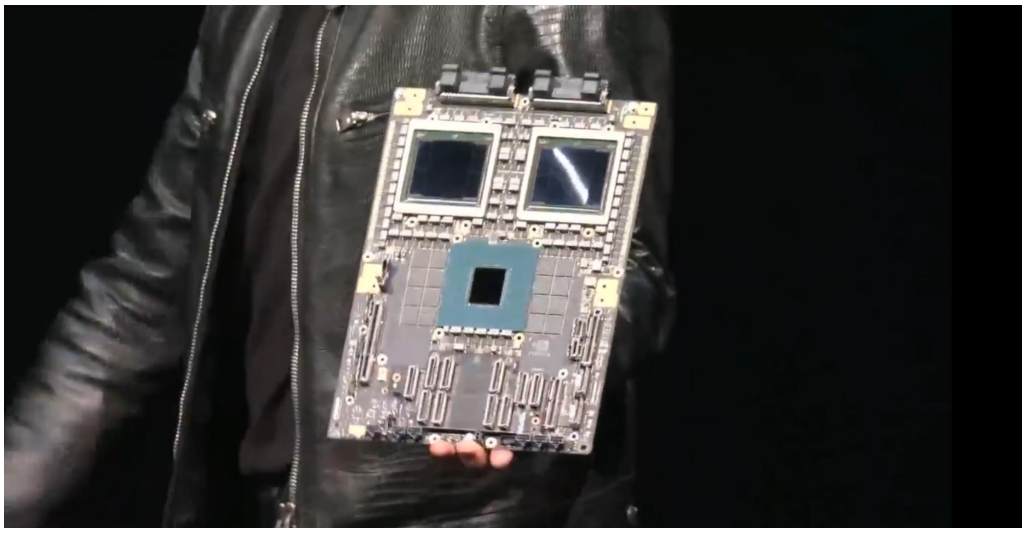
CS205 · C/C++ Programming

Project5 Report:

The beginning of Accelerated Computing

赖海斌
12211612



> “这是一个令人惊叹的时代，因为我们正处于一场新的工业革命的开始，过去蒸汽机、电力、PC 和互联网带来了信息革命，现在是人工智能。前所未有的是，我们正在同时经历两种转变：通用计算的结束和加速计算的开始。”
>
> —— ***Jensen Huang***[1]

摘要：nvcc 会不会对 GPUKernel 做优化？NCCL 是如何让 GPU 互相通信的？复杂的递归 DFS 算法能不能用到并行的 GPU 程序中？怎么样对单个线程进行编程？GPU 如何处理 if 分支语句？谁可能是下一代 BLAS？在本次 Project 中，上面的问题都会得到探讨。

关键词：CUDA；GPU；HPC；AI；

[1] NVIDIA CEO Jensen Huang and H.E. Omar Sultan Al Olama Discuss AI at the World Governments Summit 2024.2.16

目录 Content

Part 1: 为什么需要加速计算

截至 5 月 28 日，NVIDIA 股价突破 1100 美元，Q1 季度营收 260 亿美元[2]，对应市值 2.6 万亿美元，超过 Intel+AMD+Qualcomm+TSMC+ASML+Micron+TI 的总市值。

为什么我们会有 GPU？换句话说，为什么我们需要加速计算？技术诞生的背后是时代的需求。我们回看过去 30 年这家公司在 GPU 应用上的发展，我们可以发现，这是一家面向市场，创造市场的企业，每一代的 GPU 设计在需求基础上形成了独特的时代架构。

在本次 Project 中我们将探讨 NVIDIA GPU 在 SGEMM 中的运行情况与 CUDA 优化策略，同时查看在不同应用场景与价位下 GPU 的架构设计、技术细节与性能表现情况，通过这些学习，对加速计算给一个简单的概览。

[2] https://investor.nvidia.com/financial-info/financial-reports/default.aspx

Part 2: B=aA+b 实现

本次实现建立在 Project4 的实现的矩阵类上，增添了这么几个函数：

1. a*A 函数

GPU Kernel:

```cpp
template<typename T>
__global__ void scalarMatrixMulKernel(const T* A, T* C, int size, T scalar) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < size) {
        C[tid] = scalar * A[tid];
    }
}
```

CPU host:

```cpp
template<typename T>
Matrix<T> operator*(const T& scalar, const Matrix<T>& mat) {
    Matrix<T> result(mat.rows,mat.cols);
    int blockSize = 256;
    int numBlocks = (mat.rows * mat.cols + blockSize - 1) / blockSize;

    scalarMatrixMulKernel<<<numBlocks, blockSize>>>(mat.data, result.data, mat.rows * mat.cols, scalar);
    cudaDeviceSynchronize();

    return result;
}
```

2. A’ + b

GPU:

```cpp
template <typename T>
__global__ void matrixAddWithConstantKernel(T* A,  T* C, int size, T c) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < size) {
        C[tid] = A[tid]  + c;
    }
}
```

