持久性CUDA GPU 编程及其应用

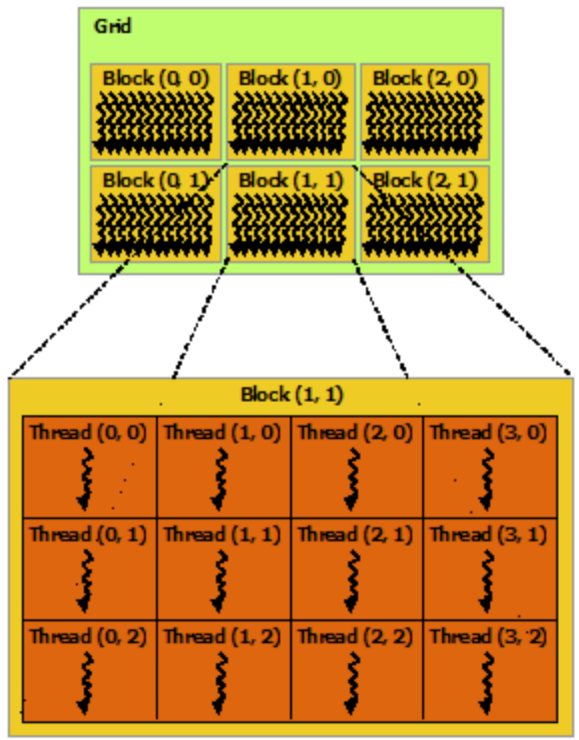
Fan Yu (郁凡), 12/19/2019

AGENDA

- 常规CUDA编程模型
  - CUDA编程模型简介
  - CUDA编程模型实例
  - CUDA编程模型小结
- 持久性CUDA编程模型
  - 常规CUDA编程模型的不足
  - 持久性CUDA编程模型简介
  - 持久性CUDA编程模型与常规编程模型对比
  - 持久性CUDA编程模型的优势与不足
- 案例: Persistent GEMM for WaveRNN

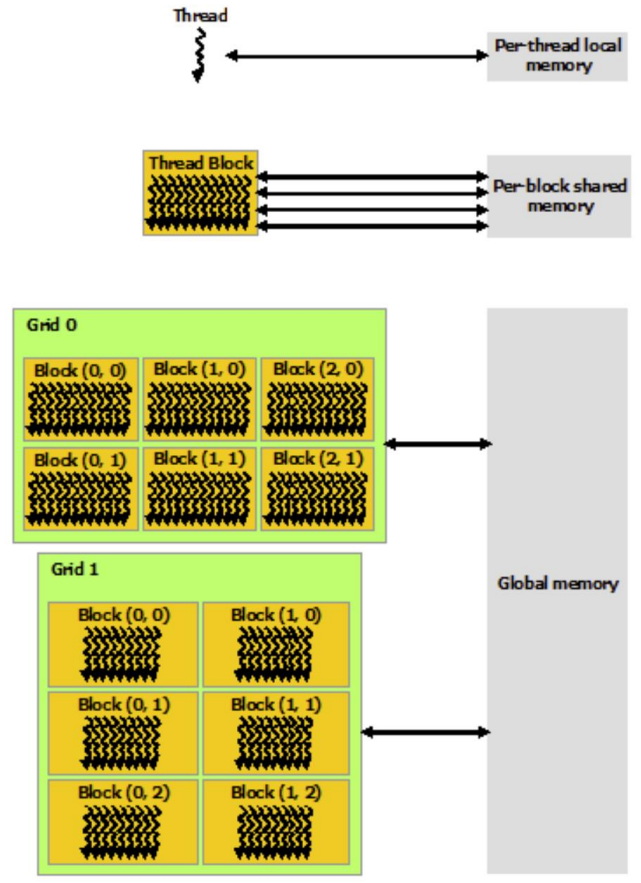
CUDA编程模型简介

- 基于数据并行的高度并发编程模型
  - SIMT模型：发射大量线程，对不同的数据执行同一段代码
  - 2层级的线程组织形式
  - 每个CUDA thread映射到一个数据元素上
  - 发射足够多的线程以覆盖全部要处理的数据



CUDA编程模型简介

- 基于并发模型的内存层次结构
  - 多层级内存层次
  - 减少局部性内存访问开销
  - 与编程模型紧密配合



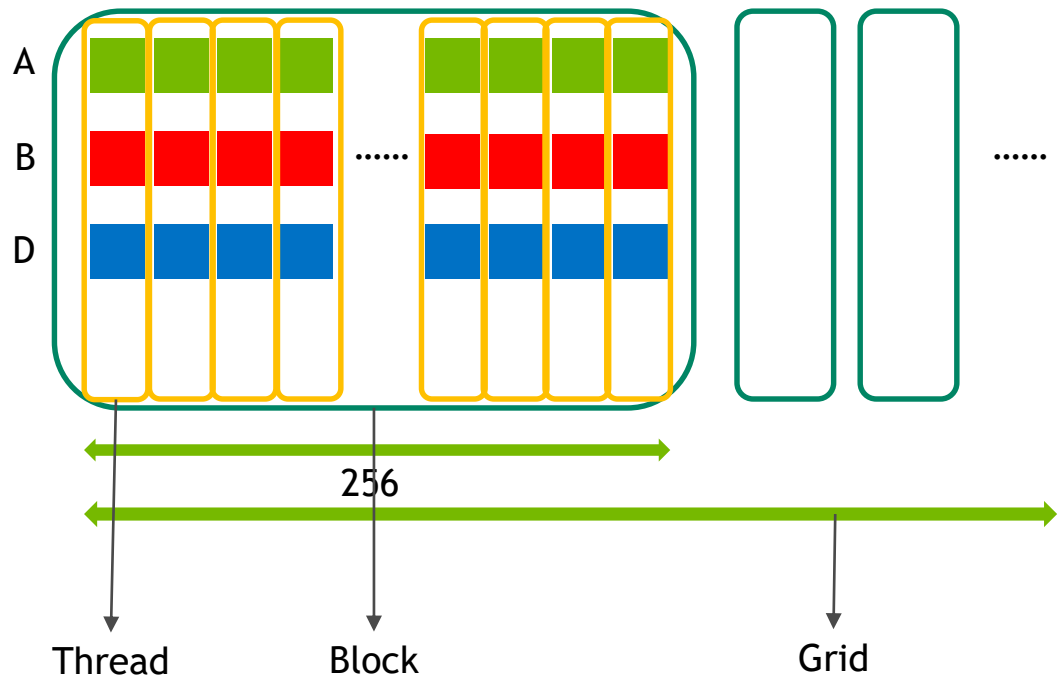
CUDA编程模型实例

- 向量相加
  - 分配源和目标向量的GPU内存
  - 将A和B向量拷贝至GPU内存
  - 书写GPU kernel：每个线程的程序负责一个元素的相加
  - 发射GPU kernel：block的大小 * grid的大小 >= 向量长度
  - 等待kernel完成将结果拷回并使用

