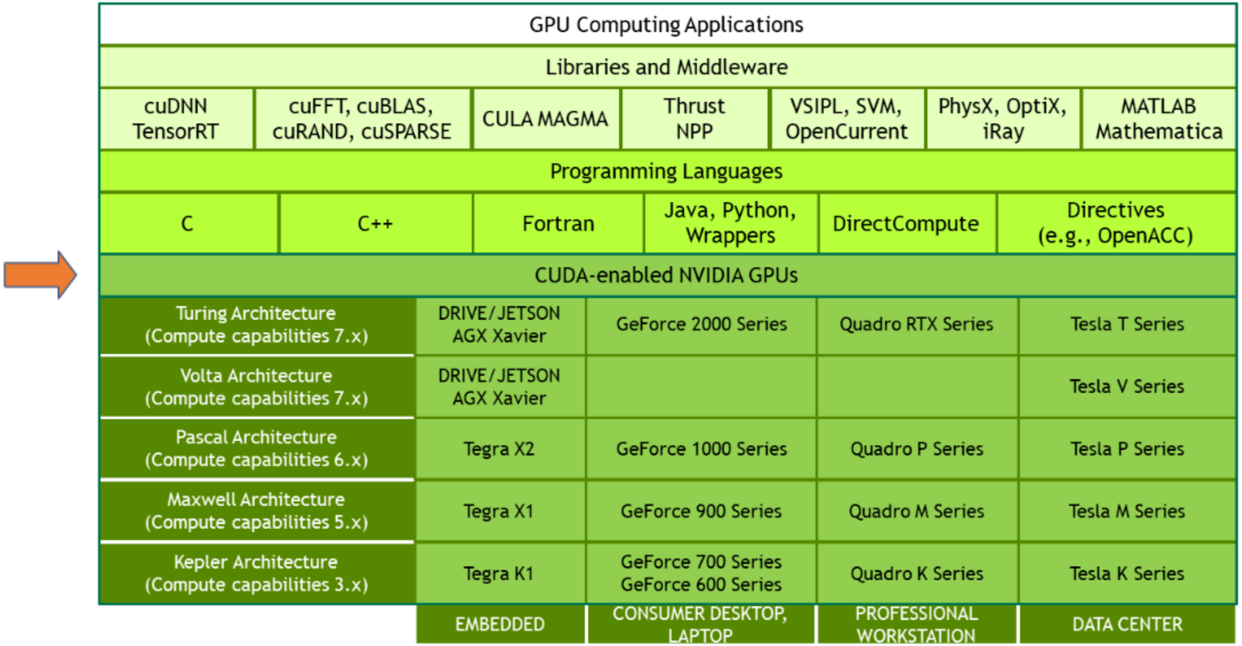
1. GPU和CUDA编程模型
2. CUDA优化 – 矩阵乘法
3. Fastertransformer概述
4. TensorRT概述

CUDA – 并行计算平台和编程模型



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica

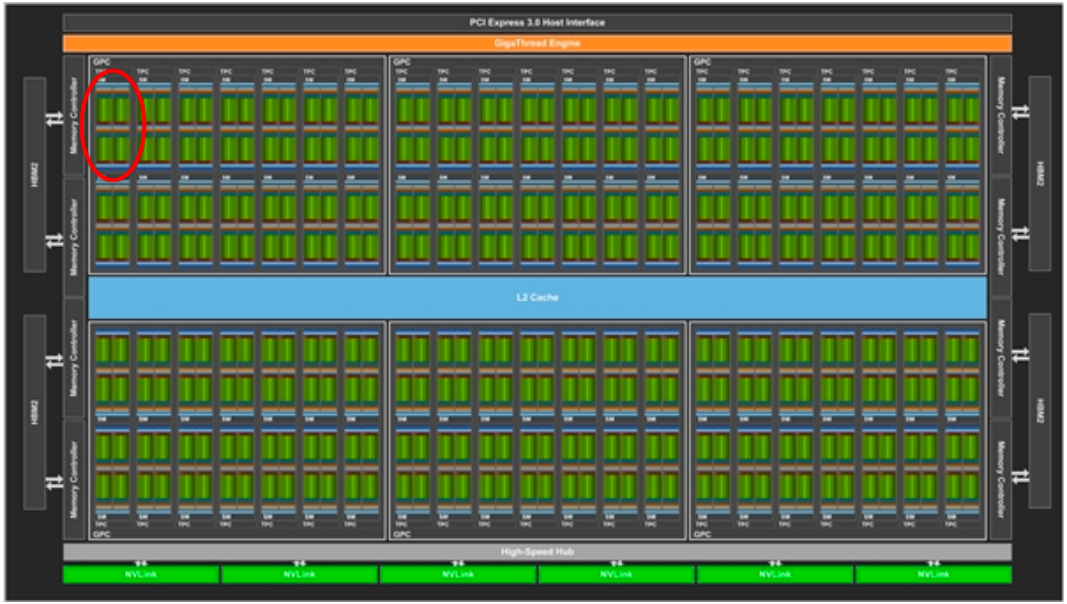
Programming Languages

C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)

CUDA-enabled NVIDIA GPUs

	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series

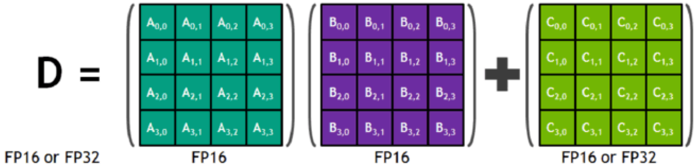
V100架构：84个SM(Streaming Multiprocessor)



