSIGHT
开发者工具

NSIGHT 产品家族



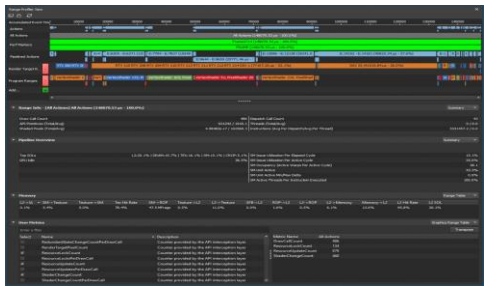
Nsight Systems

系统级应用算法调优



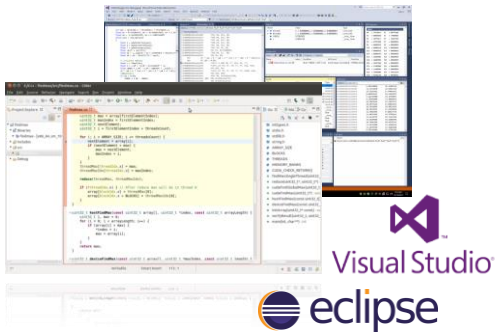
Nsight Compute

CUDA Kernel函数分析和调试



Nsight Graphics

图形渲染分析和调试



IDE 插件

Nsight Eclipse Edition/Visual Studio (编辑器，调试器)

NSIGHT SYSTEMS

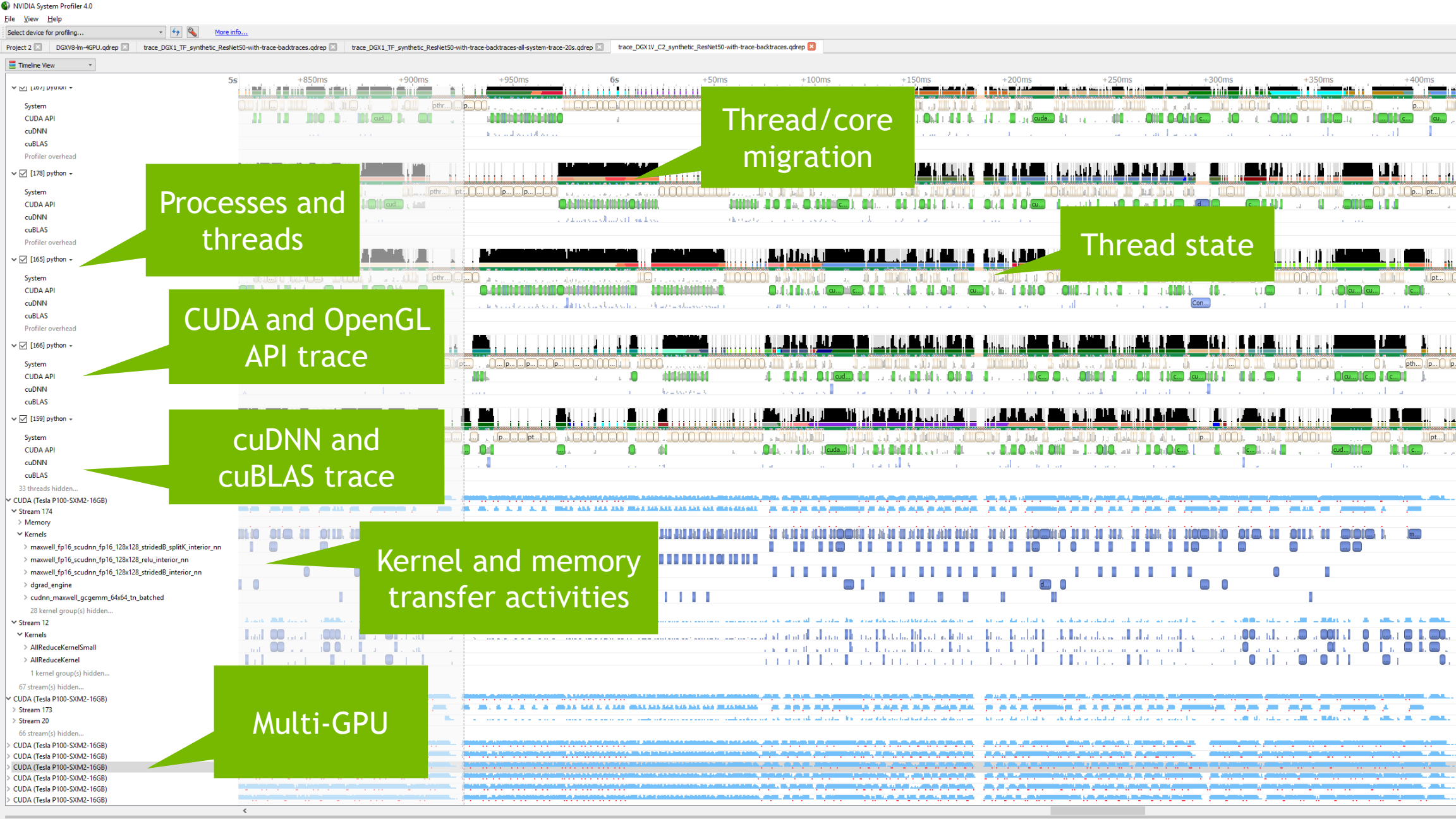
系统级性能分析

观测应用程序行为：CPU 线程，GPU 追踪，内存带宽，…

定位优化机会：CUDA & OpenGL APIs, Unified Memory 迁移，NVTX用户附注

支持大数据量：超过1000万个事件的可视化，支持 Container，最小化用户权限

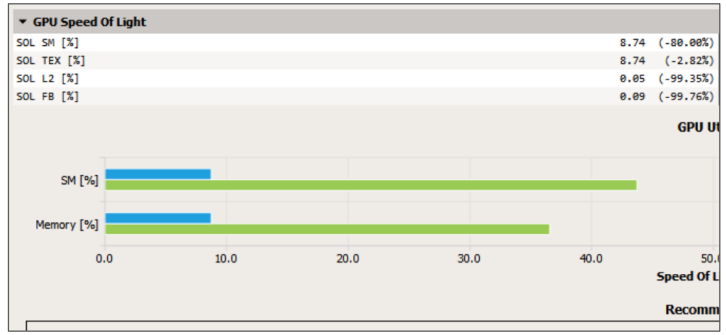
https://developer.nvidia.com/nsight-systems


NVIDIA System Profiler 4.0
File View Help
Select device for profiling...
More info...
Project 2
DGXV8-lm-4GPU.qdrep
trace_DGX1_TF_synthetic_ResNet50-with-trace-backtraces.qdrep
trace_DGX1_TF_synthetic_ResNet50-with-trace-backtraces-all-system-trace-20s.qdrep
trace_DGX1V_C2_synthetic_ResNet50-with-trace-backtraces.qdrep
Timeline View
5s
+850ms
+900ms
+950ms
6s
+50ms
+100ms
+150ms
+200ms
+250ms
+300ms
+350ms
+400ms
[178] python
[165] python
[166] python
[159] python
System
CUDA API
cuDNN
cuBLAS
Profiler overhead
33 threads hidden...
CUDA (Tesla P100-SXM2-16GB)
Stream 174
Memory
Kernels
maxwell_fp16_scudnn_fp16_128x128_stridedB_splitK_interior_nn
maxwell_fp16_scudnn_fp16_128x128_relu_interior_nn
maxwell_fp16_scudnn_fp16_128x128_stridedB_interior_nn
dgrad_engine
cudnn_maxwell_gcgemm_64x64_tn_batched
28 kernel group(s) hidden...
Stream 12
AllReduceKernelSmall
AllReduceKernel
1 kernel group(s) hidden...
67 stream(s) hidden...
Stream 173
Stream 20
66 stream(s) hidden...
Thread/core migration
Processes and threads
Thread state
CUDA and OpenGL API trace
cuDNN and cuBLAS trace
Kernel and memory transfer activities
Multi-GPU

NVIDIA NSIGHT COMPUTE

下一代kernel函数分析器

- 交互式 CUDA API 调试和 kernel 函数分析
- 快速数据收集
- 改进的工作流程和可定制化(基线, 可编程的用户界面/规则)
- 命令行方式, 独立方式, IDE 集成方式
- 平台支持
  - OS: Linux (x86, ARM), Windows
  - GPUs: Pascal, Volta, Turing



Kernel Profile Comparisons with Baseline

inst_executed [inst]	16,528.00; 16,528.00; …	13,476.00; 13,476.00; …
l1tex__sol_pct [%]	14.33	n/a
launch__block_size	128.00	128.00
launch__function_pcs	47,611,587,968.00	12,273,728.00
launch__grid_size	4,132.00	3,369.00
launch__occupancy_limit_blocks [block]	32.00	32.00
launch__occupancy_limit_registers [register]	21.00	21.00
launch__occupancy_limit_shared_mem [bytes]	384.00	384.00
launch__occupancy_limit_warps [warps]	16.00	16.00
launch__occupancy_per_block_size	3,638.00	3,638.00
launch__occupancy_per_register_count	5,792.00	5,792.00
launch__occupancy_per_shared_mem_size	2,260.00	2,260.00
launch__registers_per_thread [register/thread]	17.00	17.00
launch__shared_mem_config_size [bytes]	49,152.00	49,152.00
launch__shared_mem_per_block_dynamic [bytes/block]	0.00	0.00
launch__shared_mem_per_block_static [bytes/block]	20.00	20.00
launch__thread_count [thread]	528,896.00	431,232.00
launch__waves_per_multiprocessor	3.23	42.11
ltc__sol_pct [%]	6.93	7.18
memory_access_size_type [bytes]	2.00; 32.00; 32.00; 32…	2.00; 32.00; 32.00; 32…

Metric Data

Source	Live Registers	Sampling Data (All)	Sampling Data (No Issue)
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;	0	223	0
MOV R1, c[0x0][0x28];	1	13	44
S2R R0, SR_CTAID.X;	2	143	75
S2R R2, SR_TID.X;	3	0	38
IMAD R0, R0, c[0x0][0x0], R2;	3	599	94
ISETP.GE.AND P0, PT, R0, c[0x0][0x170]	2	125	26
@P0 EXIT;	2	259	86
MOV R2, R0;	3	386	29
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;	2	0	0
MOV R4, 0x4;	3	0	0
IMAD.WIDE R4, R2, R4, c[0x0][0x160];	4	0	0
LDG.E.SYS R3, [R4];	3	0	0
BSSY B0, 0xb00976780;	3	0	0
SHF.R.S32.HI R0, RZ, 0x1f, R2;	4	0	0

Source Correlation

编程模型

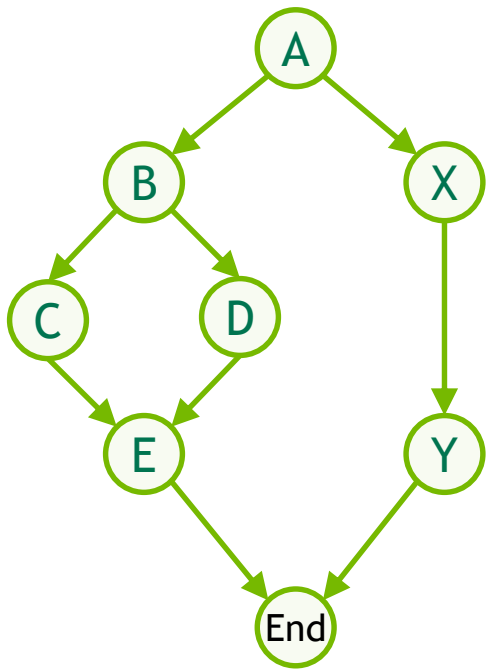
CUDA GRAPH 的定义

图节点并不仅仅指kernel函数的加载

一系列的操作，由依赖关系相连接.

这些操作包含:

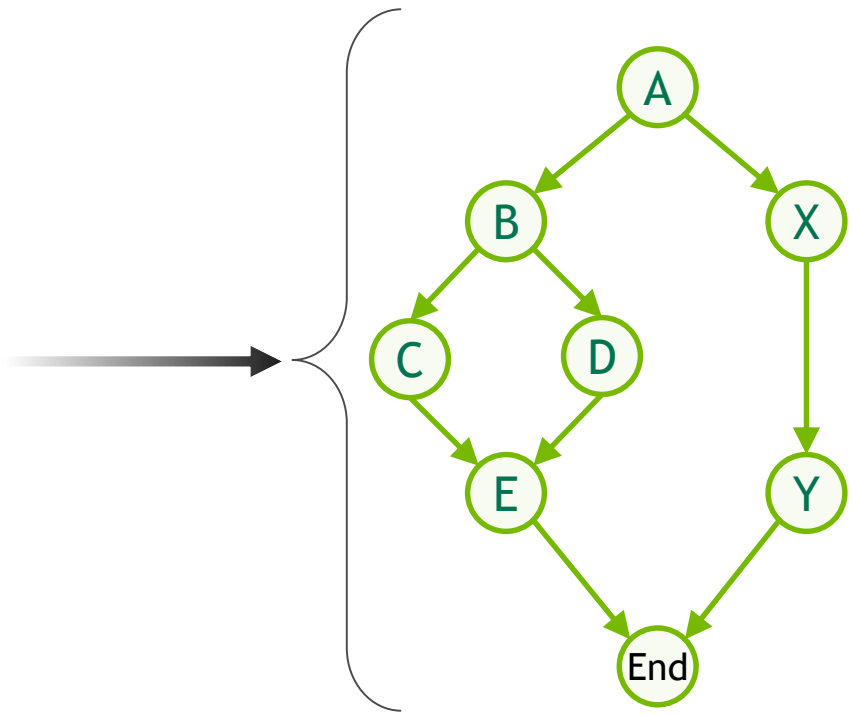
Kernel Launch	CUDA kernel 函数
CPU Function Call	CPU 上的回调函数
Memcopy/Memset	GPU 数据管理
Sub-Graph	图是可分级的



新的执行机制

图可以只产生一次，然后被重复的加载

```
for(int i=0; i<1000; i++) {
    launch_graph( G );
}
```

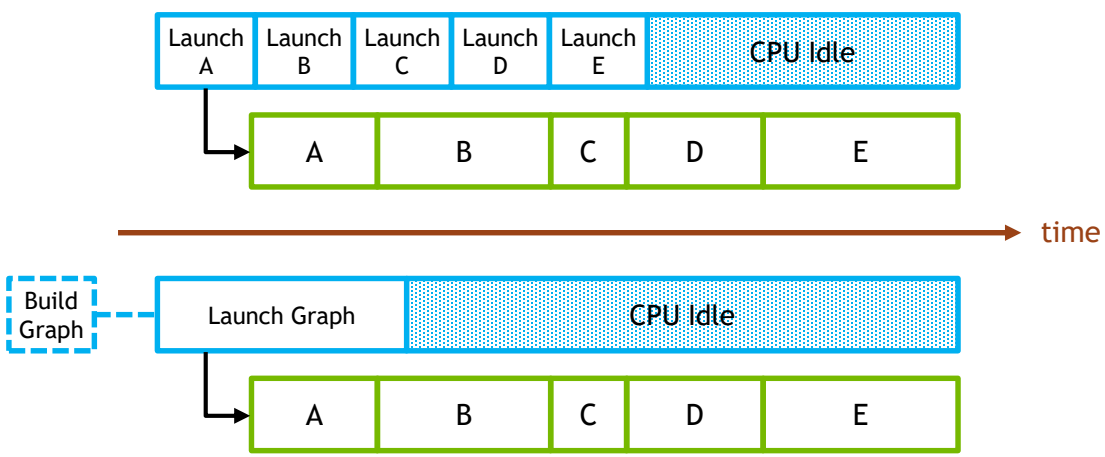


执行优化

降低延迟和开销

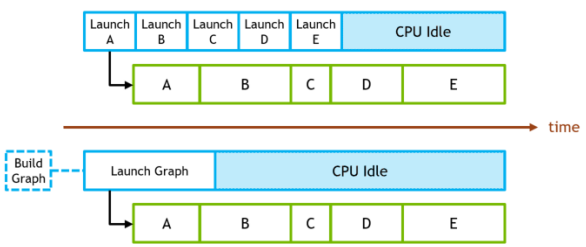
加载kernel函数的延迟开销：

- CUDA 10.0 在linux系统中至少需要2.2us 的时间来加载1个CUDA kernel
- 预先定义的图允许在1次操作中加载任意个数的kernel函数