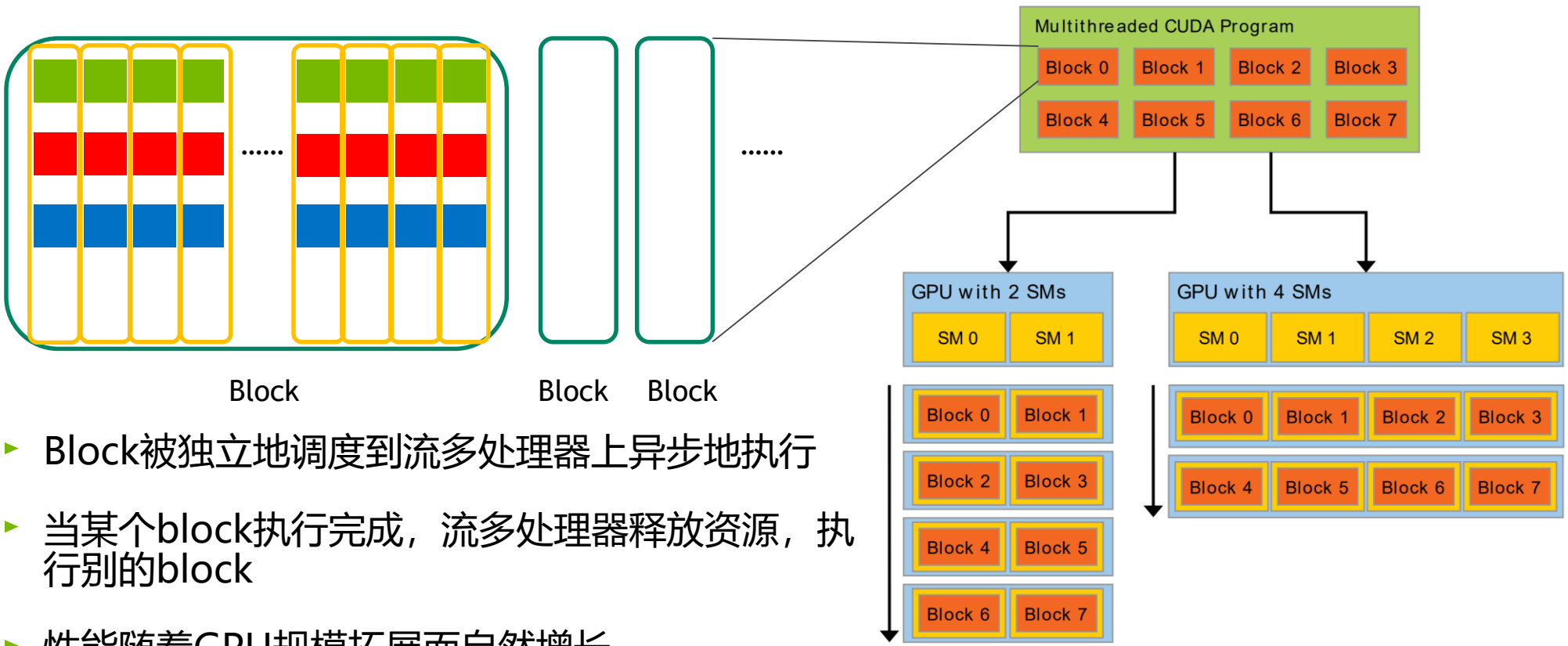
```
#define N 10000
#define BLOCK_SIZE 256
__global__ void vec_add(float * A, float * B, float * D, size_t len){
    size_t index = threadIdx.x + blockIdx.x * blockDim.x;
    if(index < len){
        D[index] = A[index] + B[index];
    }
}

int main(){
    // Allocate vector buffers, copy vectors to GPU etc.
    size_t grid_size = (N - 1) / BLOCK_SIZE + 1;
    vec_add<<<grid_size, BLOCK_SIZE>>>(A, B, D, N);
    // Wait for kernel to finish, copy data back and consume etc.
}
```

CUDA编程模型实例

```
#define N 10000
#define BLOCK_SIZE 256
__global__ void vec_add(float * A, float * B, float * D, size_t len){
    size_t index = threadIdx.x + blockIdx.x * blockDim.x;
    if(index < len){
        D[index] = A[index] + B[index];
    }
}

int main(){
    // Allocate vector buffers, copy vectors to GPU etc.
    size_t grid_size = (N - 1) / BLOCK_SIZE + 1;
    vec_add<<<grid_size, BLOCK_SIZE>>>(A, B, D, N);
    // Wait for kernel to finish, copy data back and consume etc.
}
```