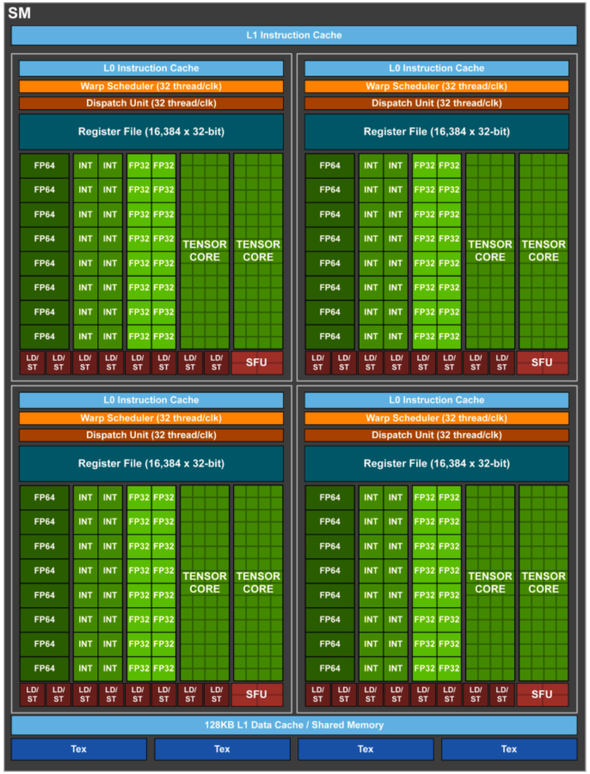
SM：运算和调度的基本单元

1. CUDA cores
2. Scheduler/Dispatcher
3. Shared memory, register file, L1 cache

Tensor Core支持混合精度：



D =
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$
FP16 or FP32
FP16
FP16
FP16 or FP32



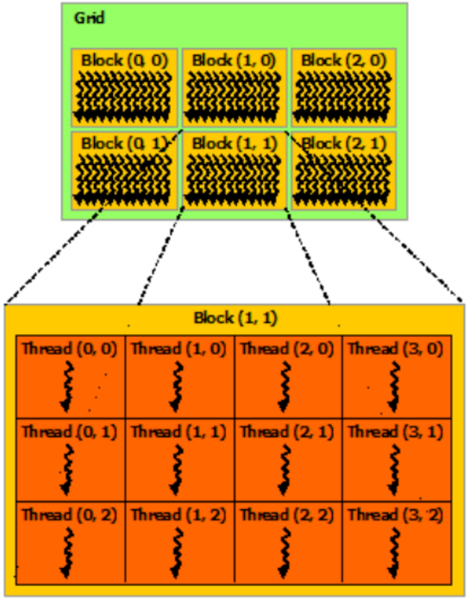
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

CUDA编程模型：Kernel、Thread、Block、Grid

- Kernel：GPU上每个Thread执行的程序
- Grid > Block > Thread

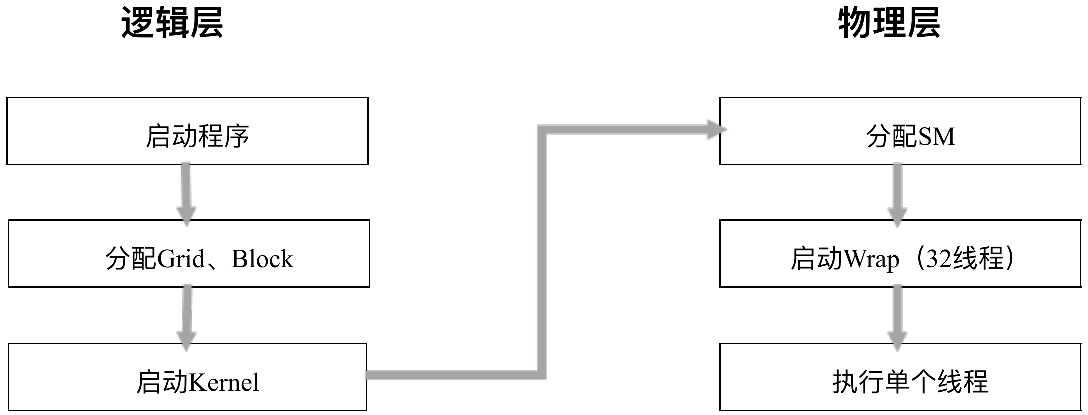
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int row = blockIdx.x;
    int col = threadIdx.x;
    if (row < N && col < N)
        C[row][col] = A[row][col] + B[row][col];
}

int main()
{
    ...
    // Kernel invocation
    dim3 numBlocks(N); //grid dim
    dim3 threadsPerBlock(N); //block dim
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

CUDA编程模型：Kernel、Thread、Block、Grid



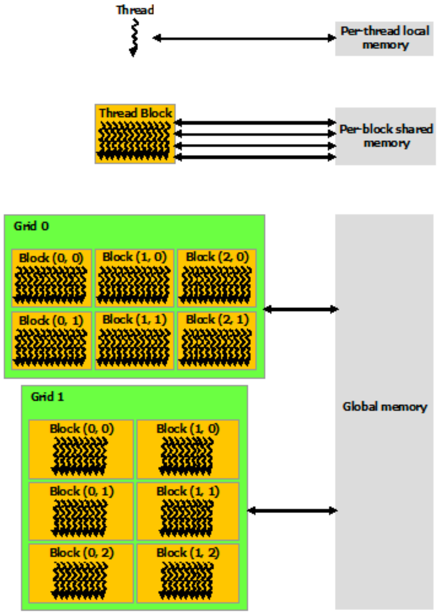
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x; //find row
    int j = blockIdx.y * blockDim.y + threadIdx.y; //find col
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

CUDA程序执行流程



- Wrap：SM中的SIMD线程（Single Instruction Multiple Data，单指令流多数据流）
- CPU使用的是MIMD（Multiple Instruction Multiple Data，多指令流多数据流）架构