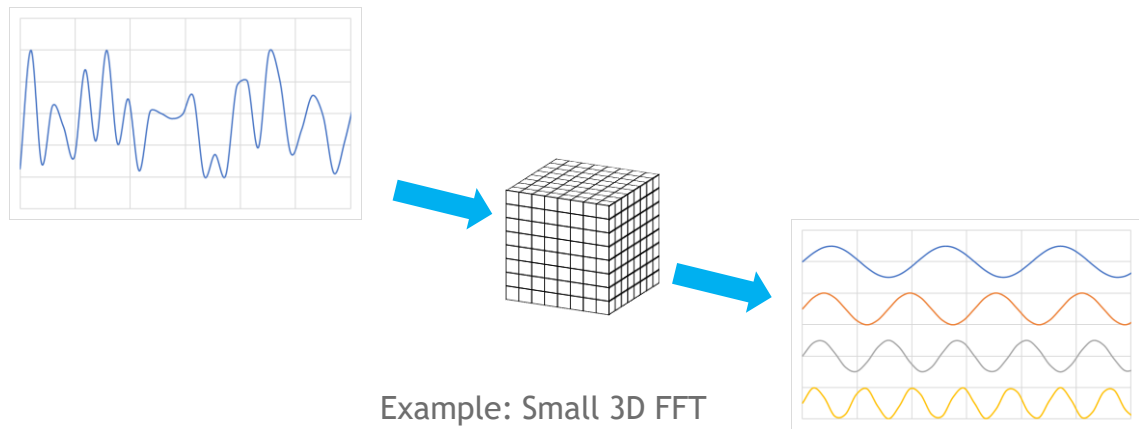
性能影响

对运行时间较短的kernel有明显的优化效果



CPU launch time improvements

典型情况：相比于Stream加载方式有33%加速



Example: Small 3D FFT

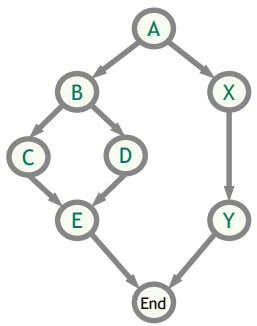
对于32^3 3D-FFT有25%的优化效果
(stream加载需要16us，graph 加载需要12us)

注意：性能影响是基于工作负载独立的条件下的

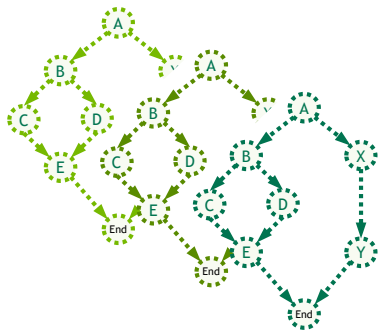
主要为运行时间较短的kernel函数带来好处，此类情况下kernel函数的加载开销相对较大

执行模型的三个阶段

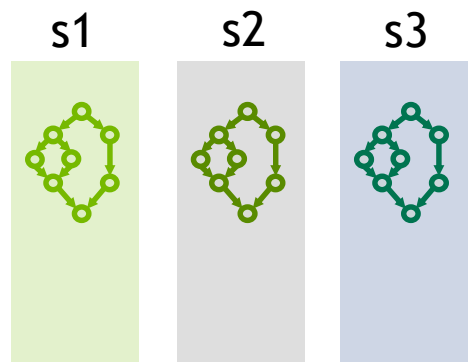
定义



实例化



执行



单个图“模板”

在主机代码中创建，或者
在库中建立

多个“可执行图”

模板设置的快照 & 实例化GPU
执行结构
(创建一次，对此执行)

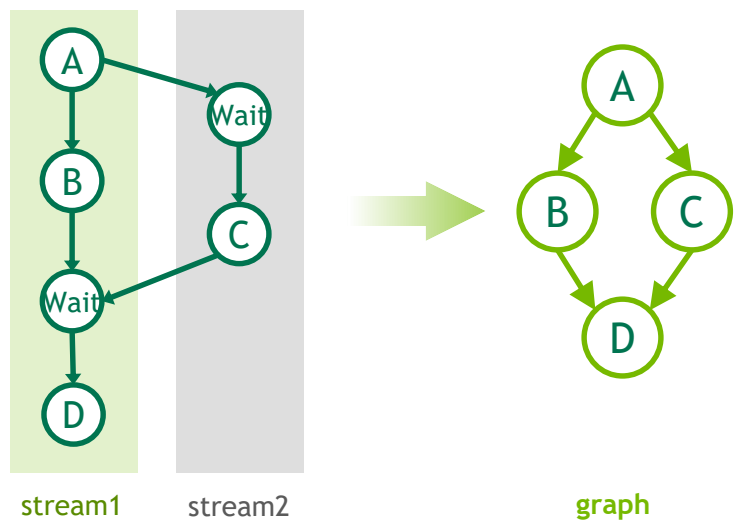
可执行图运行在cuda
stream中

图中的并发不受stream的
限制

转换 CUDA STREAM 到图中

通过普通的 CUDA STREAM 语法来构建图

```
// Start by initating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

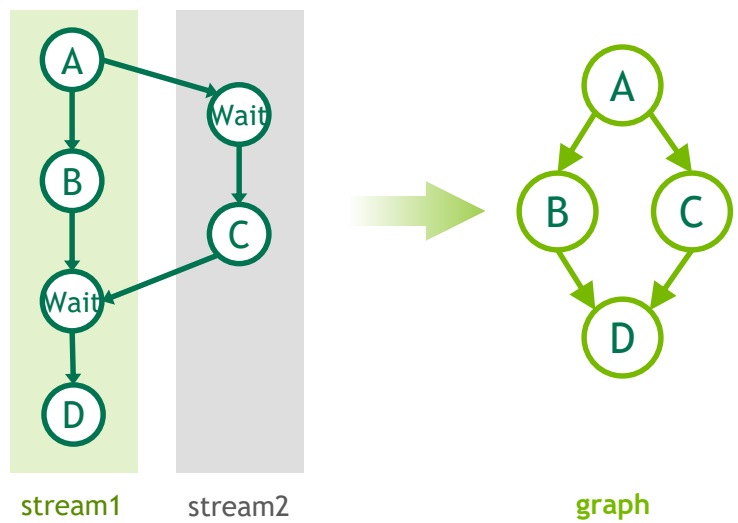


转换 CUDA STREAM 到图中

通过普通的 CUDA STREAM 语法来构建图

```
// Start by initating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

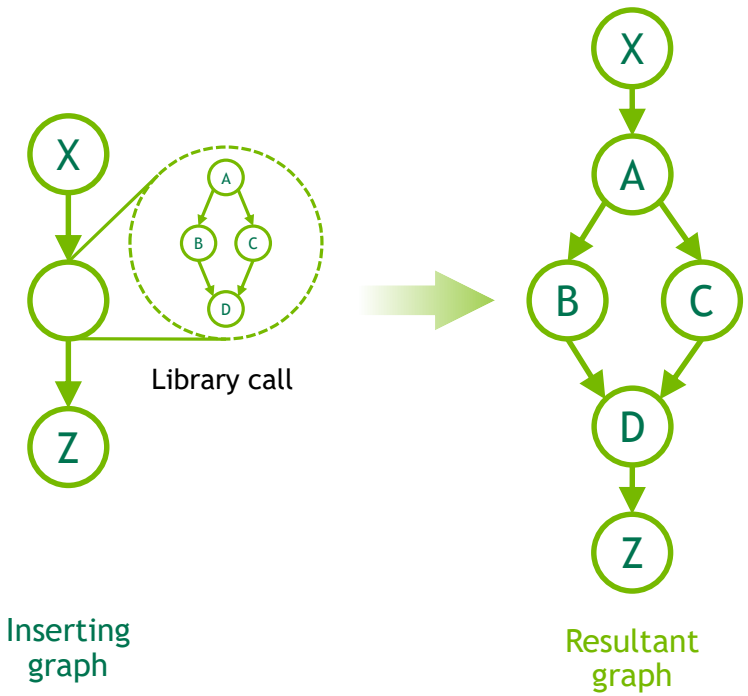
捕获stream间的依赖来创建分叉和联接



捕获外部任务

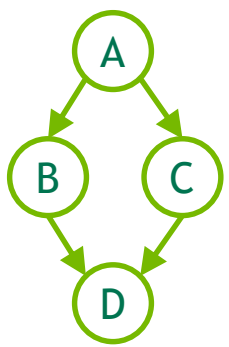
Stream 捕获可以持续到库调用中

```
// Start by initating stream capture
cudaStreamBeginCapture(&stream);

// Captures my kernel launches, recurse into library calls
X<<< ..., stream >>>();
libraryCall(stream);        // Launches A, B, C, D
Z<<< ..., stream >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream, &graph);
```



直接创建图

将基于图的工作流映射到 CUDA 中



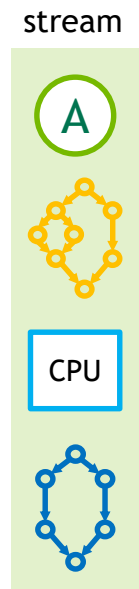
Graph from framework

```
// Define graph of work + dependencies
cudaGraphCreate(&graph);

cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times
for(int i=0; i<100; i++)
      cudaGraphLaunch(instance, stream);
```

图执行的语义

基于图化的任务与其他非图化的CUDA任务一起执行

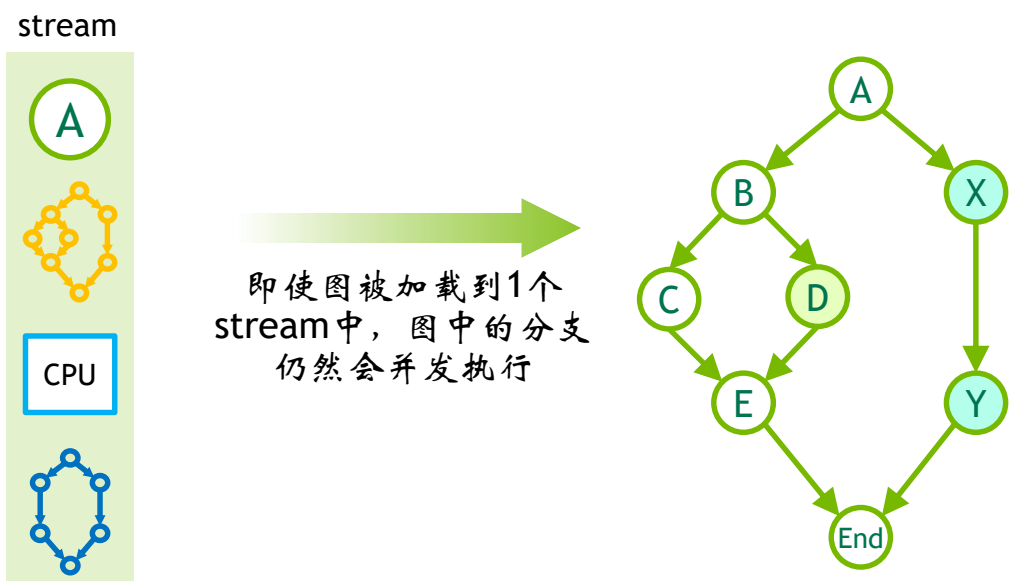
```
launchWork(cudaGraphExec_t i1, cudaGraphExec_t i2,
           CPU_Func cpu, cudaStream_t stream) {

    A <<< 256, 256, 0, stream >>>();        // Kernel launch
    cudaGraphLaunch(i1, stream);            // Graph1 launch
    cudaStreamAddCallback(stream, cpu);     // CPU callback
    cudaGraphLaunch(i2, stream);            // Graph2 launch

    cudaStreamSynchronize(stream);
}
```