CUDA编程模型实例



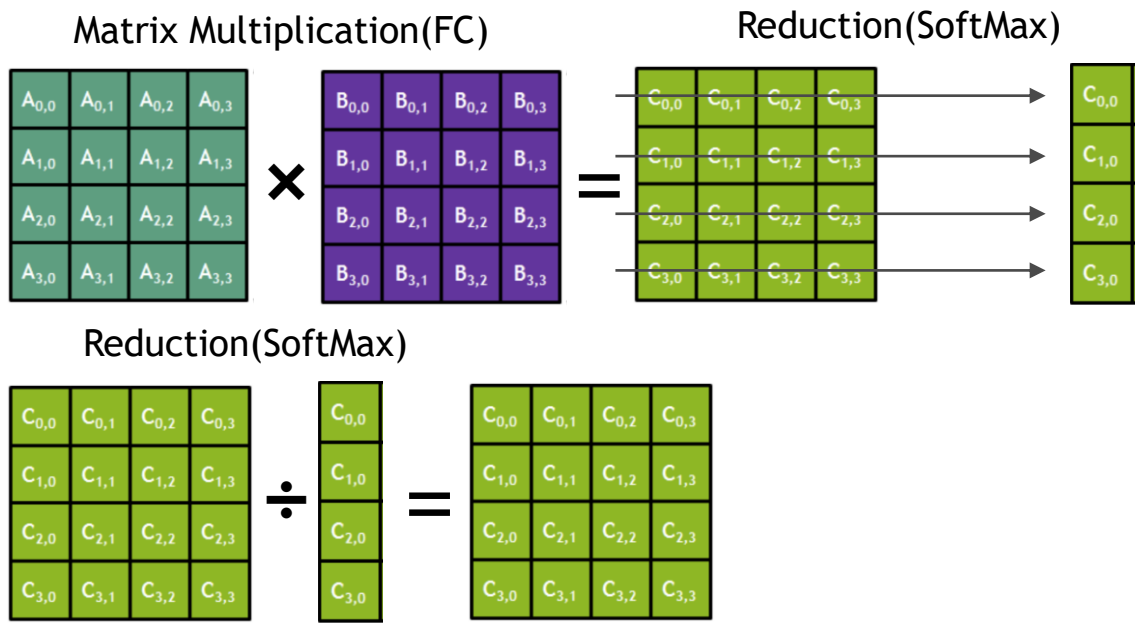
- Block被独立地调度到流多处理器上异步地执行
- 当某个block执行完成，流多处理器释放资源，执行别的block
- 性能随着GPU规模拓展而自然增长

CUDA编程模型小结

- 通用性强：操作写好后可反复调用，以任何规模运行
- 高并发行：大量线程并发执行，并发内存访问
- 高拓展性：block彼此独立，自适应于不同规模的GPU

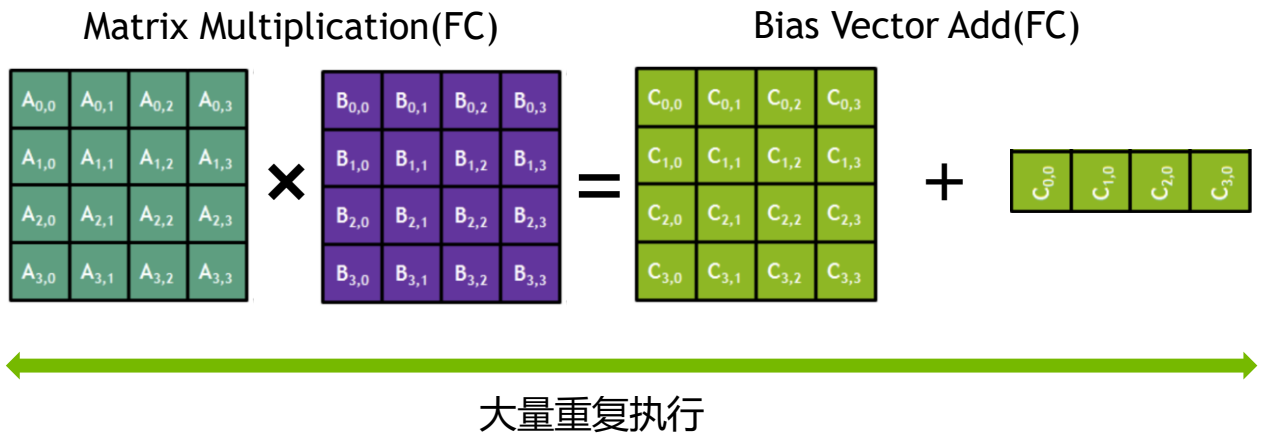
CUDA编程模型不足

- 场景1：
  - A和B两个操作
  - B操作依赖于A操作的结果
  - 需要全局同步
  - 必须分成两个kernel执行
  - 付出kernel launch等代价
- CUDA block 完全异步，同步依赖于kernel边界



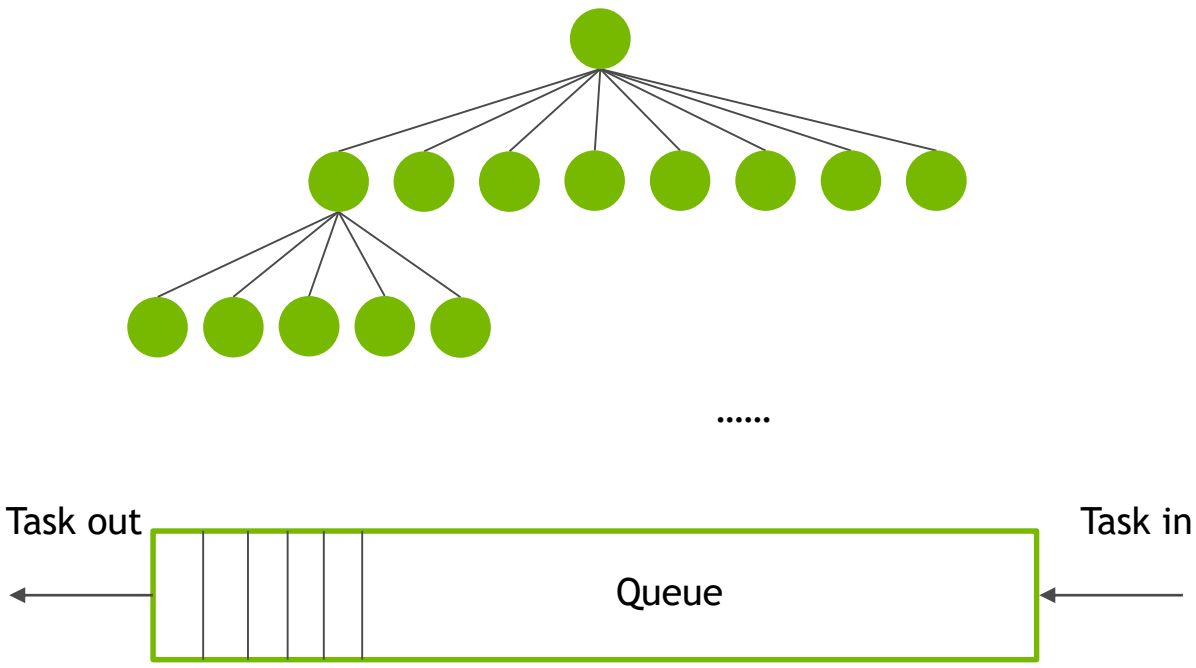
CUDA编程模型不足

- 场景2：
  - A操作是常用操作，反复执行
  - A操作有大量局部数据环境
  - 每次执行A操作需要加载数据环境
  - 付出内存访问地代价
- Block以及thread的局部数据环境只在生存期内有效