CPU host:

```cpp
Matrix<T>& operator+(const  T c) {
    int blockSize = 256;
    int numBlocks = (cols * rows + blockSize - 1) / blockSize;
    matrixAddWithConstantKernel<<<numBlocks, blockSize>>>(data, data, rows * cols, c);
    cudaDeviceSynchronize();
    return *this;
}
```

测试：

```
S12211612@lab01:~/Project/Project4/final$ nvcc main.cu -o p5
S12211612@lab01:~/Project/Project4/final$ ./p5
A:
0 1 2
1 2 3
2 3 4

a:6
b:10
Time for the kernel: 0.003424 ms

a*A+b: finish
time=0.151672

B:
10 16 22
16 22 28
22 28 34

A:
0 1 2
1 2 3
2 3 4
end of main
```

Part 3: cuBLAS 速度对比

1. 时间测量：

CUDA 专门提供了测量时间的 API 函数: cudaEvent_t, cudaEventRecord(), cudaEventElapsedTime() 等函数，具体操作如下：

```
cudaEvent_t start, stop;
    float esp_time_gpu;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);// start

    // your CUDA program

    cudaEventRecord(stop, 0);// stop

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&esp_time_gpu, start, stop);
    printf("Time for the kernel: %f ms\n", esp_time_gpu);
```

在 CPU 情况下，我们依旧使用 project3 中 HPL 的测量方法进行时间测量。

我们首先得出 CPU BLAS SGEMM 的计算时间：

CPU	N
Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz	4096

我们对 MKL 和 OpenBLAS 进行测试：

单位：秒

MKL 单次	MKL 1000 次循环平均值	OpenBLAS
0.118	0.5962	0.114

具体的测试分析我们已经在 Project3 中提到。我们这里是第一次对 MKL 的 JIT 进行测试，看来它还是取得了不错的效果。

我们接下来实现 cuBLAS:

```cpp
Matrix matmul(const Matrix& other) const {
    if (cols != other.rows) {
        throw std::invalid_argument("Matrix dimensions do not match for multiplication");
    }

    int m = rows;
    int n = other.cols;
    int k = cols;

    Matrix result(m, n);

    cublasHandle_t handle;
    cublasCreate(&handle);

    const float alpha = 1.0;
    const float beta = 0.0;

    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, &alpha, data, m, other.data, k, &beta, result.data, m);

    cublasDestroy(handle);
    return result;
}
```

我们在 RTX2080Ti 上进行多次测试，求得 4096*4096 下的 CUDA SGEMM 时间约为 78.06 ms，比 CPU 的 BLAS 有一定的提升。

```cpp
// Create two matrices
Matrix<float> C(rows2, cols2, GPU_device_2);
Matrix<float> D(rows2, cols2, GPU_device_3);

// Initialize matrices A and B
for (float i = 0; i < rows2; ++i) {
    for (float j = 0; j < cols2; ++j) {
        C(i, j) =i+j;
        D(i, j) =i-j;
    }
}

cudaEvent_t start, stop;
float esp_time_gpu;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);// start

// your CUDA program
// 执行矩阵乘法

    Matrix E = C.matmul(D);



cudaEventRecord(stop, 0);// stop

cudaEventSynchronize(stop);

cudaEventElapsedTime(&esp_time_gpu, start, stop);
printf("Time for the kernel: %f ms\n", esp_time_gpu);
```

```
S12211612@lab01:~/Project/Project4/final$ ./p5
A:
0 1 2
1 2 3
2 3 4

a:6
b:10

a*A+b: finish
time=0.155486

B:
10 16 22
16 22 28
22 28 34

A:
0 1 2
1 2 3
2 3 4
Time for the kernel: 79.119553 ms
end of main
```

有趣的是，如果我们这里改进一下，把 Matrix E 这里加一个 for 循环：

```cpp
// your CUDA program
// 执行矩阵乘法

for(int i = 0 ; i< 100 ;i++){
    Matrix E = C.matmul(D);
}
```

按照道理它会耗时 78*100 毫秒，但是实验的结果是，它仅耗时约 1600ms。我认为造成这样的主要原因是缓存仍停留在 SM 中，这使得 CUDA 可以重复利用数据。

诶，那用我们原始的 plain 方法，大概能多少呢？

151ms。

```
Time for the kernel: 151.254181 ms
end of main
```

虽然比 CPU 还好，但是跟 cuBLAS 还是差了一倍。

```cpp
template <typename T>
__global__
void matrixMulKernel(T *a, T *b, T *result, size_t aRows, size_t aCols, size_t bCols) {
    size_t row = blockIdx.y * blockDim.y + threadIdx.y;
    size_t col = blockIdx.x * blockDim.x + threadIdx.x;// 尝试了改变顺序，但是都很大

    if (row < aRows && col < bCols) {
        T sum = 0;
        for (size_t k = 0; k < aCols; ++k) {
            sum += a[row * aCols + k] * b[k * bCols + col];
        }
        result[row * bCols + col] = sum;
    }
}
```