如果能放入cuda stream中，它就可以和图一起执行

图会忽略STREAM的串行规则

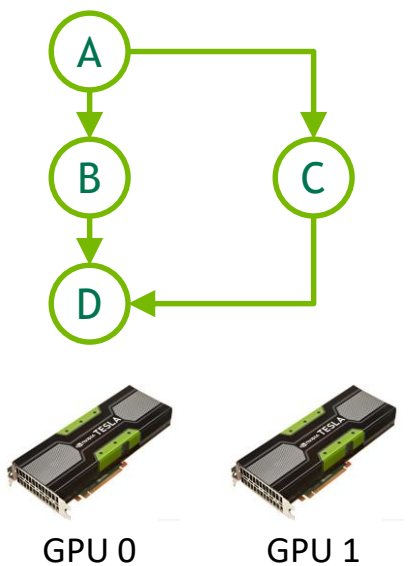
图加载到stream中，仅会影响到它与其他任务的执行顺序



交叉设备依赖关系

图可以跨越多个GPU

多GPU执行



异构执行

CUDA最接近操作系统和硬件

- 可以优化多GPU设备间的依赖关系
- 可以优化CPU+GPU异构的依赖关系
- 在每个节点上定义局部性

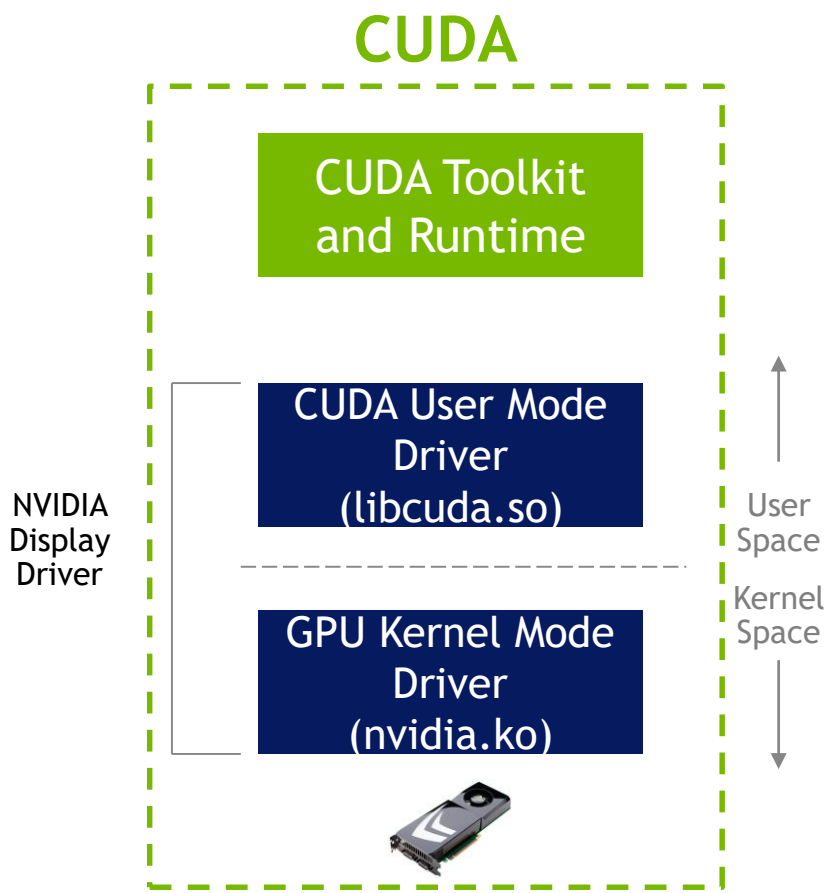

CUDA 部署

CUDA安装组件

CUDA由以下三个部件组成：

1. CUDA Toolkit (程序编译)
2. CUDA User Mode Driver (程序执行)
3. NVIDIA Kernel Mode Driver (程序执行)

注意：部件2和3在NVIDIA驱动包中

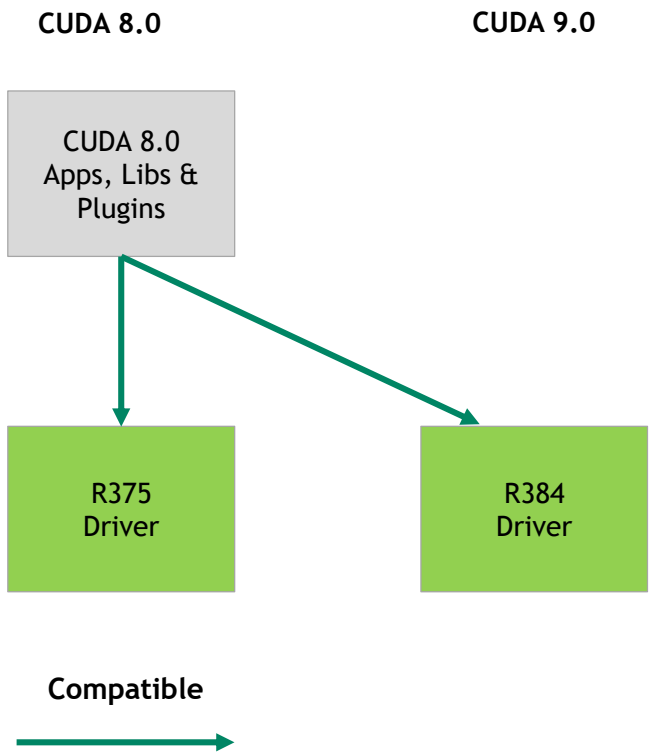


CUDA 兼容性

老版本的CUDA能够运行在新版本的驱动上

CUDA driver API 后向兼容但非前向兼容

- 每个 CUDA 版本都有最小驱动号要求
- 针对某个CUDA编译的应用闯红灯徐能够运行在新的驱动版本中
- 例如
  - CUDA 8.0 needs >= R375
  - CUDA 9.0 needs >= R384

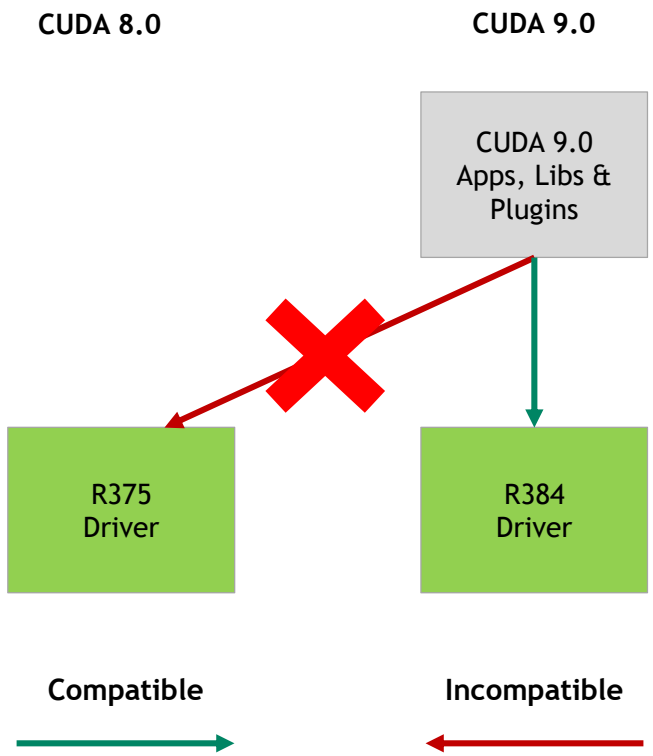


CUDA 兼容性

新版本的CUDA不能运行在老版本的驱动上

CUDA driver API 后向兼容但非前向兼容

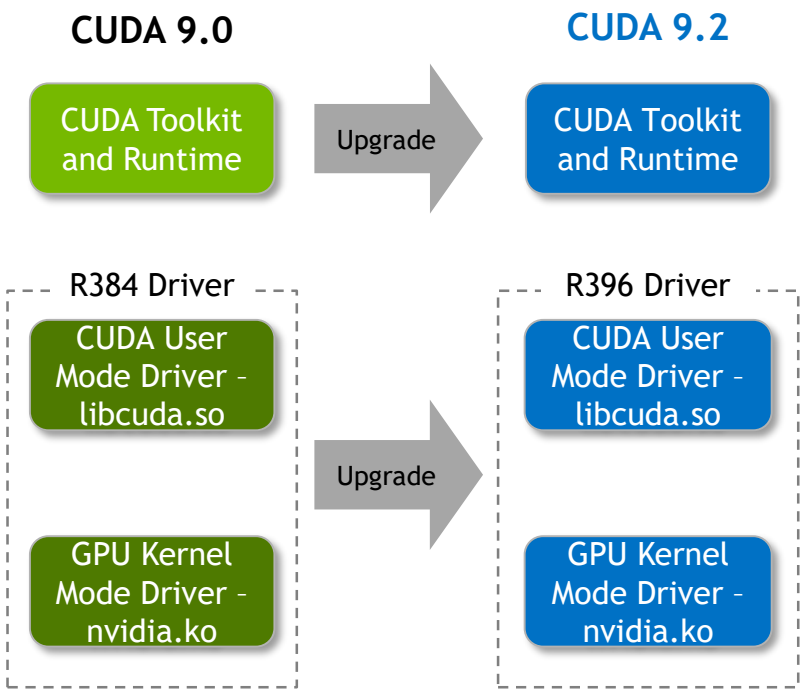
- 每个 CUDA 版本都有最小驱动号要求
- 针对某个CUDA编译的应用闯红灯徐能够运行在新的驱动版本中
- 例如
  - CUDA 8.0 needs >= R375
  - CUDA 9.0 needs >= R384



CUDA 兼容性 - 新的升级路径

后向兼容

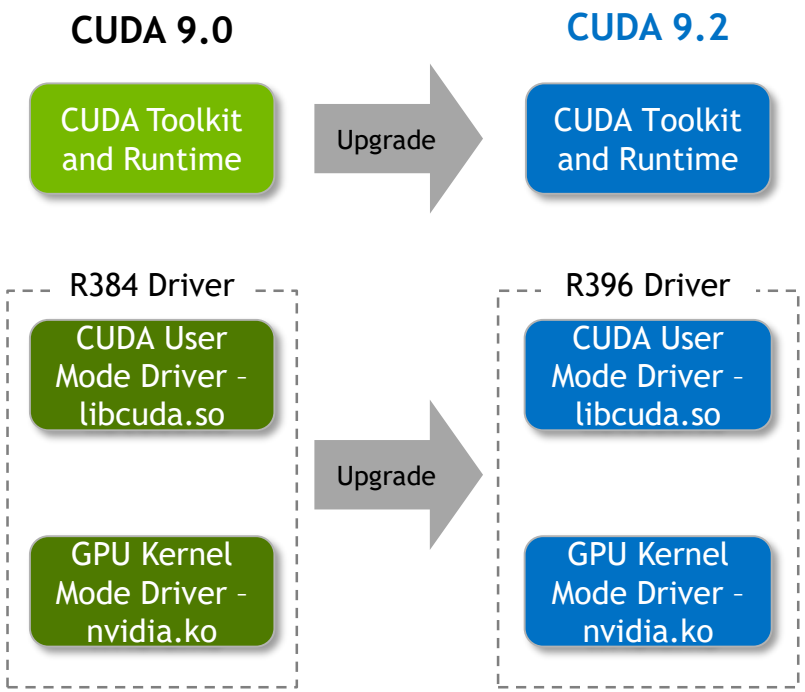
同时升级工具包和驱动



CUDA 兼容性 - 新的升级路径

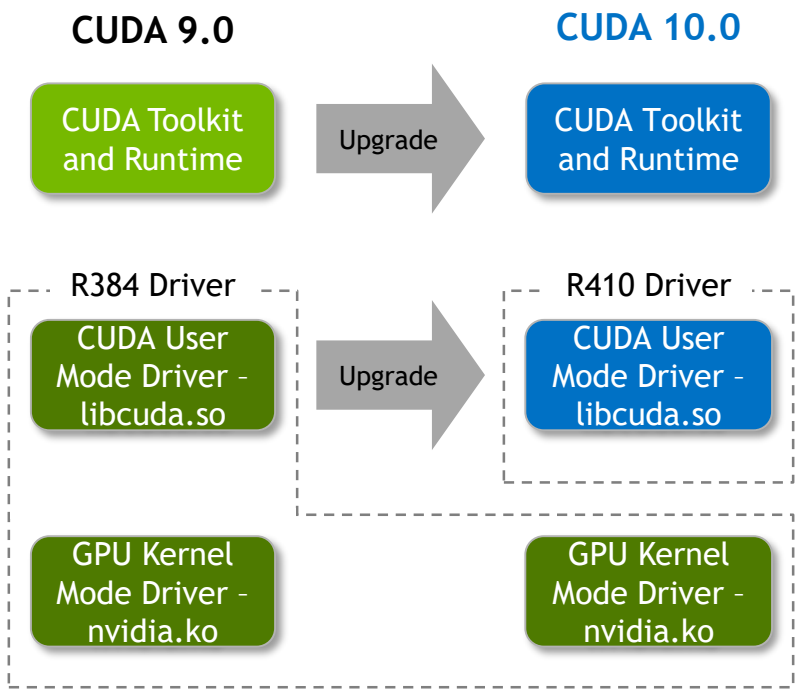
后向兼容

同时升级工具包和驱动



新的前向兼容选项

只升级用户模式的 CUDA 组件*



*需要新的‘cuda-compat-10-0’安装包

CUDA 兼容性 - 新的升级路径

从CUDA 10.0开始

新的兼容性平台下可用的升级路径

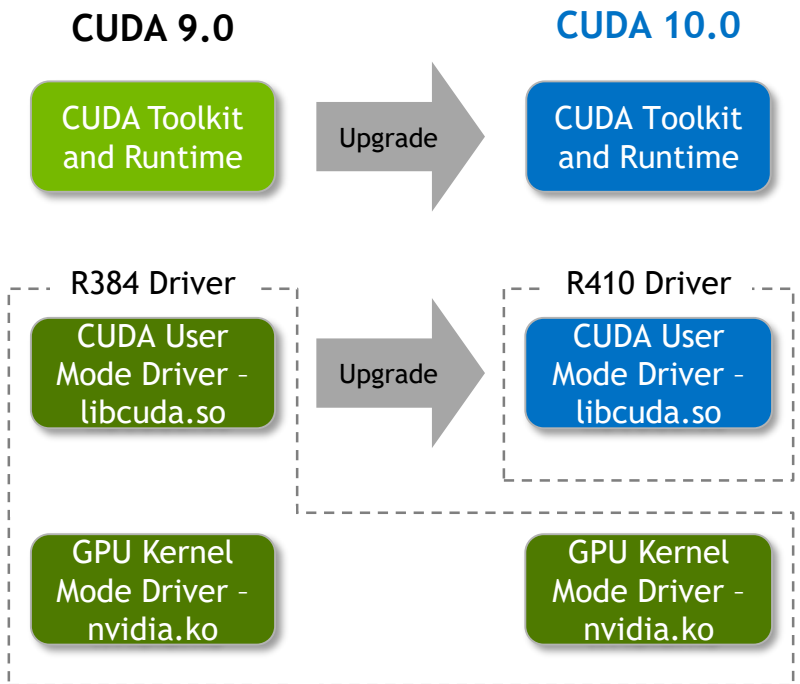
- 在老版本的驱动下使用新版本的CUDA工具包
- 兼容性仅仅针对特定的老版本驱动

系统要求

- 仅支持Tesla GPU – 不支持Quadro或GeForce
- 仅支持Linux

新的前向兼容选项

只升级用户模式的 CUDA 组件*

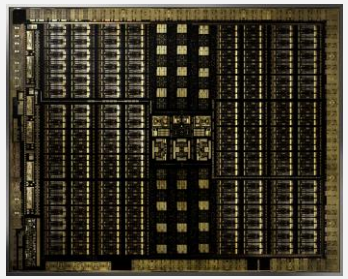


*需要新的‘cuda-compat-10-0’安装包

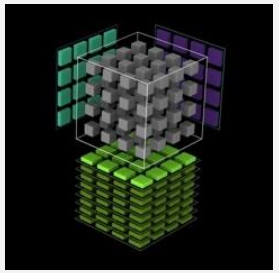

CUDA TURING

CUDA 10 - TURING

Turing 架构

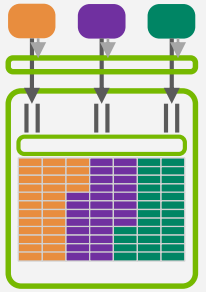


多种精度的
Tensor Core

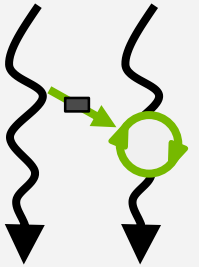


RT Core

Turing MPS



独立线程调度



推理加速，图形改造，Volta's 可编程性

TESLA T4

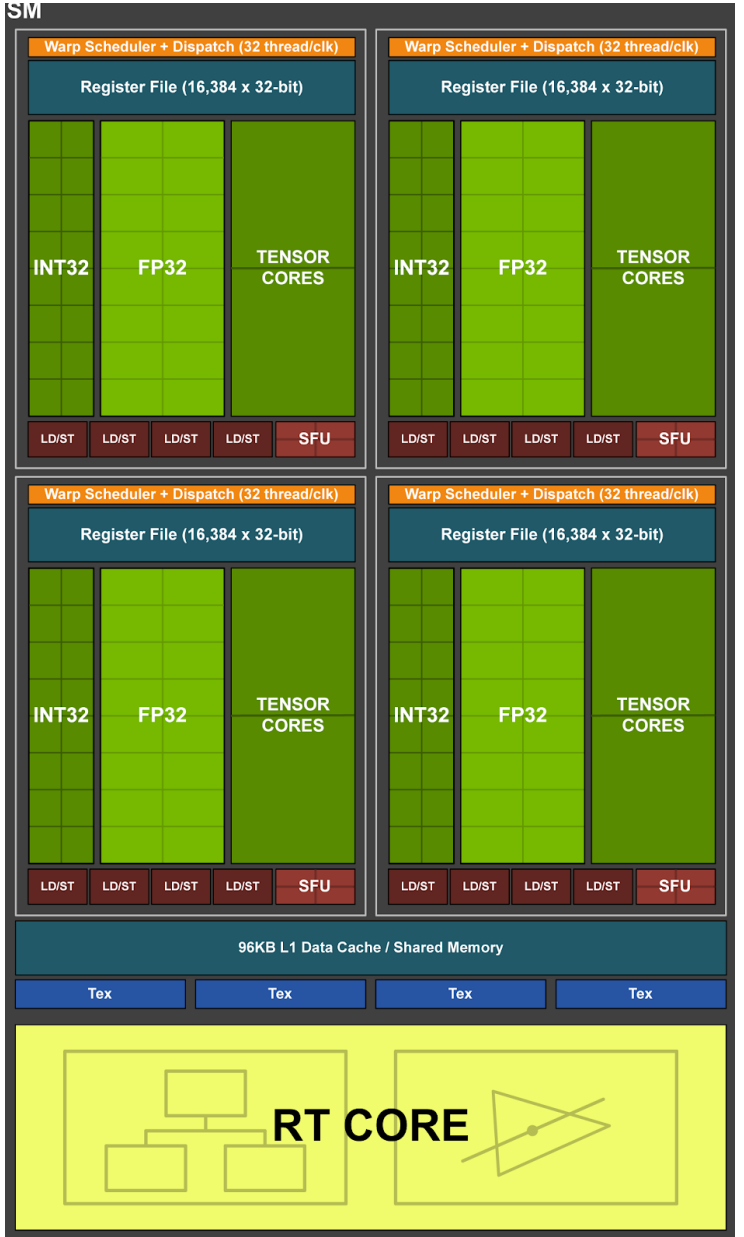
世界上最先进的推理GPU

通用推理加速
320 Turing Tensor cores
2,560 CUDA cores
65 FP16 TFLOPS | 130 INT8 TOPS | 260 INT4 TOPS
16GB | 320GB/s

TURING SM

	TU102
INT32	64
FP32	64
Tensor Cores	8
RT Core	1
Register File	256 KB
L1 and shmem	96 KB
Max threads	1024
Compute Capability	75*

*Volta (cc70) 代码在Turing处理器运行不需要运行时编译或重编译!