CUDA优化：内存优化



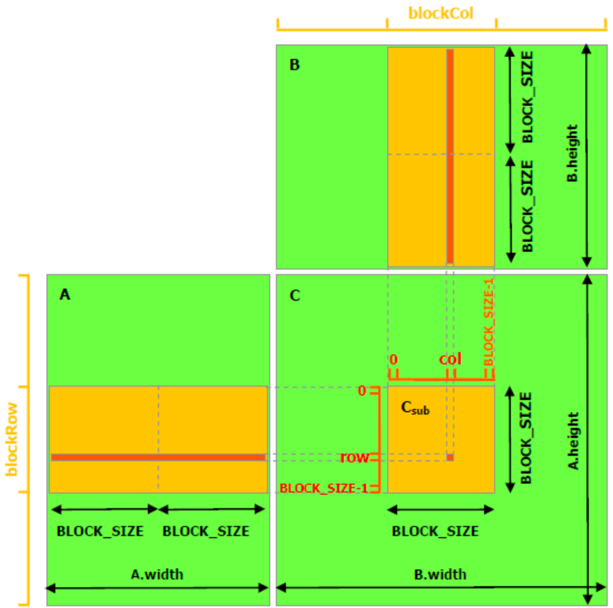
- Kernel运行时只能访问到GPU的内存，所以任务执行前后都需要在host(CPU)和device(GPU)之间进行数据交互