CUDA编程模型不足

- 场景3：
  - 遍历不规则数据结构，或者CPU不断送达任务
  - 任务动态的产生，动态的分配
  - 负载需要均衡，以最大化利用GPU
- Block的调度工作完全由硬件负责，每个线程的状态（激活或退出）不能控制



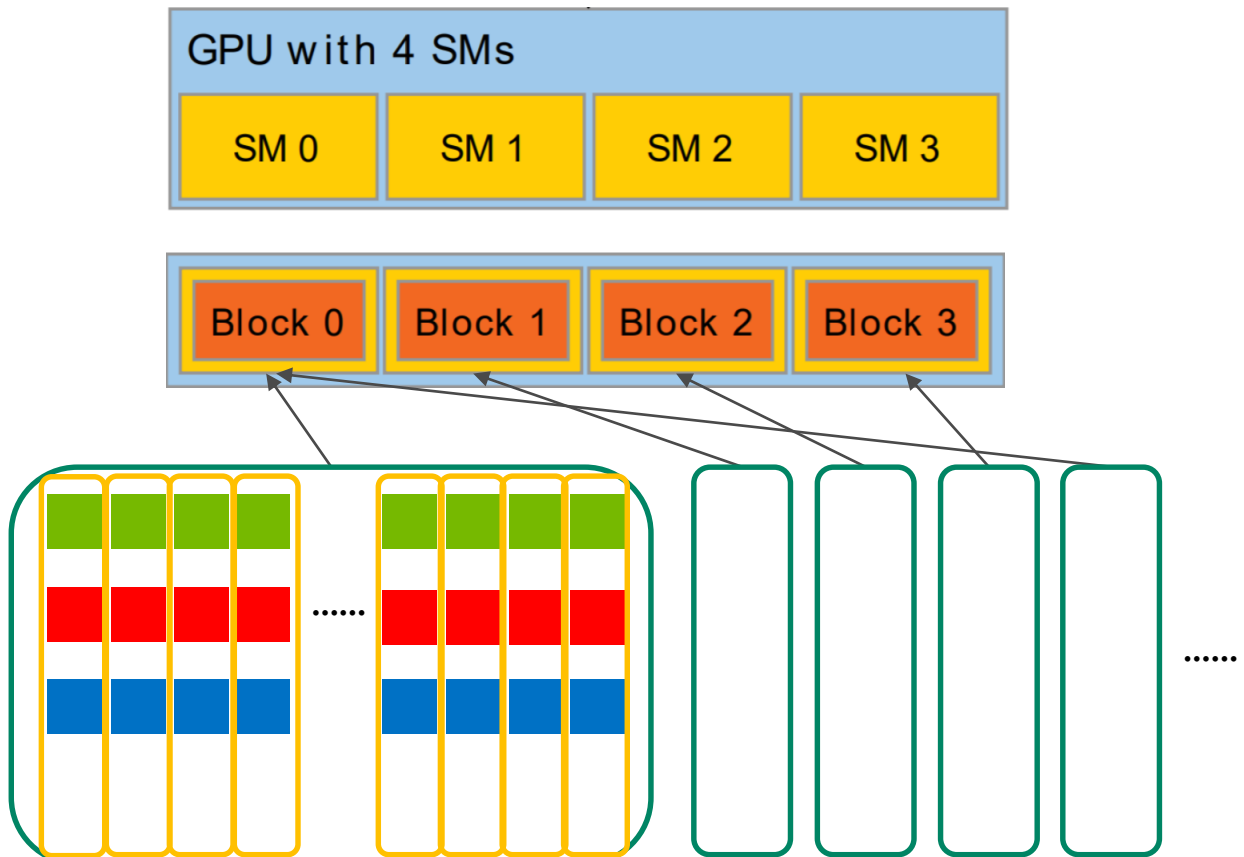
持久性CUDA编程模型简介

- 持久性向量相加
  - 书写GPU kernel：每个线程负责多个元素相加
  - 发射GPU kernel：grid的大小 = 流多处理器的数量 或者 GPU所能容纳的block的数量

```
int main(){
    // Allocate vector buffers, copy vectors to GPU etc.
    int grid_size;
    cudaDeviceProp deviceProp;
    int numBlocksPerSm;

    // Launch as many blocks as SMs count
    cudaGetDeviceProperties(&deviceProp, target_device);
    grid_size = deviceProp.multiProcessorCount;

    // Alternatively, launch as many blocks as can fit simultaneously per-SM
    cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, vec_add, BLOCK_SIZE, 0);
    grid_size = deviceProp.multiProcessorCount*numBlocksPerSm;

    // Launch kernel
    vec_add<<<grid_size, BLOCK_SIZE>>>(A, B, D, N);

    // Wait for kernel to finish, copy data back and consume etc.
}

__global__ void vec_add(float * A, float * B, float * D, size_t len){
    size_t index = threadIdx.x + blockIdx.x * blockDim.x;
    size_t grid_size = gridDim.x * blockDim.x;
    for(size_t i = index; i < len; i += grid_size)
        D[index] = A[index] + B[index];
    }
}
```

持久性CUDA编程模型简介

- 持久性向量相加
  - 书写GPU kernel：每个线程负责多个元素相加
  - 发射GPU kernel：grid的大小 = 流多处理器的数量 或者 GPU所能容纳的block的数量

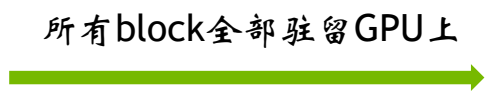


编程模型对比

- 常规CUDA编程模型特点
  - 一个线程负责一个数据元素，发射足够多的线程以覆盖数据空间
  - 线程以及block完全异步执行，硬件负责调度
  - 当block执行完成，空出资源以供其他block执行
  - 当一个操作完成时，结束kernel完成全局同步，之后发射新的kernel执行新的操作
- 持久性CUDA编程模型特点
  - Kernel只发射可以同时在GPU上执行的block和线程数量
  - 所有的block全部驻留在GPU片上，任务调度可以交由软件负责
  - 当block执行完成时，等待全局同步信号，准备执行下一个任务
  - 当一个操作完成时，等待GPU全局同步，所有block继续执行下一个操作