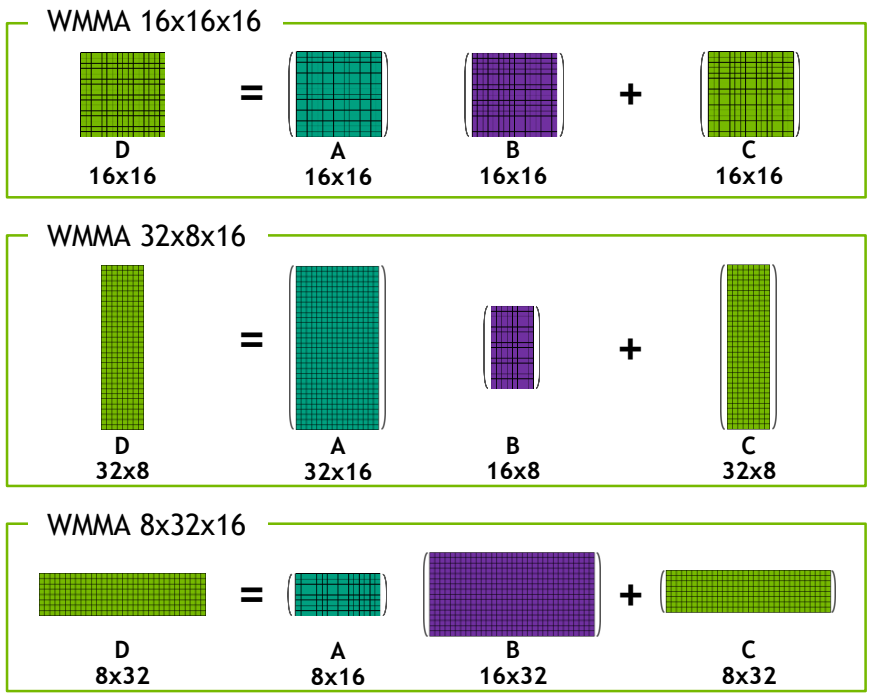


TURING TENSOR CORE

新的WARP矩阵运算功能

WMMA 操作可支持 8-bit 整型

- 仅限于Turing 架构(sm_75)
- 有符号和无符号 8-bit 整型输入
- 32-bit 整型累加器
- 与半精度的输入输出维度相匹配
- 每个SM每个周期可执行2048次操作



试验性的 WARP 矩阵运算功能

Turing 支持试验性的半字节 Tensor Core 操作

试验性的半字节操作

- 4-bit 有符号和无符号输入
- 1-bit 输入自定义矩阵运算
- 32-bit 累加器输出

通过特性的命令空间访问：

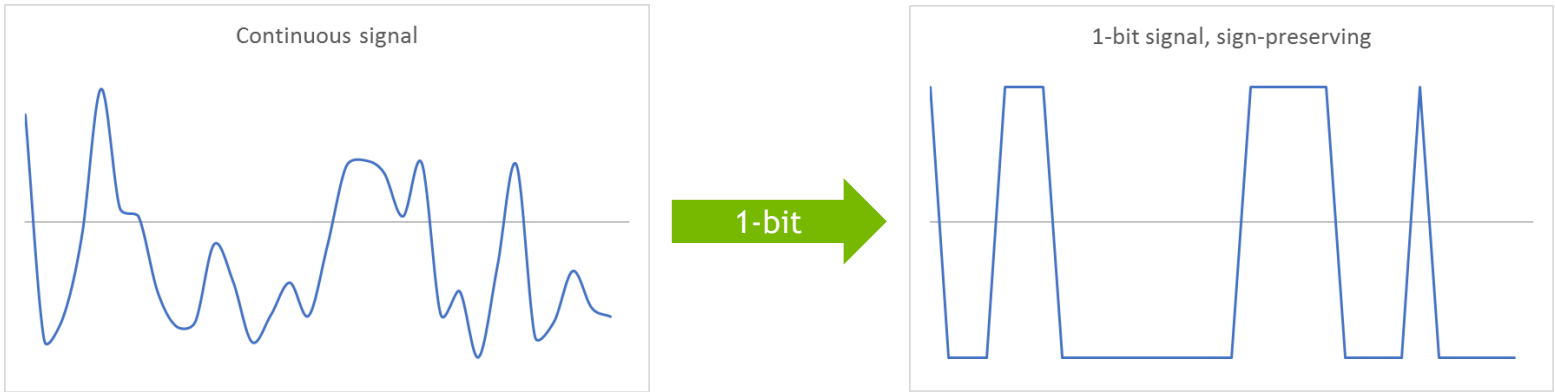
nvcuda::wmma::experimental

```
namespace experimental {
    namespace precision {
        struct u4; // 4-bit unsigned
        struct s4; // 4-bit signed
        struct b1; // 1-bit
     }
    enum bmmaBitOp { bmmaBitOpXOR = 1 };
    enum bmmaAccumulateOp { bmmaAccumulateOpPOPC = 1 };
}
```

使得研究人员能够进行超低精度的试验!

二进制 TENSOR CORES

例子：二值化神经网络



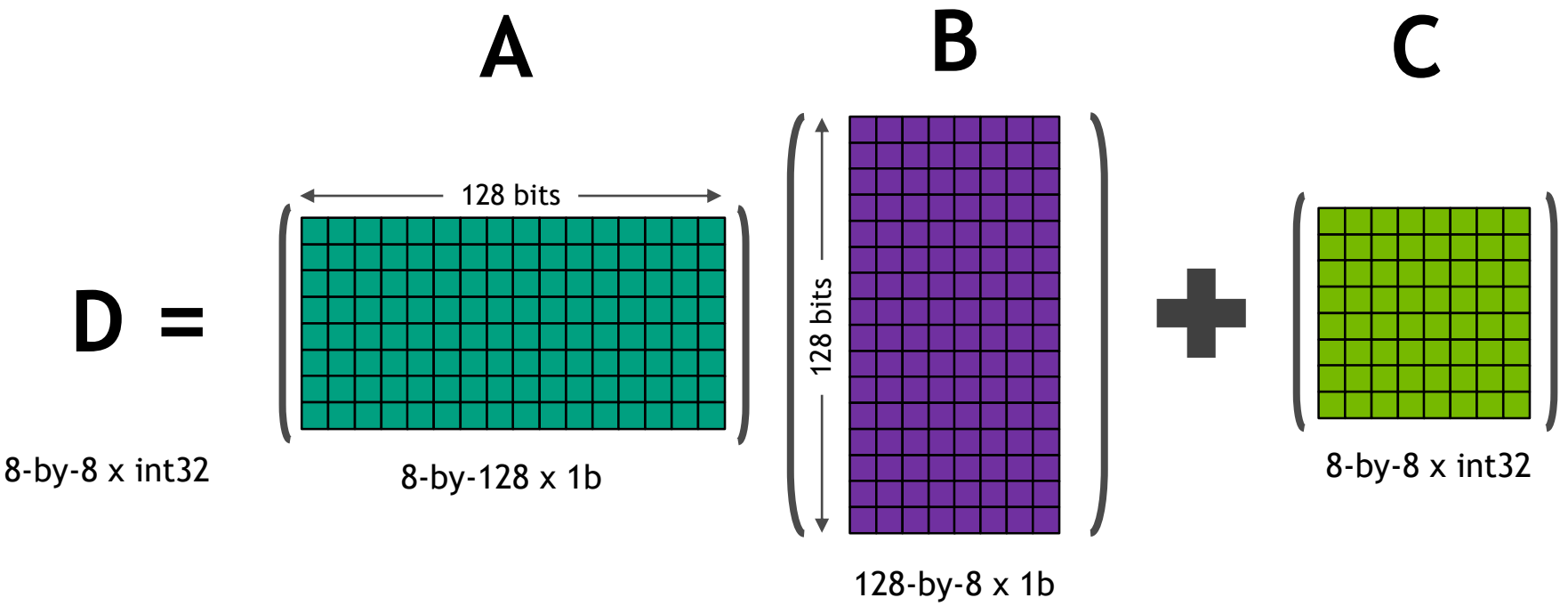
基本概念

- 用更低的精度训练神经网络: 更快的计算速度，更少的存储资源
- 将数据减少为只有正/负符号值- 可由单个bit表示(1 = +ve, 0 = -ve)
- 近对数据符号做1-bit 权值和激活运算

Ref: *Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1*, M. Coubariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y Bengio, 2016
https://arxiv.org/pdf/1602.02830.pdf

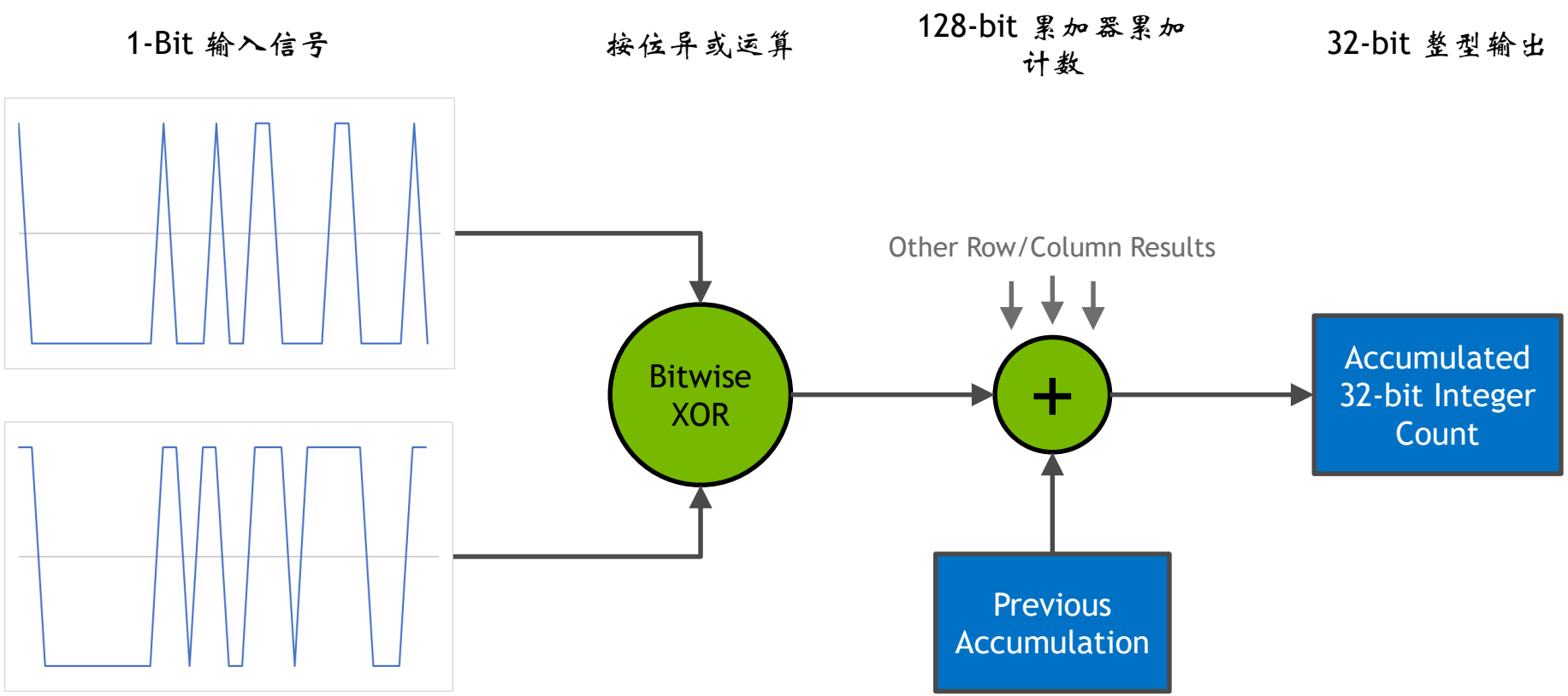
WMMA - 二进制 - 异或 POPC

Turing新特性（试验性的）



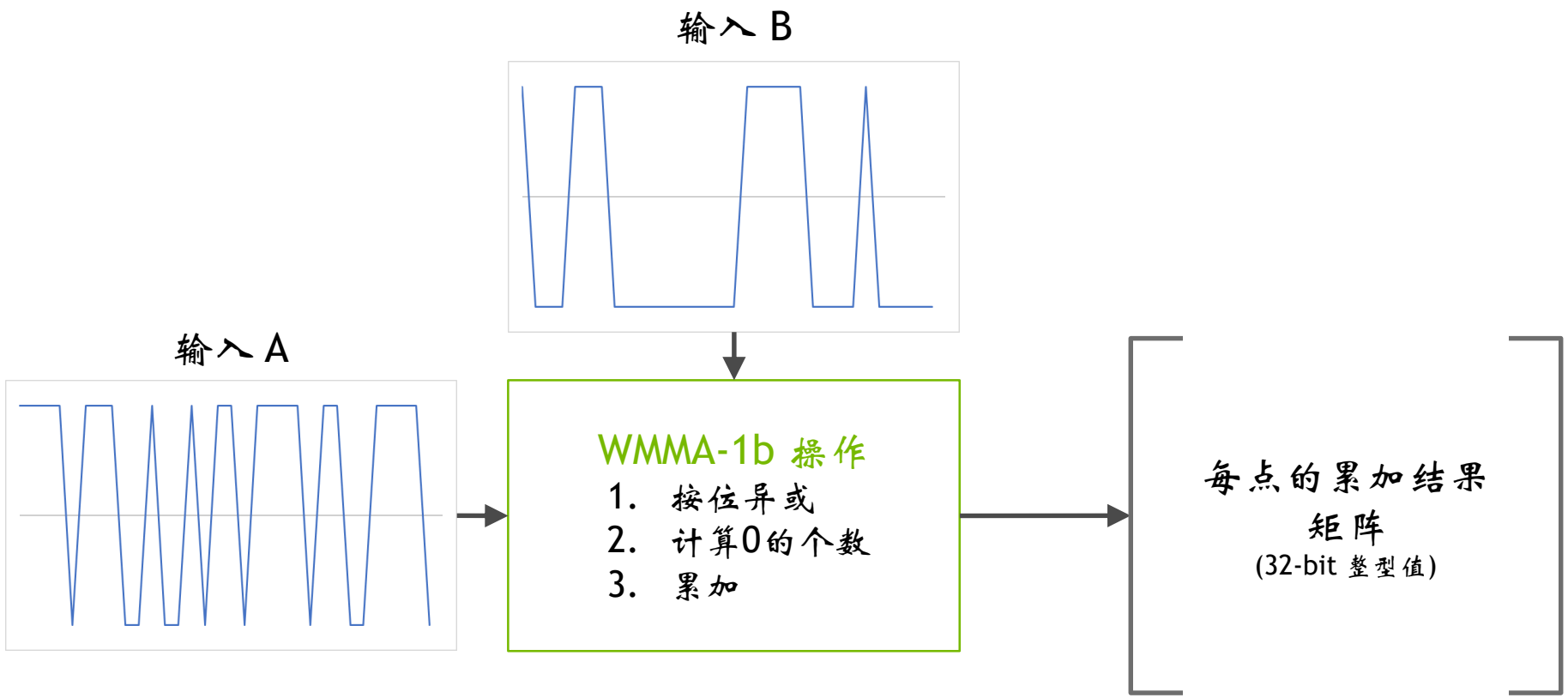
$$D_{i,j} = popc(A_{i,k} \wedge B_{k,j}) + C_{i,j} \qquad \text{for } k = 0 .. 127$$