- 访问速度：local memory > shared memory > global memory

- 优化方法：将数据拷贝到block的共享内存，加快数据访问速度

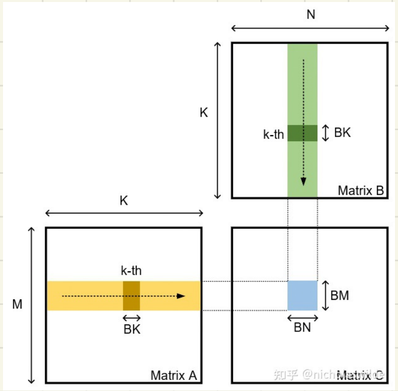
CUDA内存优化举例：矩阵乘法

```
void main()
{
    ...
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    ...
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

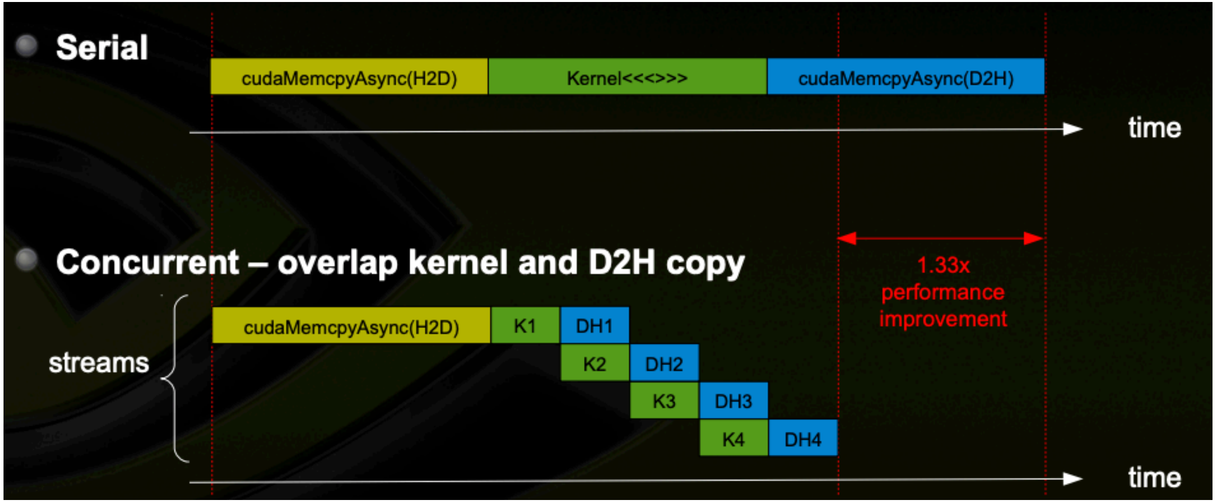


上面这种每个线程计算一个元素简单的SGEMM kernal显然是效率非常低的，需要两次Global Memory的load才能完成一次乘累加运算，计算访存比极低，没有有效的数据复用。

那么，我们接下来尝试这样一种方法：每个Block负责计算矩阵C中的BM * BN个元素，每个Thread负责计算矩阵C中的TM * TN个元素；使用Shared Memory来存放线程间可以复用的矩阵A和矩阵B的元素，一个Block的所有线程每次一起从Global Memory中load BM * BK个矩阵A的元素和BK * BN个矩阵B的元素，存放到Shared Memory中，之后每个线程再分别从Shared Memory中取数运算，这样的循环一共需要K / BK次以完成整个矩阵乘法操作，如下图所示。

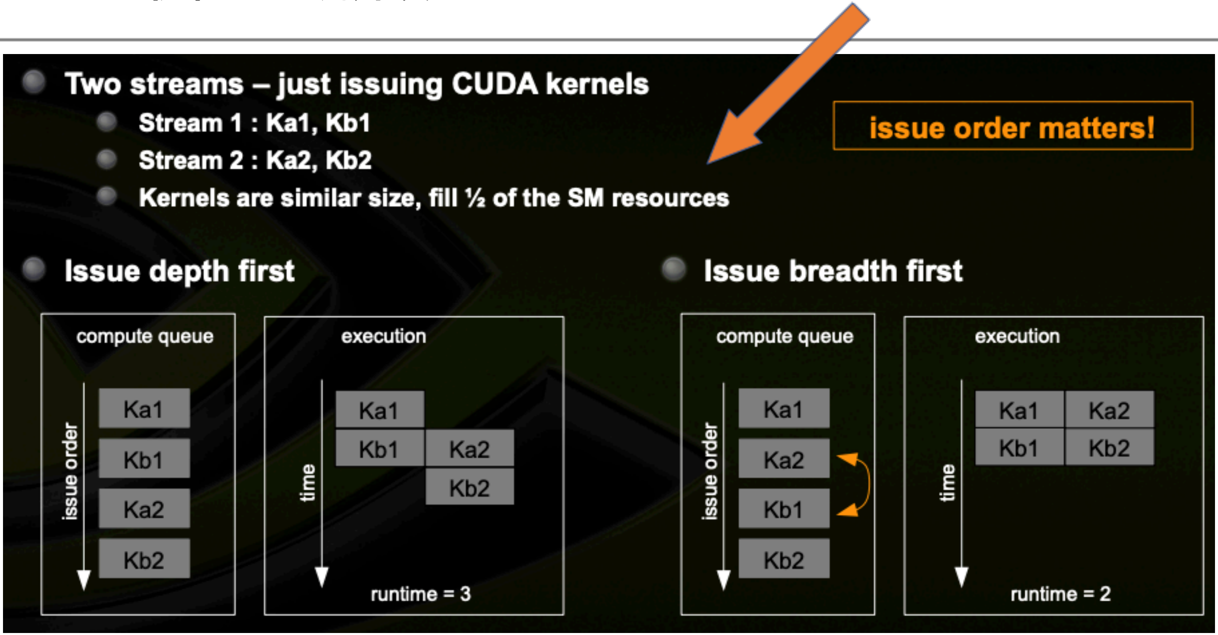


CUDA优化：多流并发



Serial
cudaMemcpyAsync(H2D)
Kernel<<<>>>
cudaMemcpyAsync(D2H)
time
Concurrent – overlap kernel and D2H copy
1.33x performance improvement
streams
cudaMemcpyAsync(H2D)
K1
DH1
K2
DH2
K3
DH3
K4
DH4
time

CUDA优化：多流并发



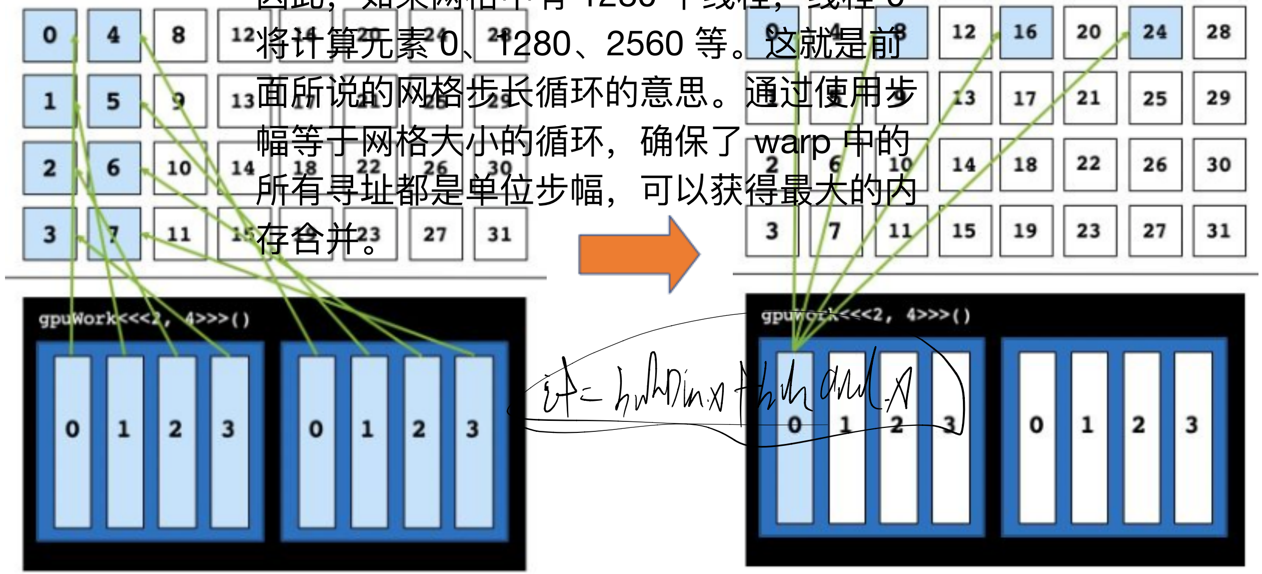
- Two streams – just issuing CUDA kernels
  - Stream 1 : Ka1, Kb1
  - Stream 2 : Ka2, Kb2
  - Kernels are similar size, fill ½ of the SM resources

issue order matters!

- Issue depth first
- Issue breadth first

CUDA优化：并发算法之Grid-Stride Loop（网格跨步循环）

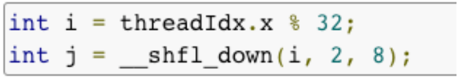
因此，如果网格中有 1280 个线程，线程 0 将计算元素 0、1280、2560 等。这就是前面所说的网格步长循环的意思。通过使用步幅等于网格大小的循环，确保了 warp 中的所有寻址都是单位步幅，可以获得最大的内存合并。



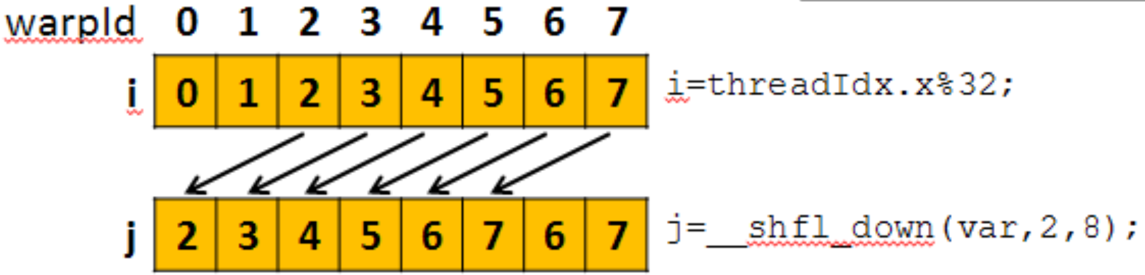
CUDA优化：并发算法之Parallel reduction （并行规约）

Parallel reductions exchange data between threads within the same thread block.

- Shuffle On Down