持久性CUDA编程模型优势

- 持久性CUDA编程模型
  - Kernel只发射可以同时在GPU上执行的block和线程数量
  - 所有的block全部驻留在GPU片上，任务调度可以交由软件负责
  - 当block执行完成时，等待全局同步信号，准备执行下一个任务
  - 当一个操作完成时，等待GPU全局同步，所有block继续执行下一个操作



- 持久性CUDA编程优势
  - 允许用户定义的GPU片上同步：不用反复发射kernel
  - 所有数据环境（寄存器，shared memory等）可以保留重用
  - 任务调度和负载均衡交由用户软件来负责

持久性CUDA编程模型不足

- 持久性CUDA编程模型
  - Kernel只发射可以同时在GPU上执行的block和线程数量
  - 所有的block全部驻留在GPU片上，任务调度可以交由软件负责
  - 当block执行完成时，等待全局同步信号，准备执行下一个任务
  - 当一个操作完成时，等待GPU全局同步，所有block继续执行下一个操作

所有block全部驻留GPU上

- 持久性CUDA编程不足
  - 通用性差：发射情况取决于GPU的规模，不同的GPU可能需要不同的kernel。对于不同的任务规模也需要重新划分任务，代码需要针对不同的场景修改
  - 可拓展性弱：一套代码只有较为固定的并行规模，只能针对比较固定的运行环境
  - 工程量大：block的驻留性，自定义同步，以及片上存储的管理等都需要开发者自行完成，没有现成的官方定义，复杂，容易出错


案例：PERSISTENT
GEMM FOR WAVERNN

持久性CUDA编程与深度学习

- 深度学习一般分为两个过程
  - 训练：通过前向和后向的过程，不断学习模型的参数
  - 推理：利用训练好的模型，处理实际的输入数据
- 推理过程需要使用大量的（基本）不会变化的参数
- 推理过程的计算会被大量重复
  - 对于前馈网络来说，每推理一批输入，需要使用一次参数
  - 对于带有反馈结构的网络来说，每个时间步都需要使用一次参数
- 内存开销巨大，尽可能复用片上存储是性能的关键
  - 对于RNN类网络尤其重要，选择WaveRNN的推理过程作为目标应用

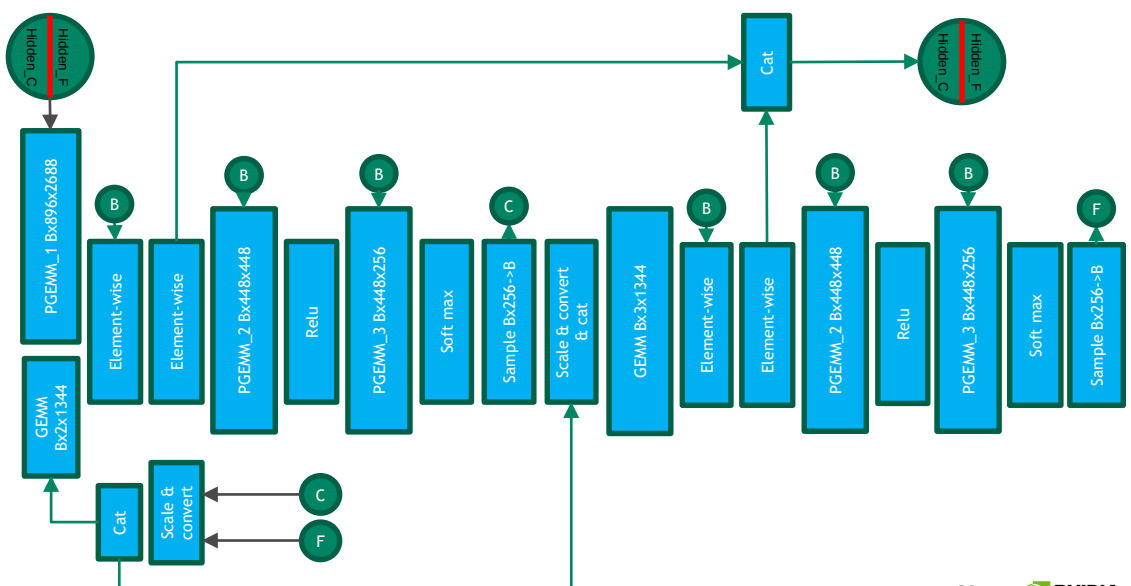
WAVERNN 简介

- Deepmind 开发的音频合成模型
- 相对较少的参数量，较为简单的模型，非常适合移动端处理器
- 不错的精度，接近标准的WaveNet

WAVERNN 简介

- WaveRNN每个时间步分为先后两轮计算
- WaveRNN两轮计算先后产生8位粗采样和8位细采样，两个部分合成一个16位采样
- 每轮计算都会对输入做矩阵相乘
- 两轮计算共用一个巨大的hidden state矩阵相乘
- 每轮计算都会对输出做两次矩阵相乘