二进制 TENSOR CORE 操作



二进制 TENSOR CORE 操作

矩阵-矩阵的符号/幅度相关运算



新的 TURING WARP 矩阵运算函数

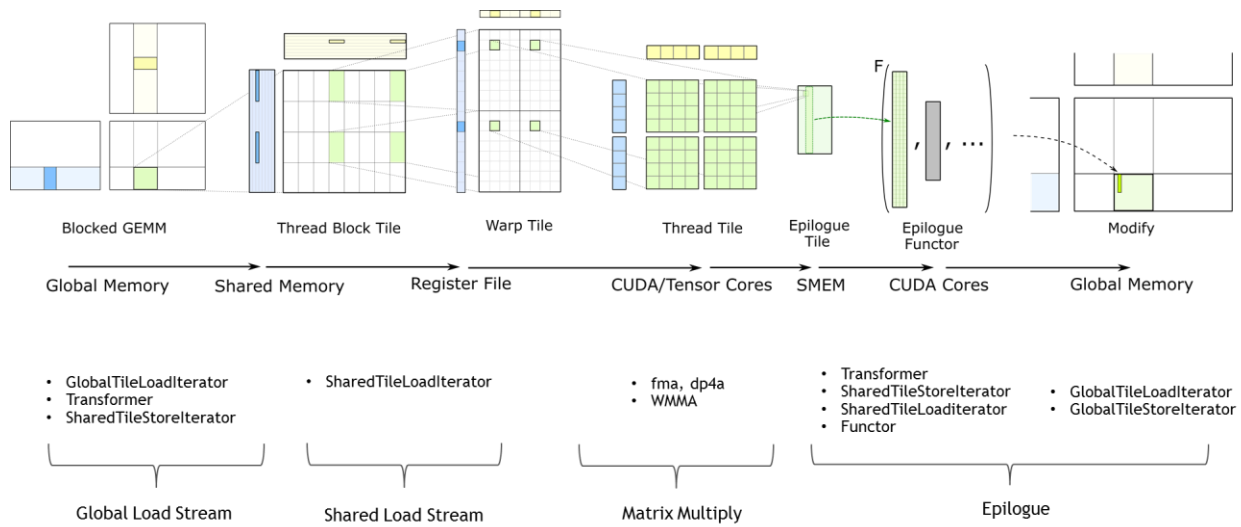
	输入精度	输入精度	支持大小	每个SM每个时钟周期最大操作数
Native Types	half *	half or float	16 x 16 x 16 32 x 8 x 16 8 x 32 x 16	1024
Native Types	char	integer (int32)		2048
Native Types	unsigned char	integer (int32)		2048
Experimental	precision::u4 (4-bit unsigned)	integer (int32)	8 x 8 x 32	4096
Experimental	precision::s4 (4-bit signed)	integer (int32)	8 x 8 x 32	4096
Experimental	precision::b1 (1-bit)	integer (int32)	8 x 8 x 128	16384

* 表示也可以用于Volta sm_70. 注意：为获取峰值性能， WMMA 要求基于Turing sm_75重新编译

CUTLASS 1.2

基于开源CUDA C++的高性能矩阵乘法加速库

- 基于Turing 架构优化的 GEMMs
  - 整型 (8-bit, 4-bit and 1-bit) WMMA 运算
- 批量跨步 GEMM / WMMA GEMM
- 并行规约
- 支持 CUDA 10.0
- 更新了文档和更多例程



CUTLASS GEMM 结构模型

https://github.com/NVIDIA/cutlass

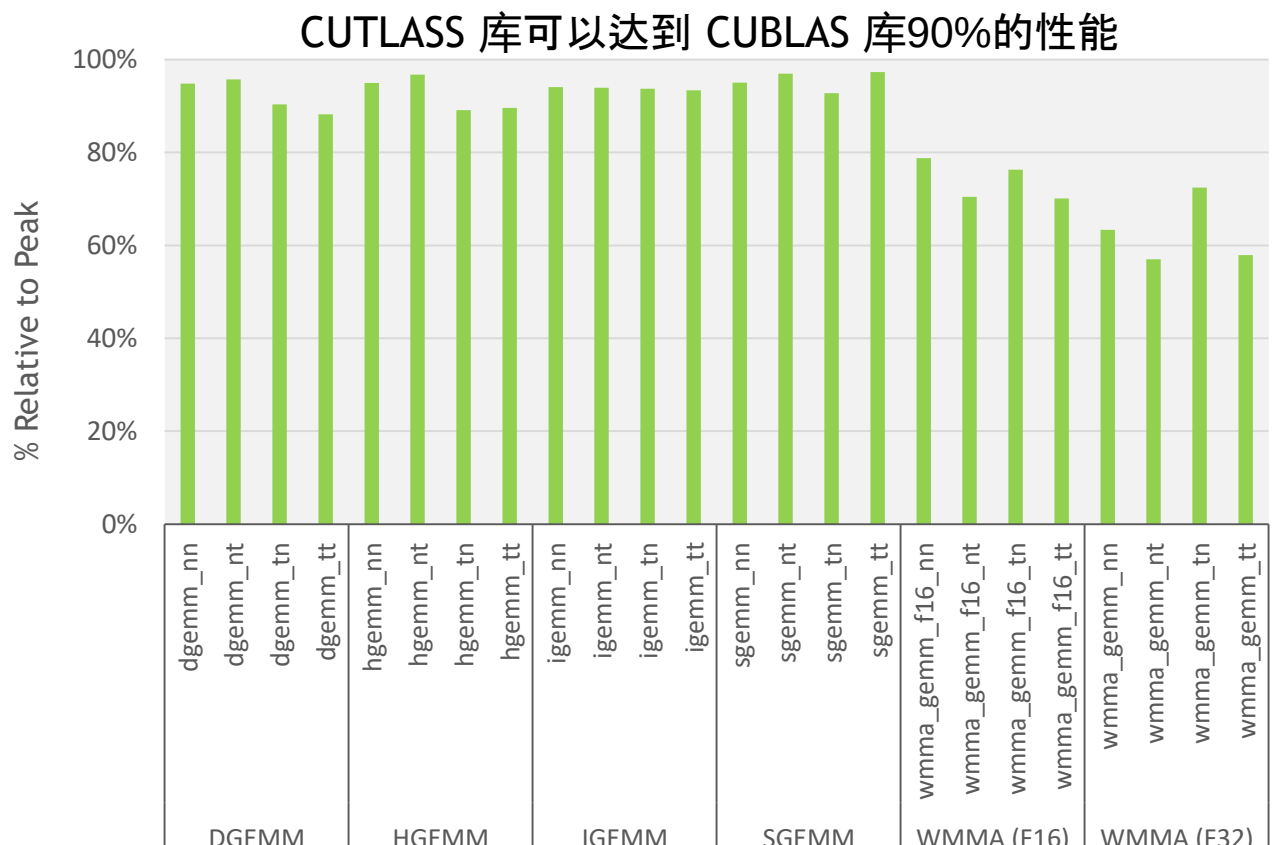
CUTLASS 1.2

基于开源CUDA C++的高性能矩阵乘法加速库

- 基于Turing 架构优化的 GEMMs
  - 整型 (8-bit, 4-bit and 1-bit) WMMA 运算
- 批量跨步 GEMM / WMMA GEMM
- 并行规约
- 支持 CUDA 10.0
- 更新了文档和更多例程

详细介绍，敬请关注“”

https://github.com/NVIDIA/cutlass



CUTLASS 1.1 on Volta (GV100)

TURING 多进程服务 (MPS)

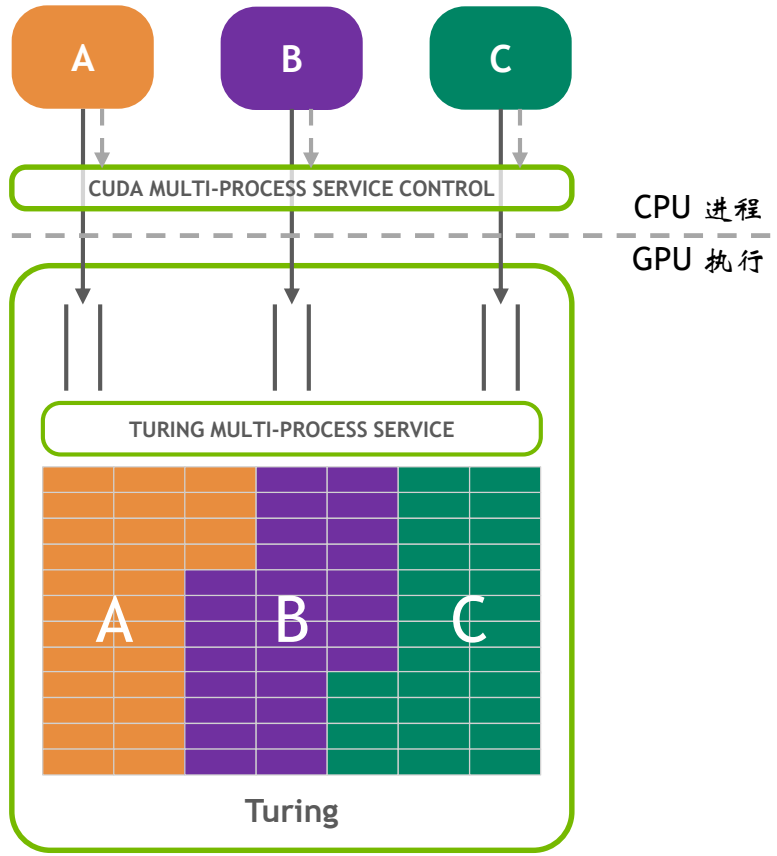
改进了小batch情况下推理计算的吞吐性能

Turing MPS:
继承了 Volta's 增强 MPS 架构

- 降低加载延迟
- 改进加载吞吐
- 利用调度程序分割机制改进服务质量
- 是Pascal架构下可容纳客户数量的三倍

硬件加速
任务提交

硬件
隔离



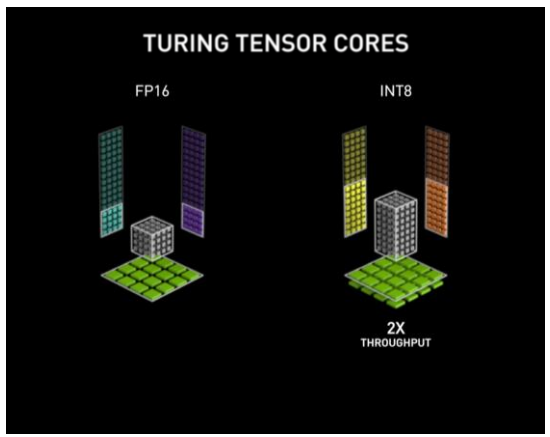
TENSORRT 5 支持 TURING GPUS

使用混合精度(FP32, FP16, INT8) 和Turing Tensor Cores实现最快推理计算

加速产品中的推荐、语音、视频、翻译等的推理计算

- 基于Turing GPUs优化的混合精度(FP32, FP16, INT8) kernels函数
- 相比于CPU平台，可实现40倍以上的推理计算加速比
- 当执行多个独立的推理计算进程时，MPS 能够最大化GPU利用率并实现更高的吞吐率

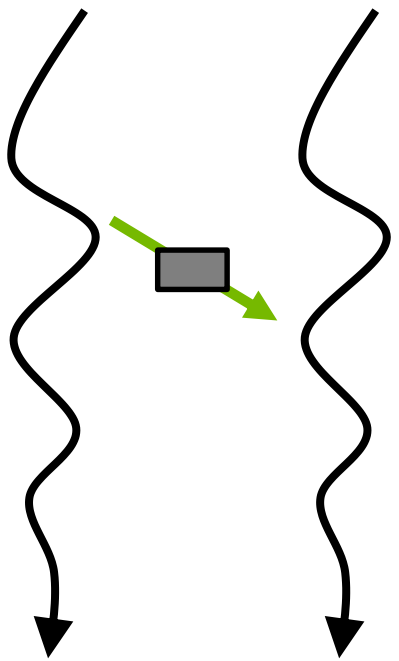
具体内容，敬请关注“”



developer.nvidia.com/tensorrt

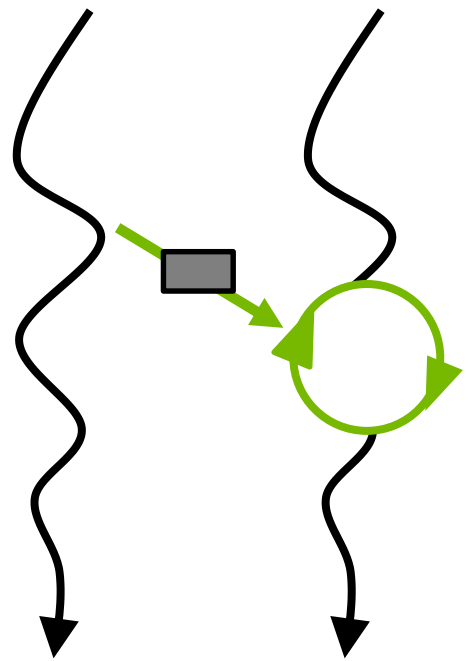
独立线程调度

通信算法



Pascal: Lock-Free 算法

线程不能等待消息



Volta/Turing: Starvation Free 算法

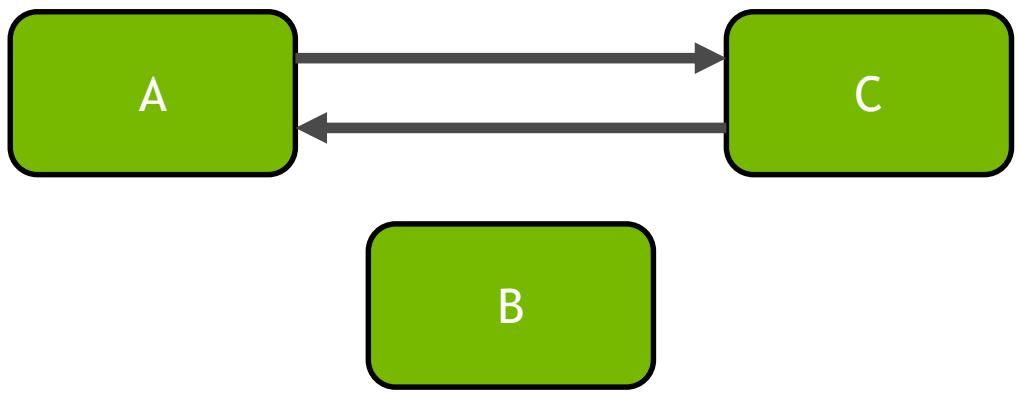
线程能够等待消息

STARVATION FREE 算法

例子

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

具有细粒度锁的双向链表

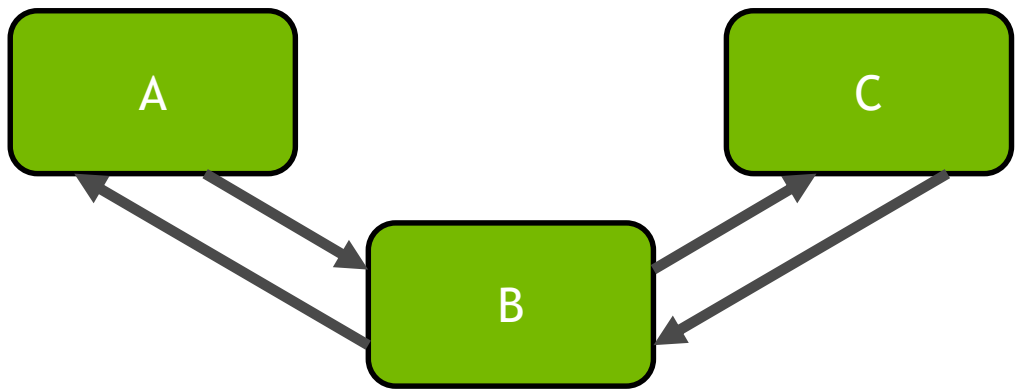


STARVATION FREE 算法

例子

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

具有细粒度锁的双向链表

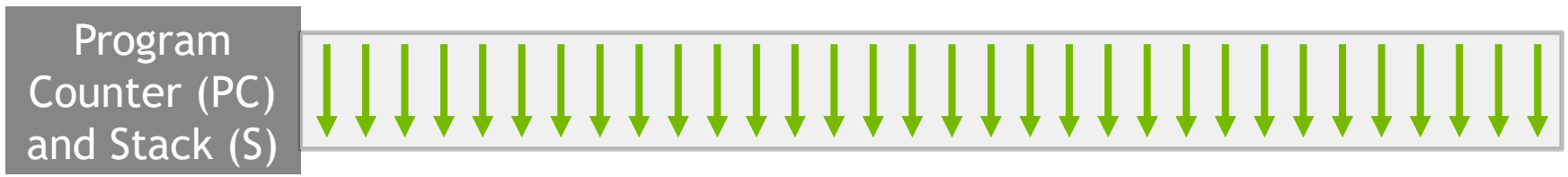


提示! GPU 能够同时运行 >100k 线程

最小化竞争!