```
int i = threadIdx.x % 32;
int j = __shfl_down(i, 2, 8);
```

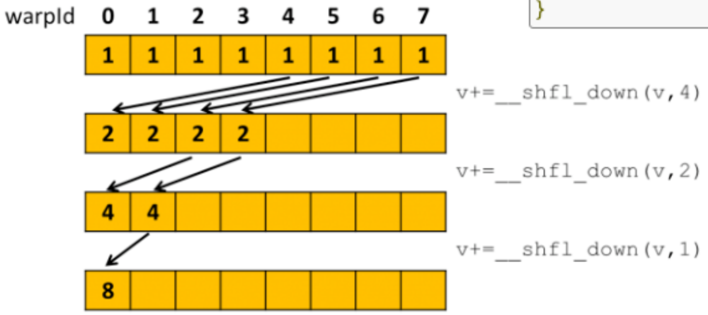


```
i=threadIdx.x%32;
j=__shfl_down(var,2,8);
```

CUDA优化：并发算法之Parallel reduction （并行规约）

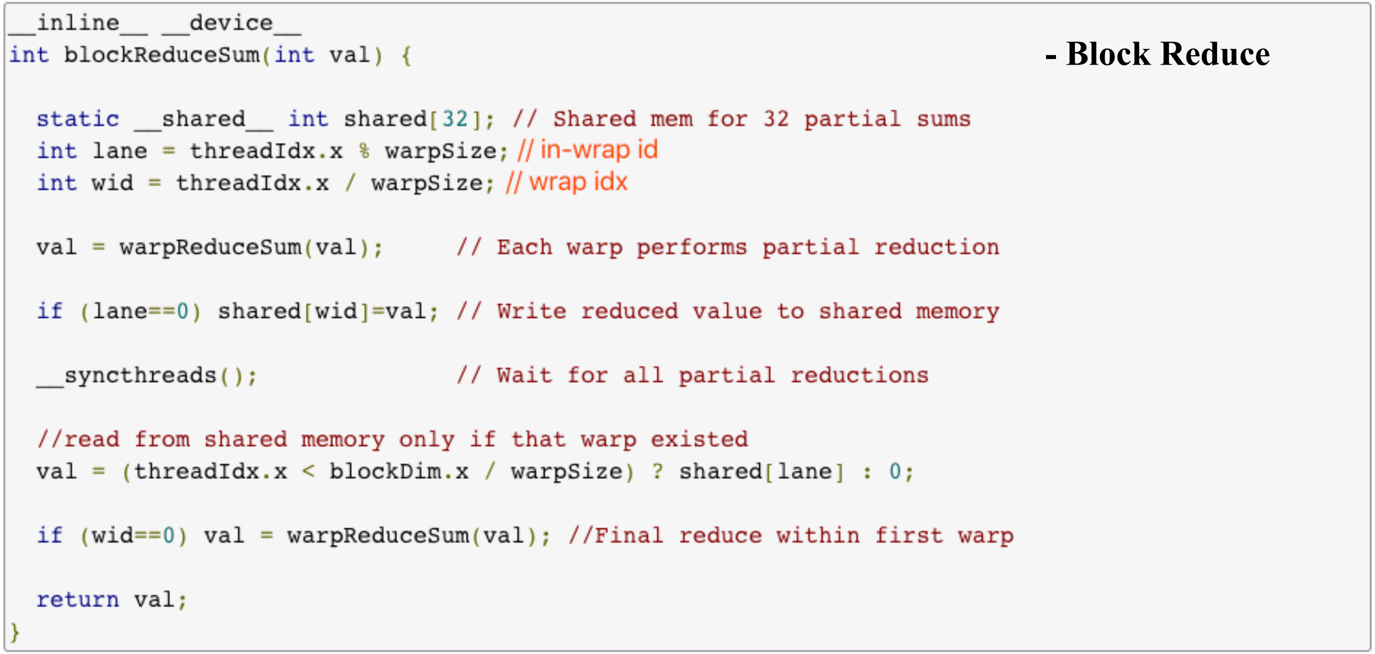
- Shuffle Warp Reduce

```
__inline__ __device__
int warpReduceSum(int val) {
  for (int offset = warpSize/2; offset > 0; offset /= 2)
    val += __shfl_down(val, offset);
  return val;
}
```



CUDA优化：并发算法之Parallel reduction （并行规约）

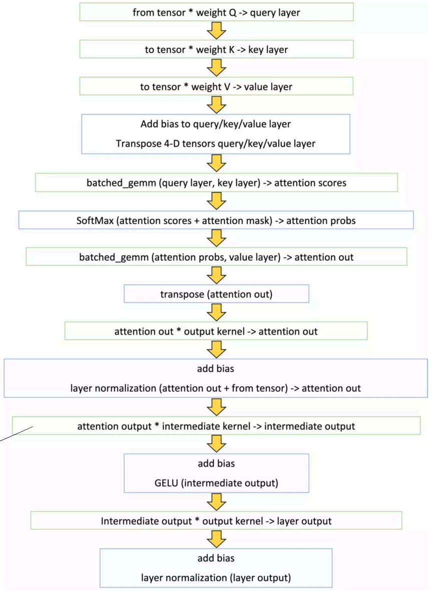
- Block Reduce



```
__inline__ __device__
int blockReduceSum(int val) {

  static __shared__ int shared[32]; // Shared mem for 32 partial sums
  int lane = threadIdx.x % warpSize; // in-wrap id
  int wid = threadIdx.x / warpSize; // wrap idx

  val = warpReduceSum(val);     // Each warp performs partial reduction

  if (lane==0) shared[wid]=val; // Write reduced value to shared memory

  __syncthreads();              // Wait for all partial reductions

  //read from shared memory only if that warp existed
  val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