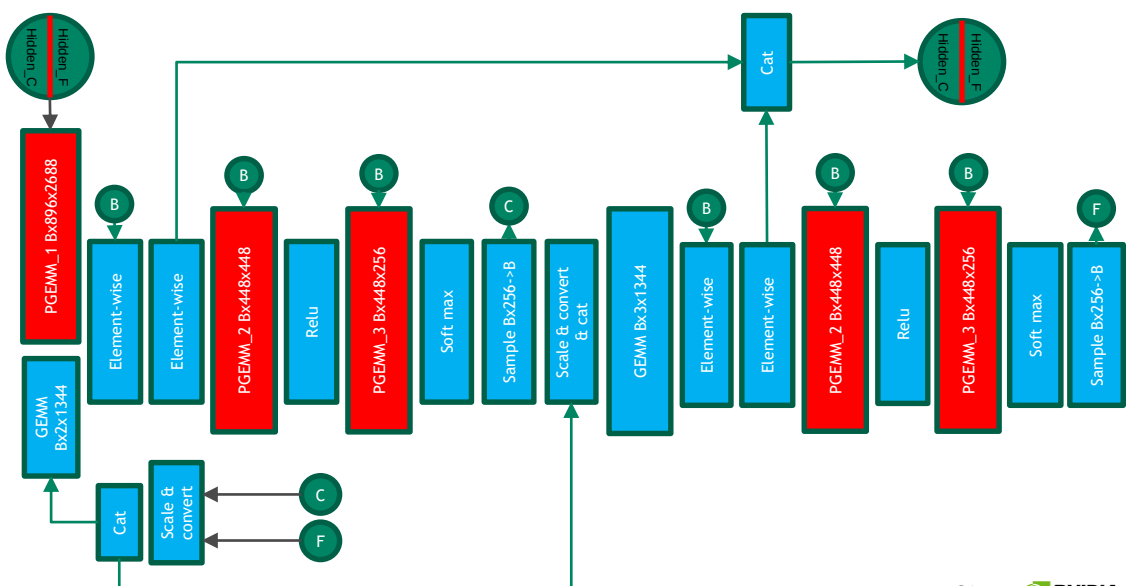
$$
\begin{aligned}
\mathbf{x}_t &= [\mathbf{c}_{t-1}, \mathbf{f}_{t-1}, \mathbf{c}_t] \\
\mathbf{u}_t &= \sigma(\mathbf{R}_u \mathbf{h}_{t-1} + \mathbf{I}_u^\star \mathbf{x}_t) \\
\mathbf{r}_t &= \sigma(\mathbf{R}_r \mathbf{h}_{t-1} + \mathbf{I}_r^\star \mathbf{x}_t) \\
\mathbf{e}_t &= \tau(\mathbf{r}_t \circ (\mathbf{R}_e \mathbf{h}_{t-1}) + \mathbf{I}_e^\star \mathbf{x}_t) \\
\mathbf{h}_t &= \mathbf{u}_t \circ \mathbf{h}_{t-1} + (1 - \mathbf{u}_t) \circ \mathbf{e}_t \\
\mathbf{y}_c, \mathbf{y}_f &= \text{split}(\mathbf{h}_t) \\
P(\mathbf{c}_t) &= \text{softmax}(\mathbf{O}_2 \text{ relu}(\mathbf{O}_1 \mathbf{y}_c)) \\
P(\mathbf{f}_t) &= \text{softmax}(\mathbf{O}_4 \text{ relu}(\mathbf{O}_3 \mathbf{y}_f))
\end{aligned}
$$



WAVERNN 简介

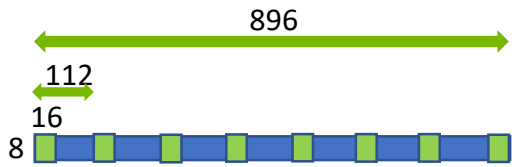
- 使用WaveRNN-896，整个网络参数量~3M
- 主要参数/计算集中在R, O_1, O_2, O_3, O_4（红色），首先实现由他们组成的Persistent GEMM kernels
- 目标设备：最先进的推理GPU：Tesla T4 (Turing, 40流多处理器)
- 混合精度实现：Persistent GEMM（红色）为int8精度，其余部分为FP32精度
- 低精度GEMM意味着更高性能的硬件支持，Persistent GEMM由Tensor Core实现

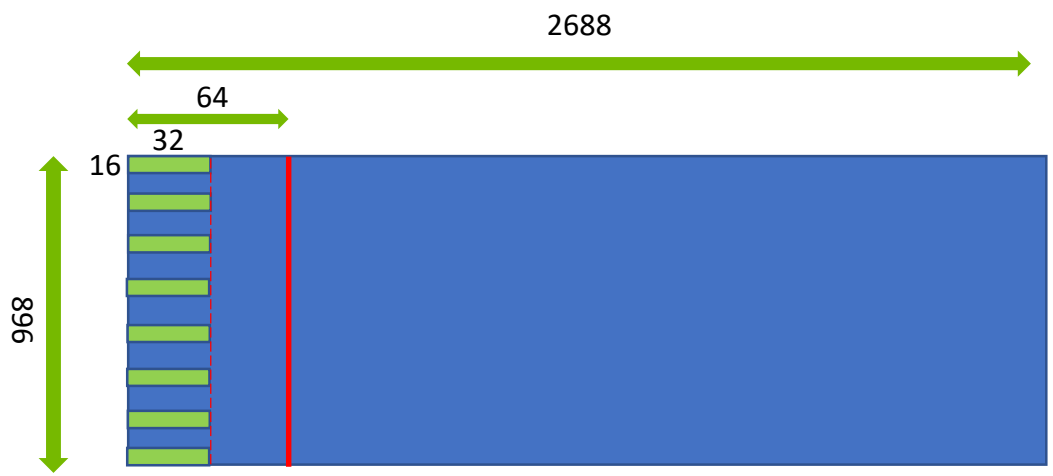
$$
\begin{aligned}
\mathbf{x}_t &= [\mathbf{c}_{t-1}, \mathbf{f}_{t-1}, \mathbf{c}_t] \\
\mathbf{u}_t &= \sigma(\mathbf{R}_u \mathbf{h}_{t-1} + \mathbf{I}_u^\star \mathbf{x}_t) \\
\mathbf{r}_t &= \sigma(\mathbf{R}_r \mathbf{h}_{t-1} + \mathbf{I}_r^\star \mathbf{x}_t) \\
\mathbf{e}_t &= \tau(\mathbf{r}_t \circ (\mathbf{R}_e \mathbf{h}_{t-1}) + \mathbf{I}_e^\star \mathbf{x}_t) \\
\mathbf{h}_t &= \mathbf{u}_t \circ \mathbf{h}_{t-1} + (1 - \mathbf{u}_t) \circ \mathbf{e}_t \\
\mathbf{y}_c, \mathbf{y}_f &= \text{split}(\mathbf{h}_t) \\
P(\mathbf{c}_t) &= \text{softmax}(\mathbf{O}_2 \text{ relu}(\mathbf{O}_1 \mathbf{y}_c)) \\
P(\mathbf{f}_t) &= \text{softmax}(\mathbf{O}_4 \text{ relu}(\mathbf{O}_3 \mathbf{y}_f))
\end{aligned}
$$