WARP 实现

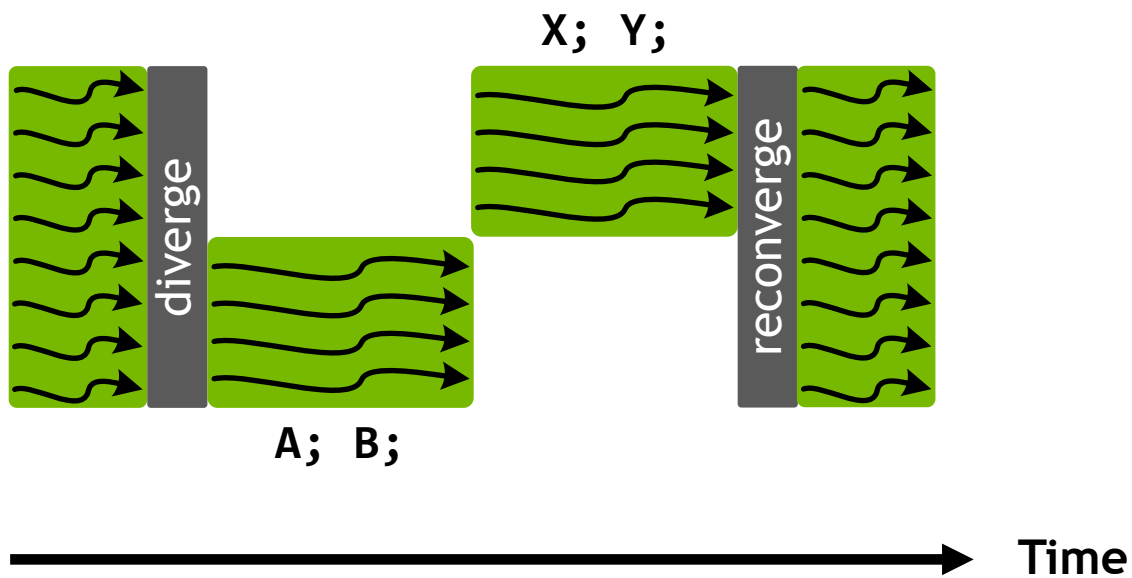
Pre-Volta



Warp内的32个线程必须统一调度

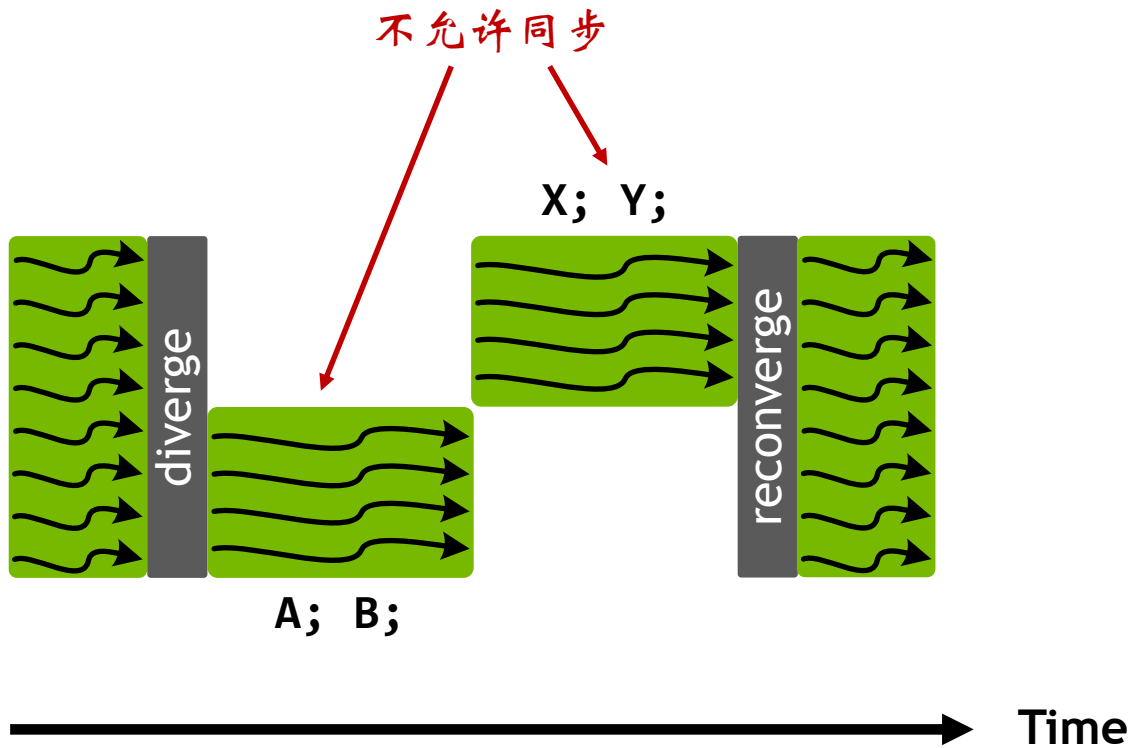
PASCAL WARP 执行模型

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
```



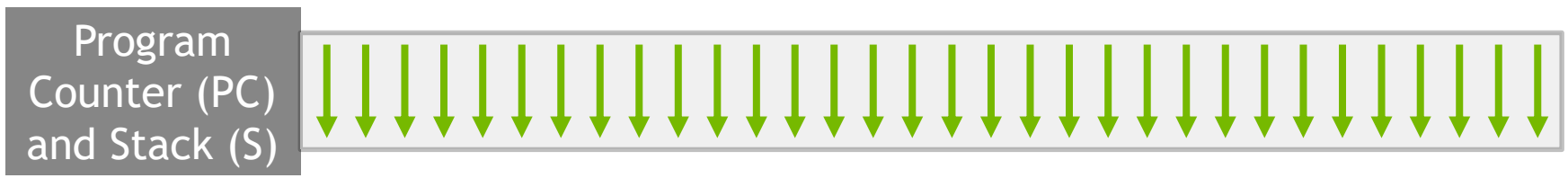
PASCAL WARP 执行模型

```
if (threadIdx.x < 4) {
    A;
    __syncwarp();
    B;
} else {
    X;
    __syncwarp();
    Y;
}
```



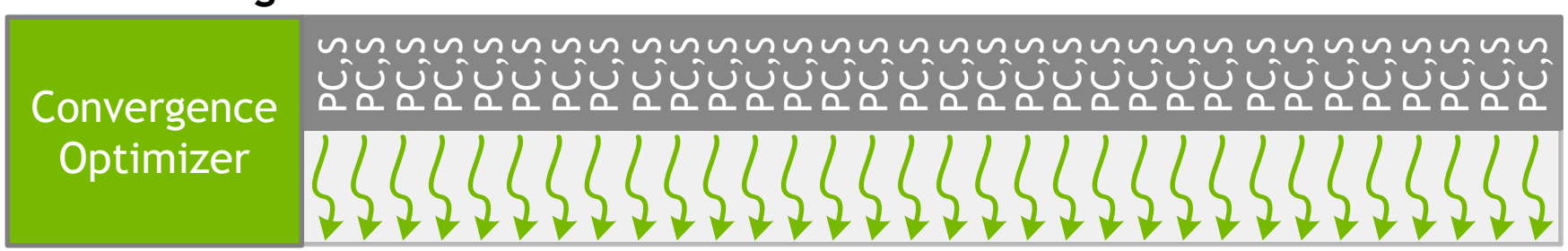
WARP 实现

Pre-Volta



Warp内的32个线程必须统一调度

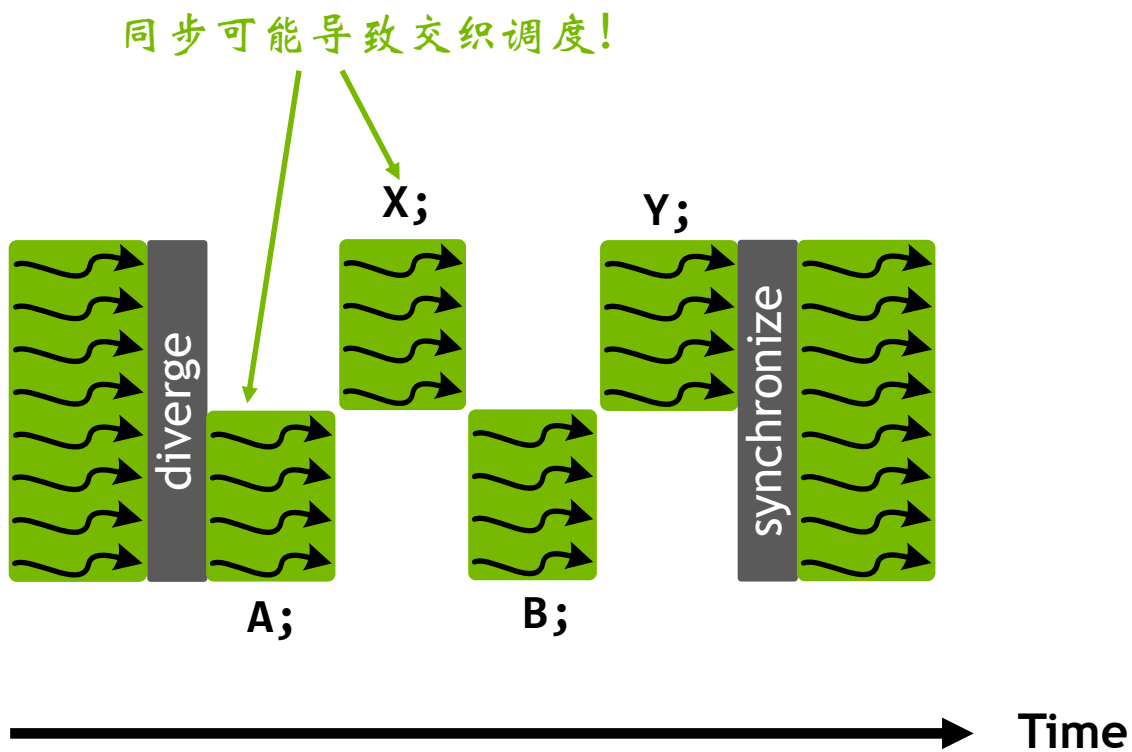
Volta/Turing



Warp内的32个线程可以独立的调度

VOLTA/TURING WARP 执行模型

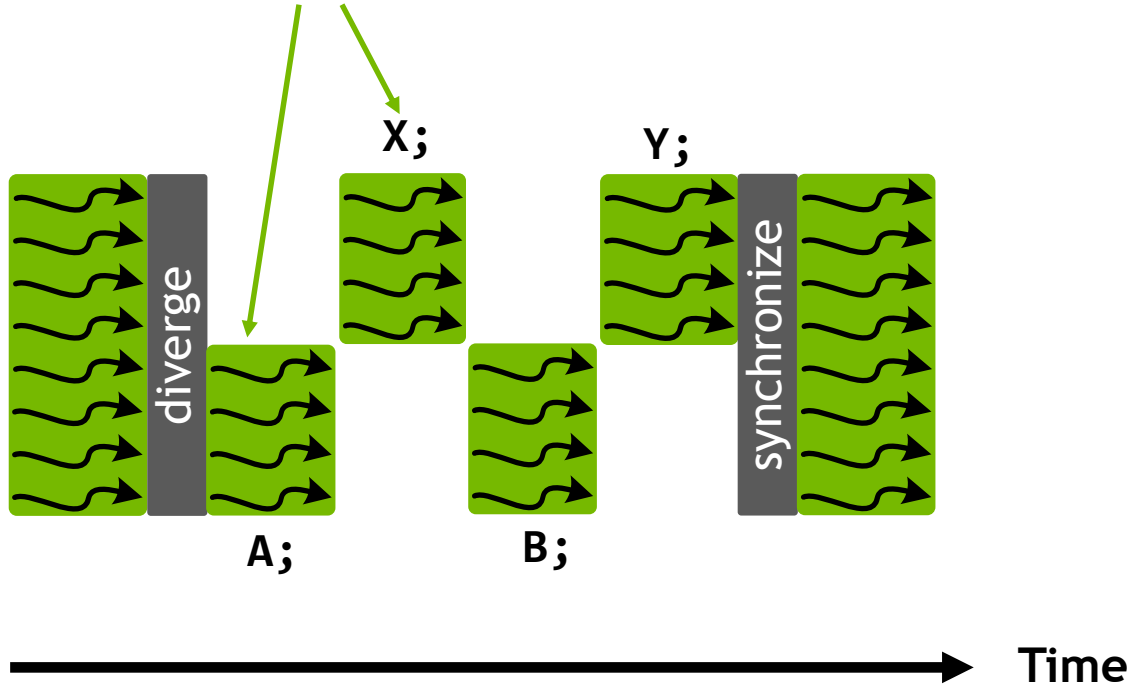
```
if (threadIdx.x < 4) {
    A;
    __syncwarp();
    B;
} else {
    X;
    __syncwarp();
    Y;
}
__syncwarp();
```



VOLTA/TURING WARP 执行模型

同步可能导致交织调度!