  if (wid==0) val = warpReduceSum(val); //Final reduce within first warp

  return val;
}
```

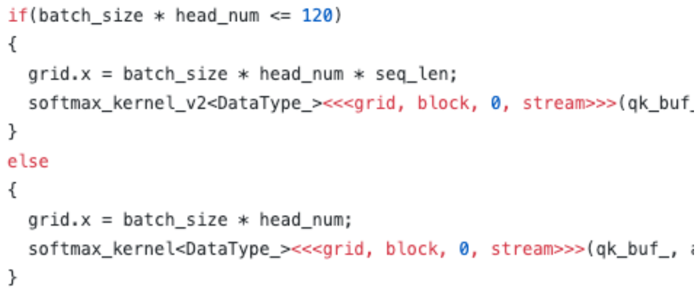
Fastertransformer

1. 为了减少kernel调用次数，将除了矩阵乘法的kernel都尽可能合并 (60 -> 14)
2. 针对大batch单独进行了kernel优化
3. 支持选择最优的矩阵乘法
4. 在使用FP16时使用half2类型，达到half两倍的访存带宽和计算吞吐
5. 优化gelu、softmax、layernorm的实现以及选用rsqrt等


from tensor * weight Q -> query layer
to tensor * weight K -> key layer
to tensor * weight V -> value layer
Add bias to query/key/value layer
Transpose 4-D tensors query/key/value layer
batched_gemm (query layer, key layer) -> attention scores
SoftMax (attention scores + attention mask) -> attention probs
batched_gemm (attention probs, value layer) -> attention out
transpose (attention out)
attention out * output kernel -> attention out
add bias
layer normalization (attention out + from tensor) -> attention out
attention output * intermediate kernel -> intermediate output
add bias
GELU (intermediate output)
Intermediate output * output kernel -> layer output
add bias
layer normalization (layer output)

Fastertransformer

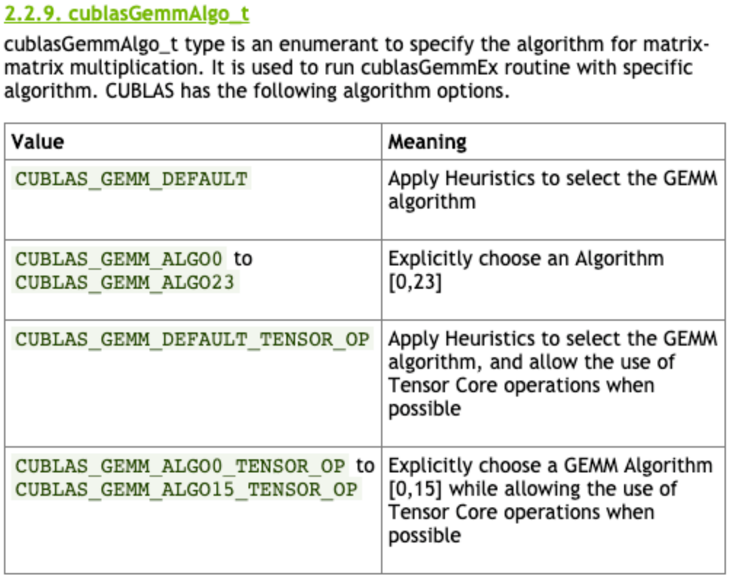
- 针对大batch单独进行了kernel优化