PERSISTENT GEMM：设计

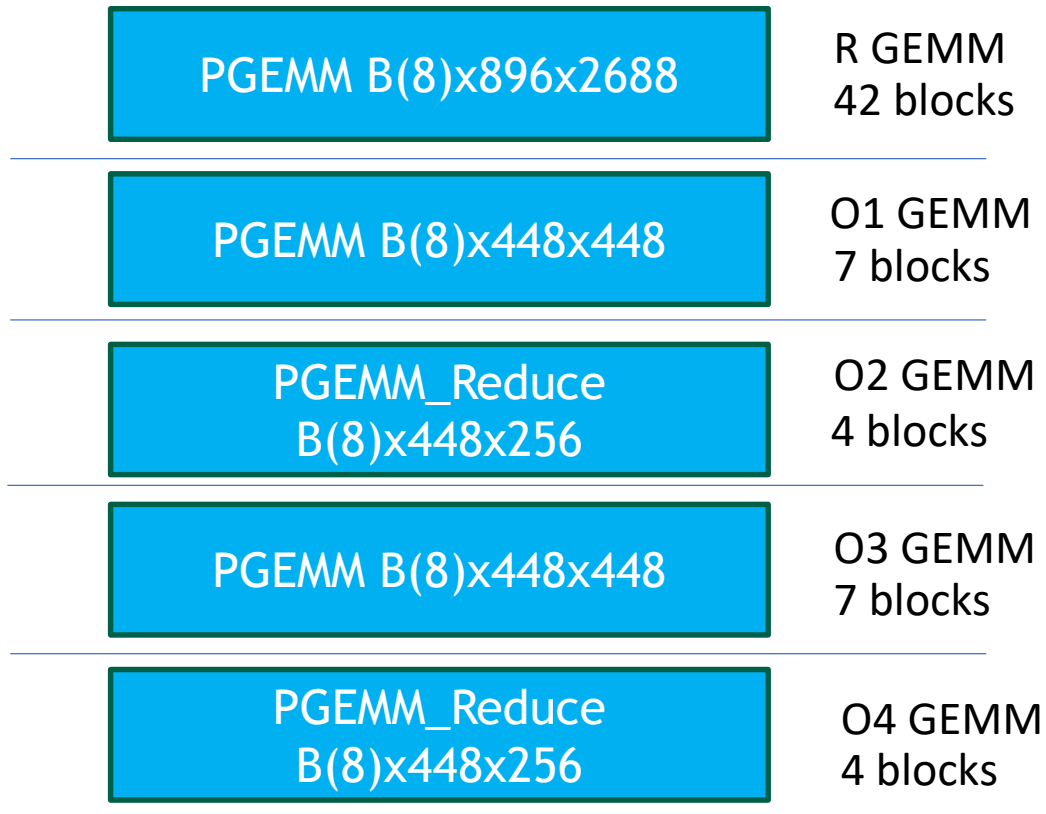
- 以R矩阵相乘设计为例
- Tensor Core最基本计算单位：A矩阵（8x16），B矩阵（16x32），结果矩阵（8x32）,8为满足需求的最小batch大小
- Tensor Core以warp为单位调度和计算：每个block 14个warp，负责8x896x64的矩阵相乘，一共需要42个block
- 每个warp需要循环完成8次Tensor core矩阵相乘
- 每个warp需要的参数矩阵R的部分，分布式地存储在组成warp的所有线程的寄存器当中





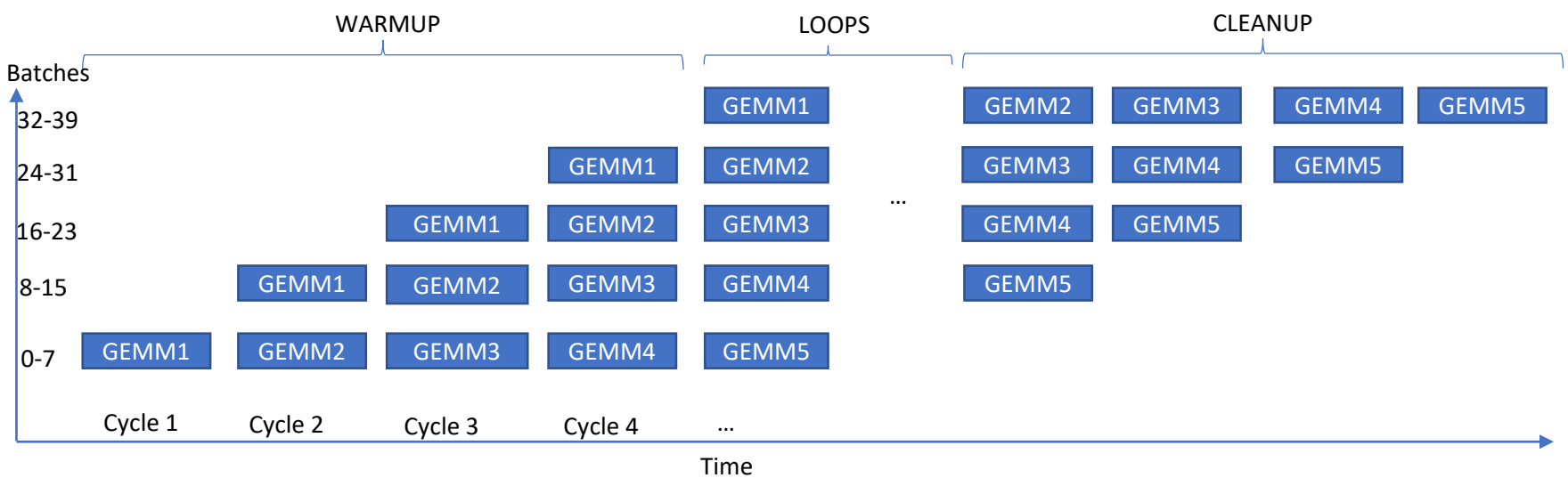
PERSISTENT GEMM：设计

- 5个GEMM一共需要64个block来完成，全部64个block驻留在GPU上，所有的参数矩阵分布在寄存器中，在kernel退出前不会失效
- 为了增加灵活性，每个GEMM均由单独的kernel实现，最终5个kernel并发的执行，并全部驻留在片上
- GEMM需要的计算资源差距较大，某些GEMM的规模不能有效地利用Tesla T4，为了最大化利用GPU，设计了一个5级片上流水线