```
if (threadIdx.x < 4) {
    A;
    __syncwarp();
    B;
} else {
    X;
    __syncwarp();
    Y;
}
__syncwarp();
```



同样也支持软件同步，例如：双向链表锁!

协作组

灵活和可扩展的线程同步和通信

对协作组中的线程进行定义、同步和分组

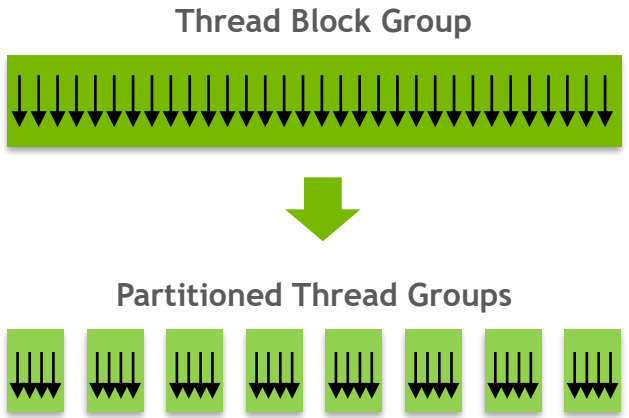
跨软件边界清理组件

优化硬件快速路径

从几个线程到所有线程的可扩展性

部署在多种GPU平台: Kepler 架构以上的 GPUs

CUDA开发工具支持



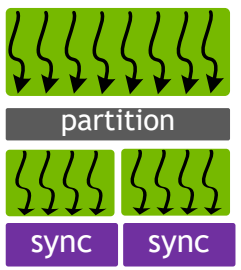
* 注意：多线程块和多GPU协作组仅支持Pascal及以上的GPUs

可在任意规模下实现同步

三大关键能力

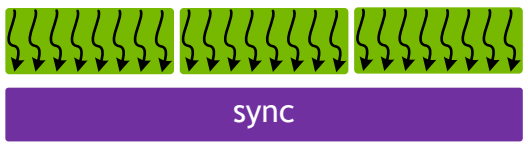
灵活的线程组

定义和同步任意线程组

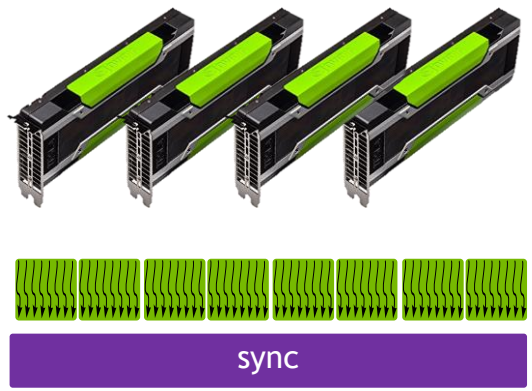


整个线程网格同步

同步多个线程块



多GPU同步

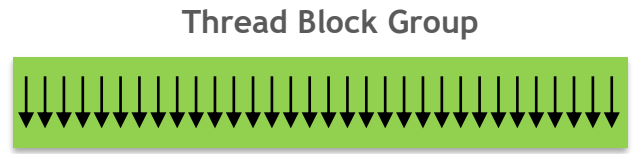


协作组基本概念

灵活、显示的同步

协作组在程序中显示的定义

```
thread_group block = this_thread_block();
```



可以同步一个协作组中的所有线程

```
block.sync();
```

创建新的组，来对已定义的组进行划分

```
thread_group tile32 = tiled_partition(block, 32);
thread_group tile4 = tiled_partition(tile32, 4);
```

划分后的小组仍可进行同步

```
tile4.sync();
```

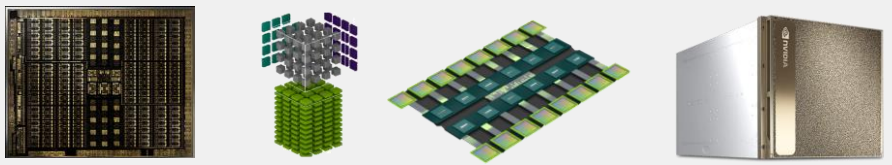
注意：绿色部分代码调用属于cooperative_groups::命名空间

CUDA 10.0 – 现在已可以在NVIDIA官网下载!

https://developer.nvidia.com/cuda-toolkit

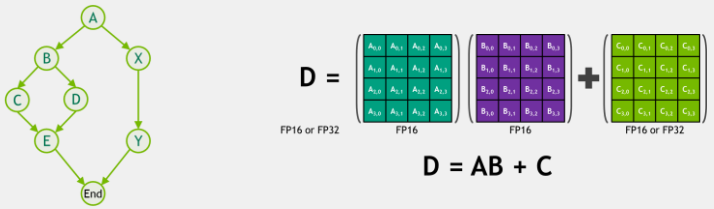
支持TURING 和最新系统

新的 GPU 架构, Tensor Cores, NVSwitch 结构



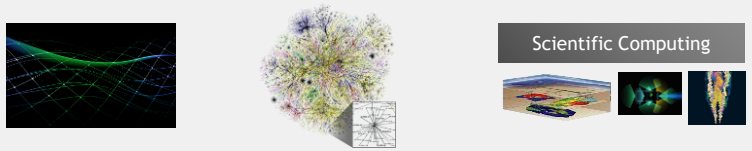
CUDA 平台

CUDA Graphs, Vulkan & DX12 中间件, Warp 矩阵计算模型



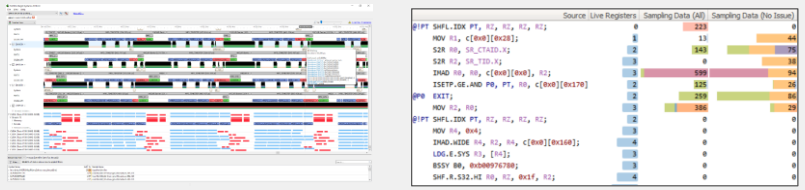
加速库

基于GPU的混合JPEG解码, 对称特征求解器, FFT 扩展计算



开发者工具

新的 Nsight 家族产品 – Nsight Systems 和 Nsight Compute




CUDNN 7.3

cuDNN 7.3

深度神经网络GPU加速库

- 支持最新 Turing GPUs
- 支持空洞卷积
- 针对几种格式的分组卷积进行了优化：HHH / HSH / SSS
- 更加方便的使用Tensor Cores: 输入通道数和输出通道数可以被自动补齐到8的倍数
- 对BN层的浮点数溢出处理做了更正

cuDNN加速深度学习框架

https://developer.nvidia.com/cudnn

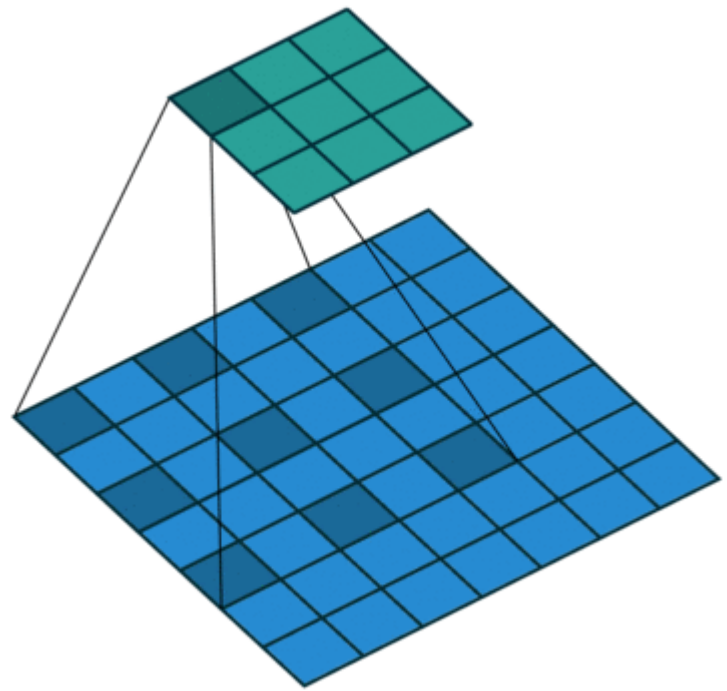
CUDNN 7.3

空洞卷积 (Dilated Convolution)

cudnnConvolutionForward() for 2D:
CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM

cudnnConvolutionBackwardData() for 2D:
CUDNN_CONVOLUTION_BWD_DATA_ALGO_1

cudnnConvolutionBackwardFilter() for 2D:
CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1



CUDNN 7.3

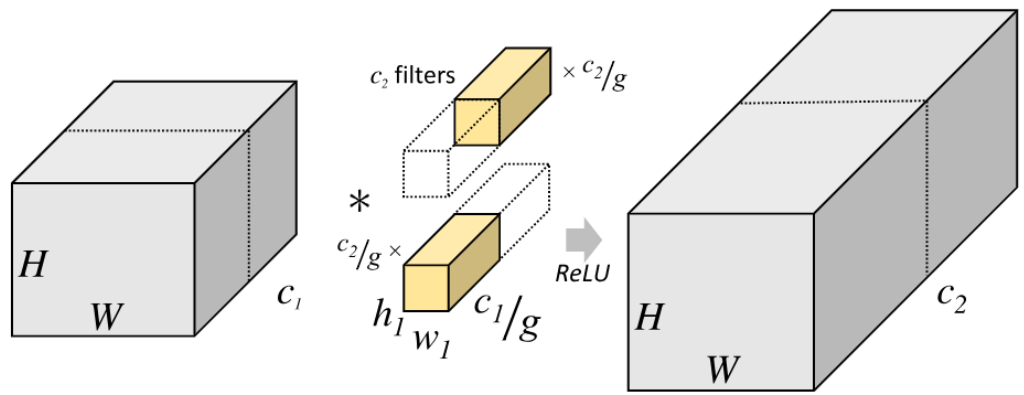
分组卷积 (Grouped Convolution)

对如下NHCW格式的分组卷积进行了优化:

HHH (输入: 半精度, 计算: 半精度, 输入: 半精度)

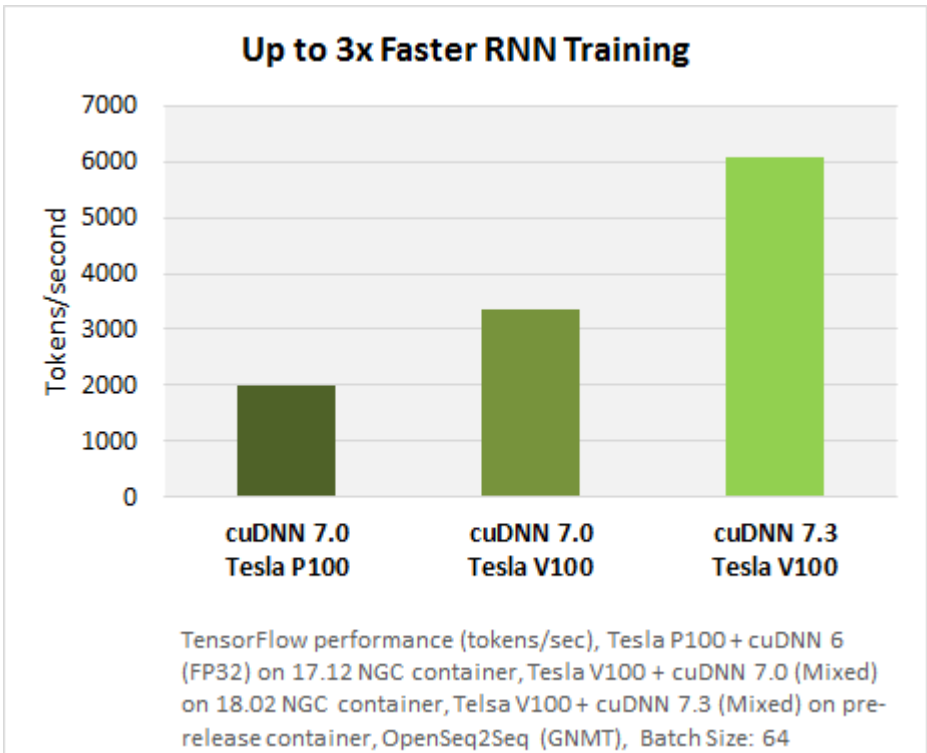
HSH (输入: 半精度, 计算: 单精度, 输入: 半精度)

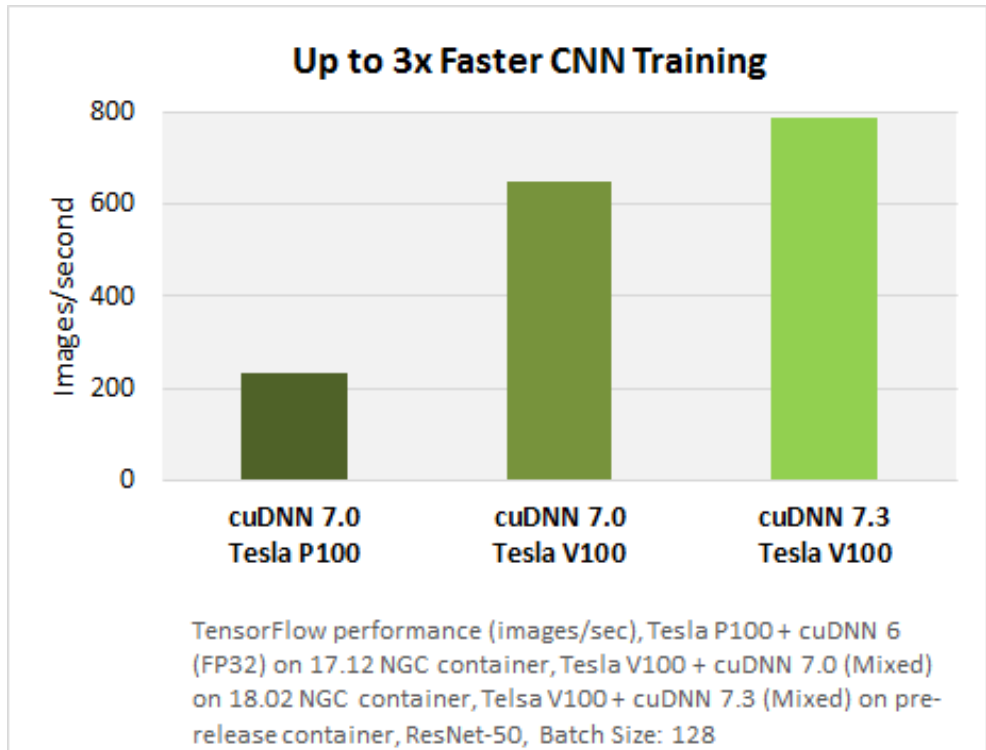
SSS (输入: 单精度, 计算: 单精度, 输入: 单精度)



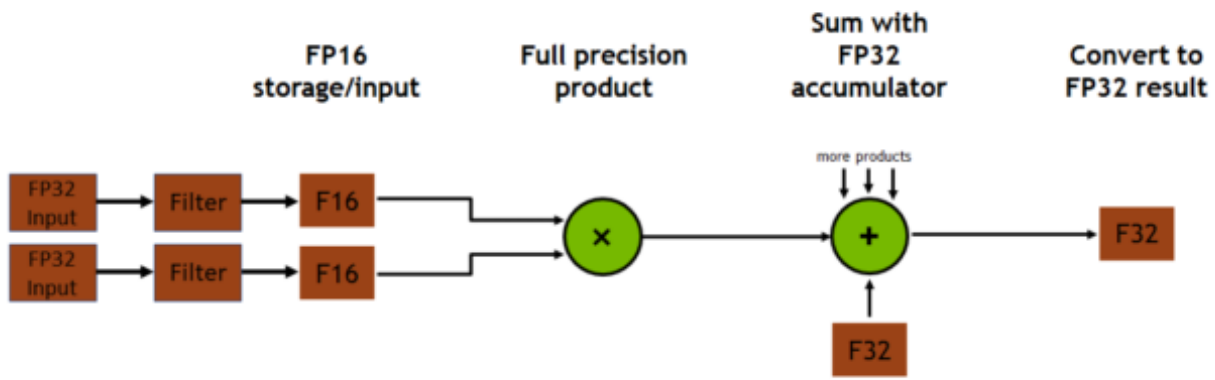
CUDNN 7.3

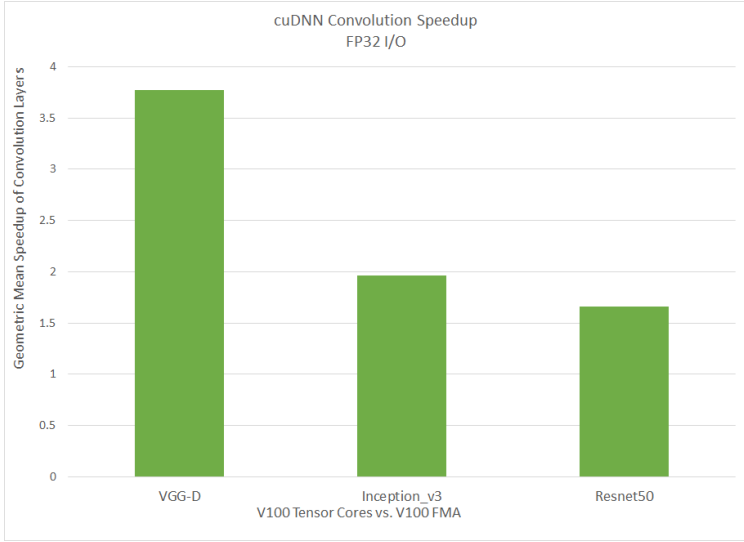
训练性能提升





使用FP32 输入数据类型来调用TENSOR CORES



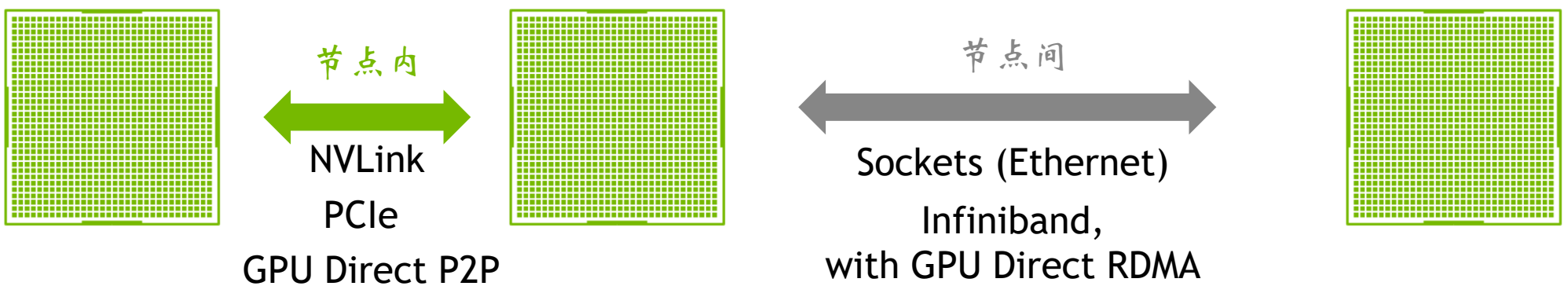