```
if(batch_size * head_num <= 120)
{
  grid.x = batch_size * head_num * seq_len;
  softmax_kernel_v2<DataType_><<<grid, block, 0, stream>>>(qk_buf_
}
else
{
  grid.x = batch_size * head_num;
  softmax_kernel<DataType_><<<grid, block, 0, stream>>>(qk_buf_,
}
```

Fastertransformer

- 支持选择最优的矩阵乘法



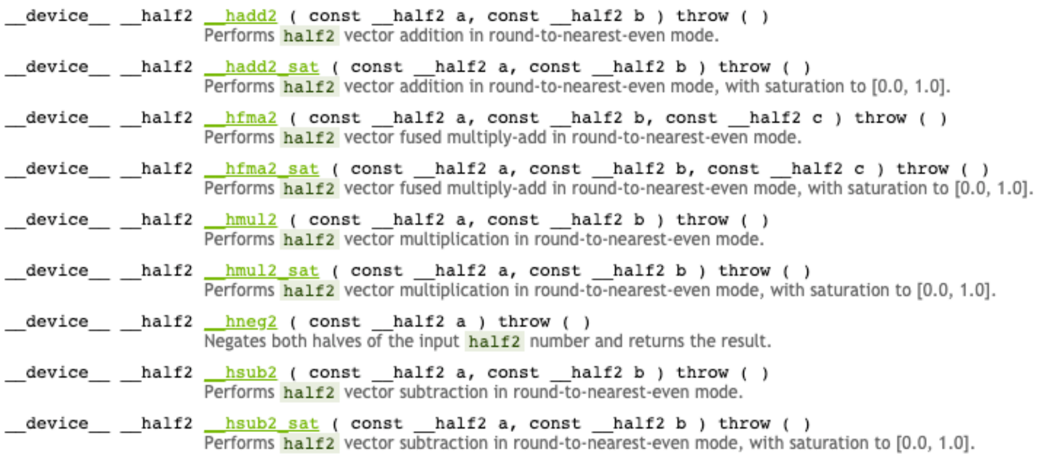
2.2.9. cublasGemmAlgo_t

cublasGemmAlgo_t type is an enumerant to specify the algorithm for matrix-matrix multiplication. It is used to run cublasGemmEx routine with specific algorithm. CUBLAS has the following algorithm options.

Value	Meaning
`CUBLAS_GEMM_DEFAULT`	Apply Heuristics to select the GEMM algorithm
`CUBLAS_GEMM_ALGO0` to `CUBLAS_GEMM_ALGO23`	Explicitly choose an Algorithm [0,23]
`CUBLAS_GEMM_DEFAULT_TENSOR_OP`	Apply Heuristics to select the GEMM algorithm, and allow the use of Tensor Core operations when possible
`CUBLAS_GEMM_ALGO0_TENSOR_OP` to `CUBLAS_GEMM_ALGO15_TENSOR_OP`	Explicitly choose a GEMM Algorithm [0,15] while allowing the use of Tensor Core operations when possible

Fastertransformer

- 在使用FP16时使用half2类型，达到half两倍的访存带宽和计算吞吐



`__device__ __half2 __hadd2 ( const __half2 a, const __half2 b ) throw ( )`
Performs `half2` vector addition in round-to-nearest-even mode.

`__device__ __half2 __hadd2_sat ( const __half2 a, const __half2 b ) throw ( )`
Performs `half2` vector addition in round-to-nearest-even mode, with saturation to [0.0, 1.0].

`__device__ __half2 __hfma2 ( const __half2 a, const __half2 b, const __half2 c ) throw ( )`
Performs `half2` vector fused multiply-add in round-to-nearest-even mode.

`__device__ __half2 __hfma2_sat ( const __half2 a, const __half2 b, const __half2 c ) throw ( )`
Performs `half2` vector fused multiply-add in round-to-nearest-even mode, with saturation to [0.0, 1.0].

`__device__ __half2 __hmul2 ( const __half2 a, const __half2 b ) throw ( )`
Performs `half2` vector multiplication in round-to-nearest-even mode.

`__device__ __half2 __hmul2_sat ( const __half2 a, const __half2 b ) throw ( )`
Performs `half2` vector multiplication in round-to-nearest-even mode, with saturation to [0.0, 1.0].

`__device__ __half2 __hneg2 ( const __half2 a ) throw ( )`
Negates both halves of the input `half2` number and returns the result.

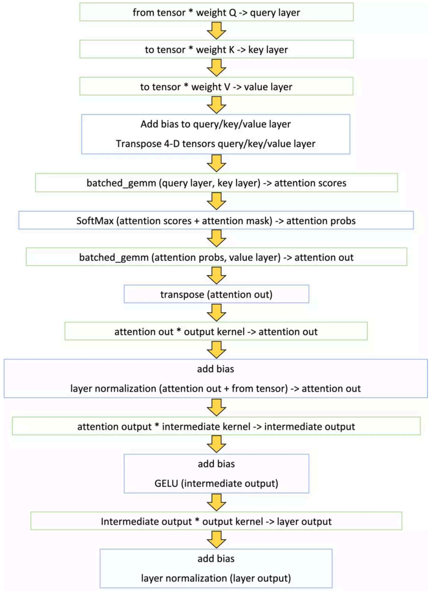
`__device__ __half2 __hsub2 ( const __half2 a, const __half2 b ) throw ( )`
Performs `half2` vector subtraction in round-to-nearest-even mode.

`__device__ __half2 __hsub2_sat ( const __half2 a, const __half2 b ) throw ( )`
Performs `half2` vector subtraction in round-to-nearest-even mode, with saturation to [0.0, 1.0].

Fastertransformer

- 优化gelu、softmax、layernorm的实现以及选用rsqrt等