PERSISTENT GEMM：设计

- 一共5组batch同时并发的执行，每组batch依次经过5级流水线以完成一个时间步的生成工作
- 流水线的吞吐为：8batch时间步/周期
- 每个周期结束后，需要对64个block全局同步，然后进入下个周期



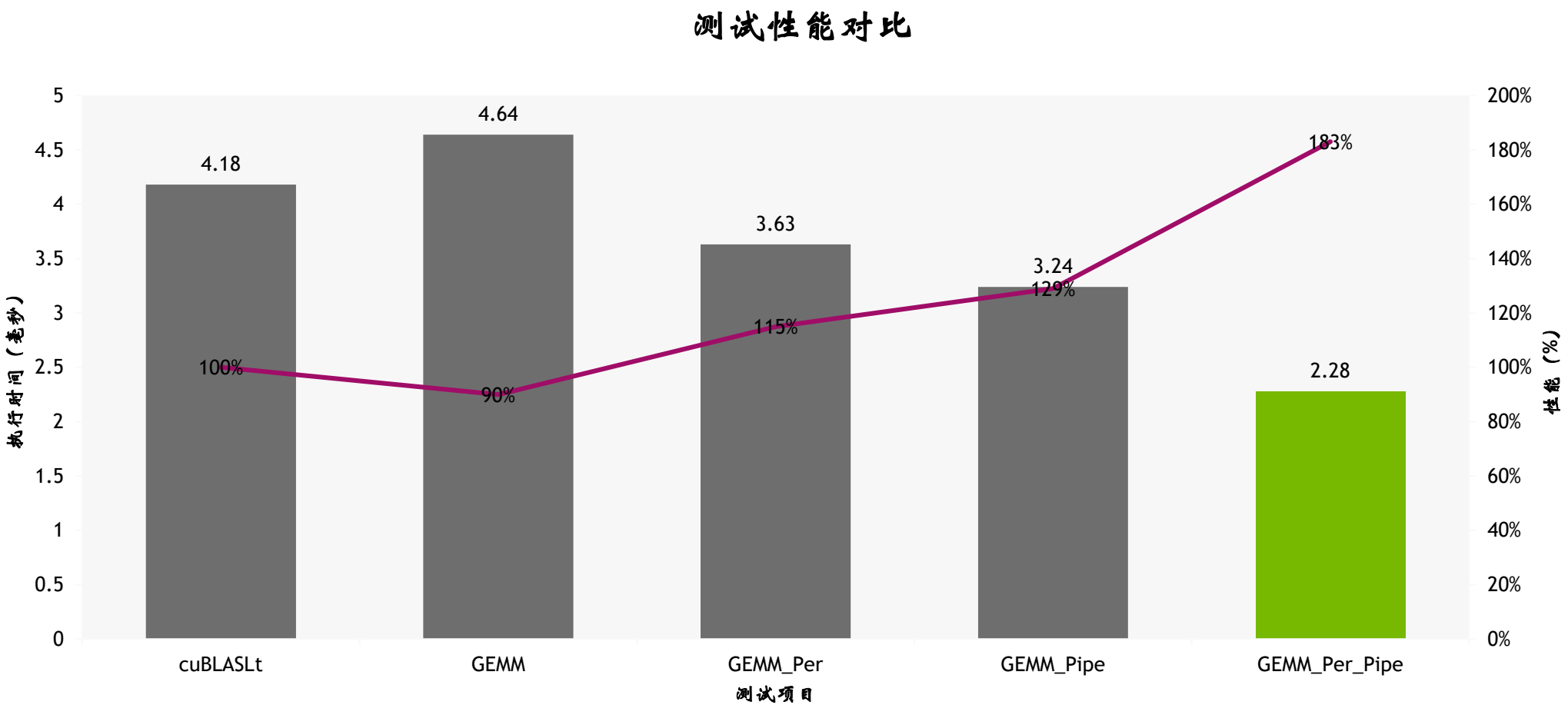
PERSISTENT GEMM：性能

- 测试任务：5组batch size为8的数据完成32个时间步
  - 每组batch需要完成5x32个GEMM计算
  - 总计算量为800个有效GEMM
- 测试1：使用NVIDIA官方的cuBLASLt库完成
  - Batch size为40，即40x896x2688，40x448x448，40x448x256的GEMM大小
  - CPU调用5个kernel顺序执行以完成40个batch的1个时间步，循环32次
- 测试2：使用我们自己实现的GEMM程序完成
  - Batch size为40
  - 既不持久性执行，也不使用流水线

PERSISTENT GEMM：性能

- 测试3：使用我们自己实现的GEMM程序完成
  - Batch size为8
  - 持久性执行，但不使用流水线
- 测试4：使用我们自己实现的GEMM程序完成
  - Batch size为8
  - 使用流水线，但不持久性执行
- 测试5：使用我们自己实现的GEMM程序完成
  - Batch size为8
  - 持久性执行，且使用流水线

PERSISTENT GEMM：性能



PERSISTENT WAVERNN：小结与展望

- 采用多种优化技术
  - 混合精度：充分使用Tensor Core
  - 持久性编程：充分利用片上存储避免反复的参数矩阵读取，利用片上全局同步避免反复地发射kernel
  - 片上流水线：充分利用整个GPU
- 未来计划
  - 完成Persistent WaveRNN剩下的操作：量化，反量化，其他操作等
  - 进一步增强持久性：kernel永不退出，和CPU形成client-server模式
  - 进一步增强负载均衡：64block不是40流多处理器的整数倍，考虑更优的实现
  - 寻求更多的持久性模式

持久性CUDA GPU编程：小结与展望

- 特点：全部block和线程都驻留在GPU上，直到完成全部的工作
- 与常规CUDA编程相比利弊明显
  - 更高的可控性，更好的性能
  - 更大的工程量，更复杂的设计，更高的难度
- 展望
  - 非常适合深度学习，尤其是RNN类网络的负载
  - 该技术以及类似的kernel fusion技术在未来会有更多的应用和支持


NVIDIA