```
// Set the math type to allow cuDNN to use Tensor Cores:
checkCudnnErr( cudnnSetConvolutionMathType(cudnnConvDesc,
CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION) );

checkCudnnErr( cudnnSetRNNMatrixMathType(cudnnRnnDesc,
CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION) );
```


NCCL 2.3

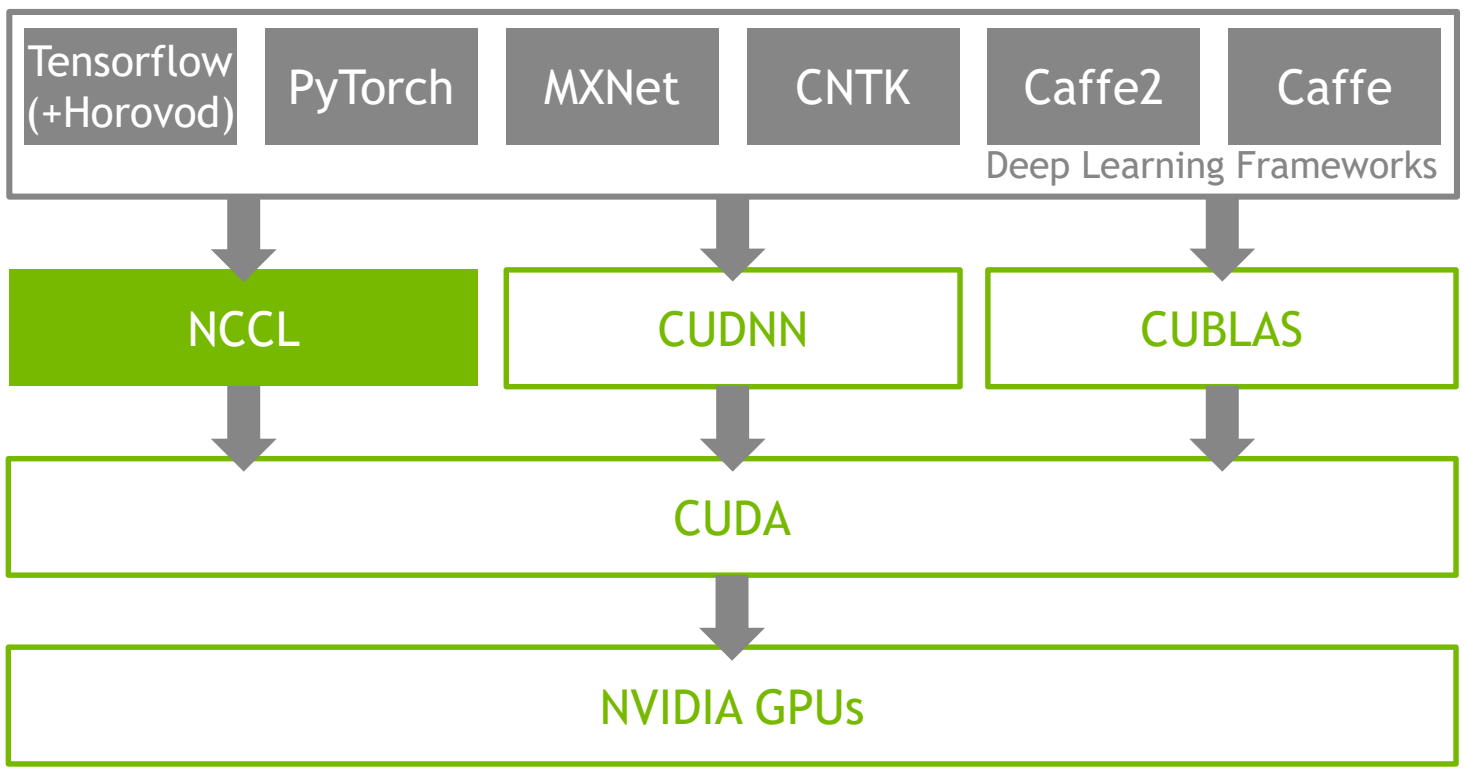
NCCL

多GPU集合通信库



NCCL

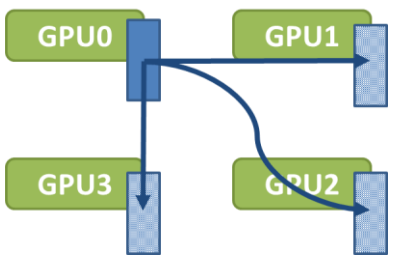
软件架构



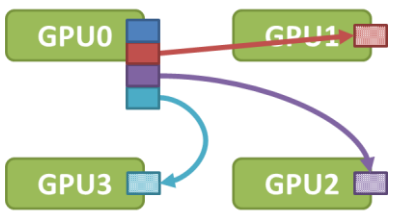
集合通信类型

多个发送方 / 接收方

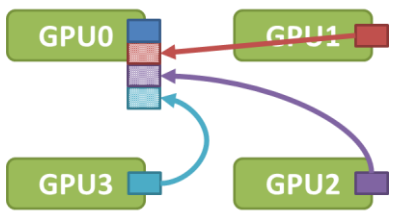
Broadcast



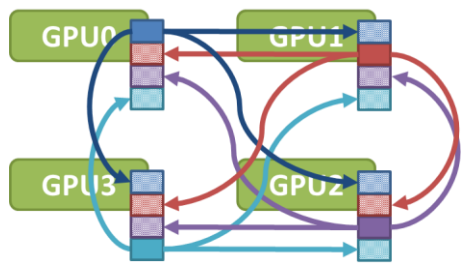
Scatter



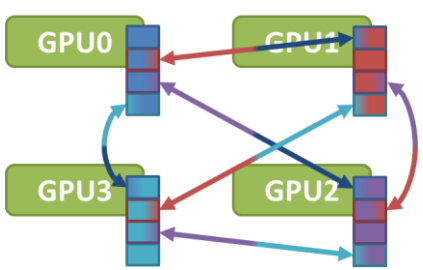
Gather



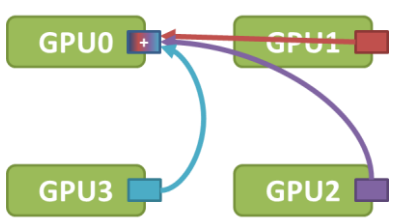
All-Gather



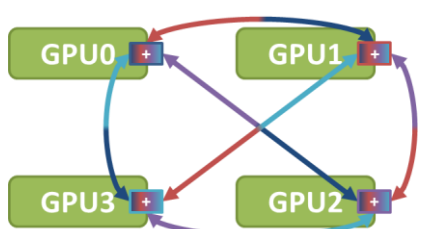
All-to-All



Reduce



All-Reduce

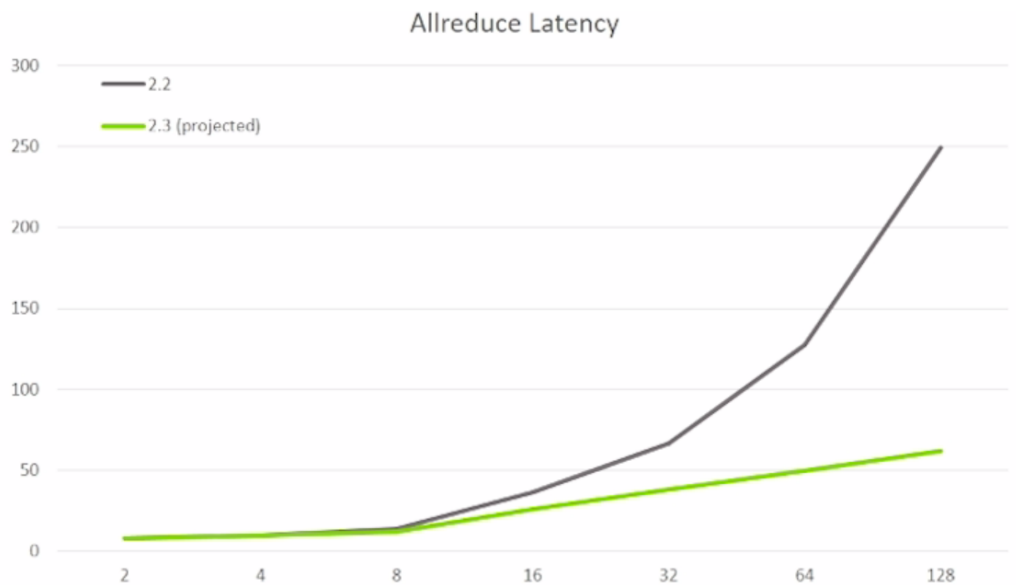


NCCL 2.3
最新特性

- 开源：https://github.com/NVIDIA/nccl
- 支持最新 Turing GPUs
- 改进小数据量消息传输的延迟性能
- 更好的控制使用GPU Direct P2P 和 GPU Direct RDMA
- 最大可支持16个rings
- 新增 ncclGetVersion() 函数

https://developer.nvidia.com/nccl

大规模GPUs通信算法的性能提升



NCCL 聚集操作

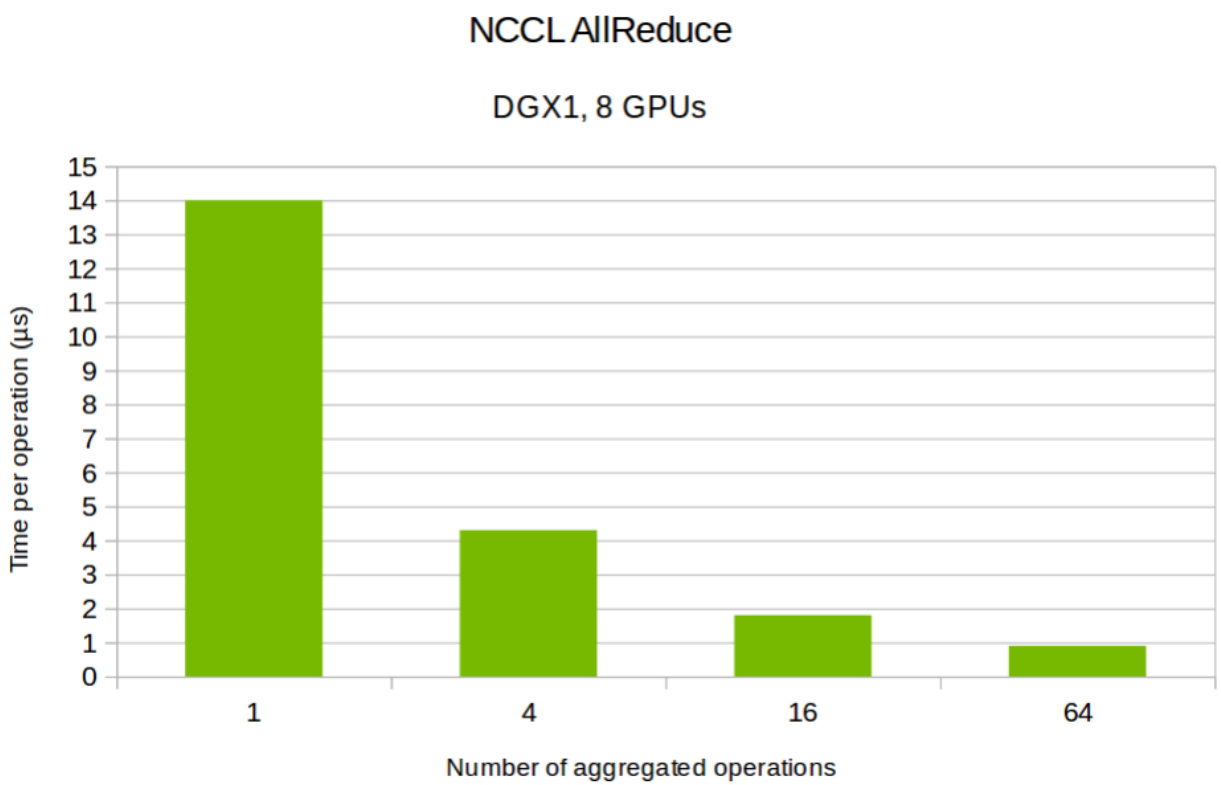
应用场景

在需要进行聚集的NCCL操作外围使用 ncclGroupStart() / ncclGroupEnd() 函数:

```
ncclGroupStart();
for (int op=0; op<nops; op++) {
  ncclAllReduce(
    layers[op].localGradients,
    layers[op].globalGradients,
    layers[op].gradientSize,
    ncclFloat, ncclSum, ncclComm, ncclStream);
}
ncclGroupEnd();
// All operations are only guaranteed to be posted on the stream after ncclGroupEnd
cudaStreamSynchronize(ncclStream);
```

NCCL

聚集操作的性能提升



https://devblogs.nvidia.com/scaling-deep-learning-training-nccl/


NVIDIA