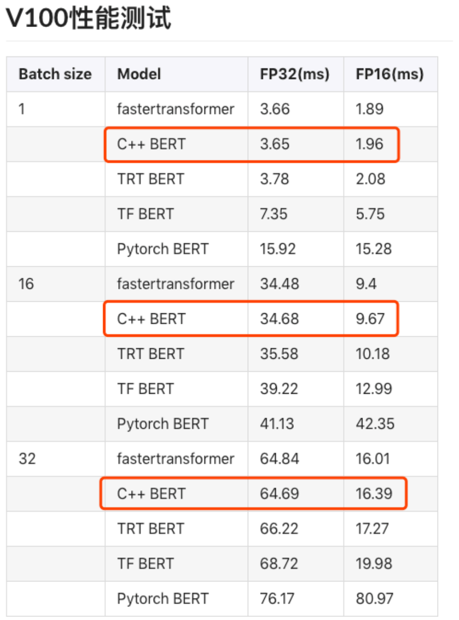
1. 使用Parallel Reduction
2. rsqrt(x) > sqrt(x)



from tensor * weight Q -> query layer
to tensor * weight K -> key layer
to tensor * weight V -> value layer
Add bias to query/key/value layer
Transpose 4-D tensors query/key/value layer
batched_gemm (query layer, key layer) -> attention scores
SoftMax (attention scores + attention mask) -> attention probs
batched_gemm (attention probs, value layer) -> attention out
transpose (attention out)
attention out * output kernel -> attention out
add bias
layer normalization (attention out + from tensor) -> attention out
attention output * intermediate kernel -> intermediate output
add bias
GELU (intermediate output)
Intermediate output * output kernel -> layer output
add bias
layer normalization (layer output)

C++ BERT

1. 合并了3个Embedding operation
2. 合并了Output的线性层和Softmax
3. 采用cuBERT的Tokenizer

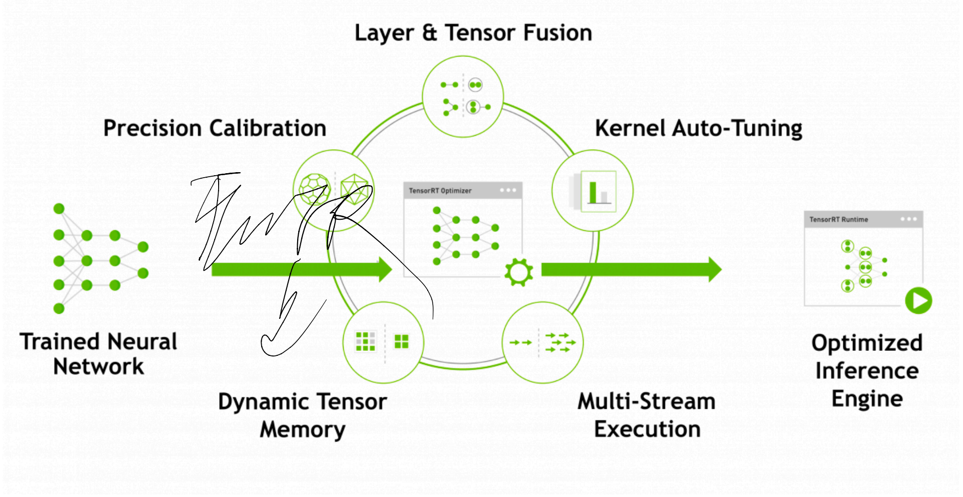


V100性能测试

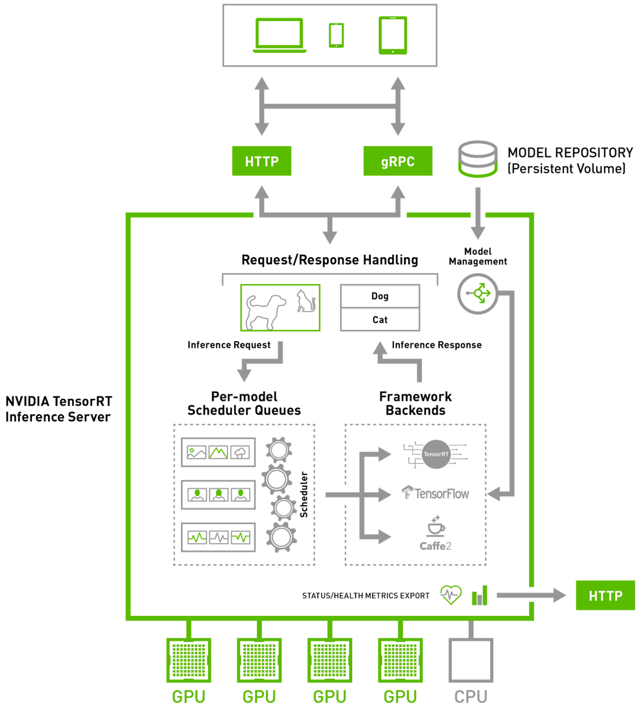
Batch size	Model	FP32(ms)	FP16(ms)
1	fastertransformer	3.66	1.89
	C++ BERT	3.65	1.96
	TRT BERT	3.78	2.08
	TF BERT	7.35	5.75
	Pytorch BERT	15.92	15.28
16	fastertransformer	34.48	9.4
	C++ BERT	34.68	9.67
	TRT BERT	35.58	10.18
	TF BERT	39.22	12.99
	Pytorch BERT	41.13	42.35
32	fastertransformer	64.84	16.01
	C++ BERT	64.69	16.39
	TRT BERT	66.22	17.27
	TF BERT	68.72	19.98
	Pytorch BERT	76.17	80.97

TensorRT

- TensorRT是一个高性能深度学习推理平台，可以对各种框架训练好的模型进行优化

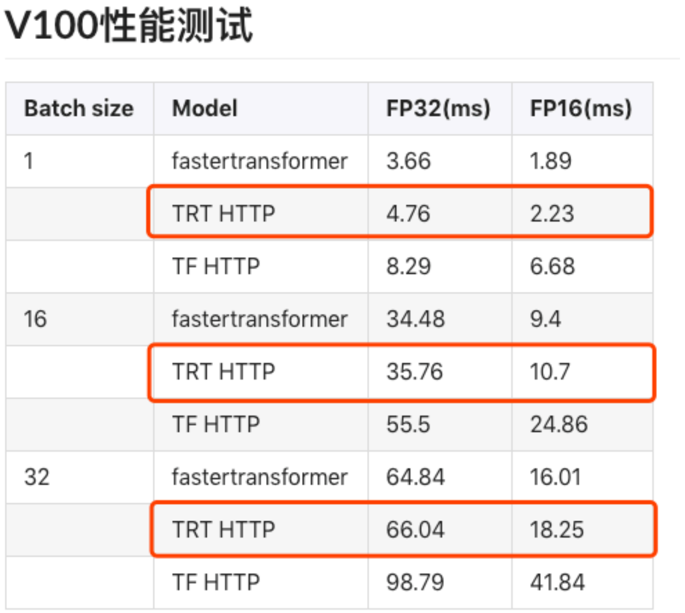


TensorRT Inference Server



1. 支持单GPU上的多模型&单模型多实例
2. 支持多种backends框架
3. 动态Batch增加吞吐
4. 提供负载均衡及状态监测

TensorRT C++ BERT



V100性能测试

Batch size	Model	FP32(ms)	FP16(ms)
1	fastertransformer	3.66	1.89
	TRT HTTP	4.76	2.23
	TF HTTP	8.29	6.68
16	fastertransformer	34.48	9.4
	TRT HTTP	35.76	10.7
	TF HTTP	55.5	24.86
32	fastertransformer	64.84	16.01
	TRT HTTP	66.04	18.25
	TF HTTP	98.79	41.84

直播摘要

1. Fastertransformer可针对线上问答（小batch）和推荐（大batch）场景
2. 优化后的Transformer中矩阵运算占比70%以上
3. 如何在已有项目中应用Fastertransformer：C++的性能最好，没有overhead
4. 如何优化：
    1. tensorflow中的操作被细分成了很多GPU小Kernel，cuda kernel lanuch有overhead。每个kernel都会进行global memory读写（400-800的clock cycle延迟）=>无法充分利用GPU
    2. 除矩阵乘之外都做Kernel fusion
    3. 优化矩阵乘法（cuBLAS最快），根据场景测试最快的矩阵乘算法
    4. 优化自定义Kernel：针对element wish的Kernel，可以进行内存优化，使用寄存器保存中间结果；计算优化（__expf,rsqrt），双下划线的算法都是低精度快速算法
    5. 在Volta和Turing卡可以进行FP16优化（half2），可以降低instraction的数量，提高吞吐
    6. Softmax的不同版本：针对不同业务类型，batch比较小的时候只有12个block，无法隐藏wrap的latency，因此多乘一个seq_len，增加block数量
5. Github issue：DSM是GPU计算能力；FP32的diff在1e-5，F16在0.02左右，如果diff过大可修改GEMM的computeType；
6. Fastertransformer2.0：实现了decoder、beam search、任意batch size和seq_len

重播地址：https://v.youku.com/v_show/id_XNDM3NzMzOTM2MA==.html?spm=a2h0k.11417342.soresults.dtitle