我接下来先干点不干人事的事情：去掉那个 if。我根据我的 CPU，我相信它不会出问题。

```cpp
void multiply(const Matrix& other, Matrix* result_ptr, int type_mul) {
    if (cols != other.rows) {
        std::cerr << "Matrix dimensions do not match for multiplication!\n";
        // Handle error condition here
        return;
    }

    // Launch CUDA kernel for matrix multiplication
    dim3 blockDim(32, 32);
    dim3 gridDim((result_ptr->getCols() + blockDim.x - 1) / blockDim.x, (result_ptr->getRows() + blockDim.y - 1) / blockDim.y);
    matrixMulKernel<<<gridDim, blockDim>>>
            (data, other.data, result_ptr->data, rows, cols, other.cols);
    cudaDeviceSynchronize();
}
```

还真是：

```
Time for the kernel: 133.427872 ms
end of main
```

具体为什么，可以看看后面的论文里 Thread divergence 的内容，因为我是先看论文再过来写的。

然后我们更换一下我们的 blockDim。查看他们所需要的毫秒数。

16*16	32*32	48*48	64*64
137	133	114	144

这些是由于 GPU 的结构引起的。我们接下来由于有一个不错的范本，我们就可以跟着这个范本来研究如何加速 cuBLAS

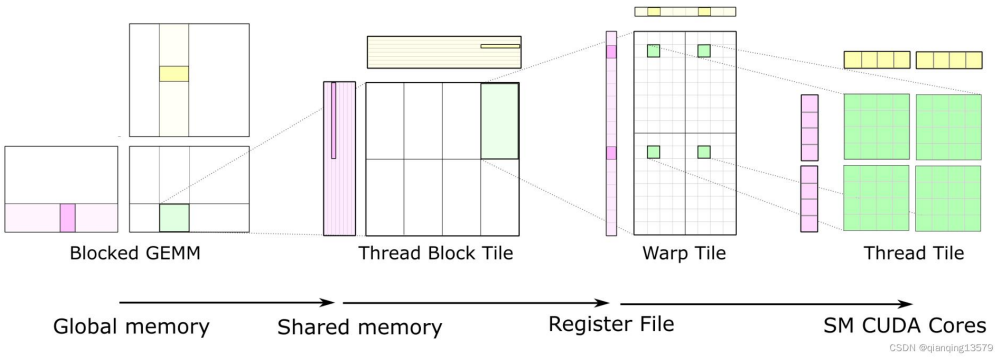
Part 4: 矩阵乘法加速分析

Github 上有个人的 GEMM 算法写的挺好的，我对此进行分析和借鉴：
[cuda_sgemm/gemm.cu at master · njuhope/cuda_sgemm (github.com)]

1. grid 数和 block 数。

```
void OptsGemm(int m, int n, int k, float* d_A, float* d_B, float* d_C,
              float alpha, float beta) {

    constexpr int BLOCK_DIM = 128;
    // subm, subn, subk
    dim3 block(256);
    dim3 grid((m + BLOCK_DIM - 1) / BLOCK_DIM, (n + BLOCK_DIM - 1) / BLOCK_DIM);

    gemm_kernel_NN<<<grid, block>>>(d_A, d_B, (float4*)d_C, alpha, beta, m, n,
                                    k);
}
```

我们将 GEMM 进行分块，每一个 grid 分到 128*128 的小块，每一个 Block 分到 256 的块。然后，我们在核函数中对这 256*256 小块进行细分地操作。



核函数：

2. 循环展开

跟在 CPU 一样，给每一个线程分配多个任务，可以让线程更加地连续，同时线程访问存储的速度也会因为连续而更快。

Reg 的使用是 PTX CUDA 汇编的操作，通过使用寄存器，我们相当于是人肉 NVCC 的 O3 优化，将数据拉到 SIMD 更近的地方进行操作。

#pragma unroll 是 NVCC 编译器独特的辅助编译方式，它可以帮助我们进一步将循环展开分配。

```
#pragma unroll
        for (int j = 0; j < TILE_K - 1; j++)
        {
            reg_a[(j + 1) % 2][0] = smem_a[load_stage_idx][(j + 1) *  TILE_Y_4 + 0 + ty];
            reg_a[(j + 1) % 2][1] = smem_a[load_stage_idx][(j + 1) *  TILE_Y_4 + 16 + ty];
            reg_b[(j + 1) % 2][0] = smem_b[load_stage_idx][(j + 1) *  TILE_X_4 + 0 + tx];
            reg_b[(j + 1) % 2][1] = smem_b[load_stage_idx][(j + 1) *  TILE_X_4 + 16 + tx];
            c[0][0].x += reg_a[j % 2][0].x * reg_b[j % 2][0].x;
            c[0][0].y += reg_a[j % 2][0].x * reg_b[j % 2][0].y;
            c[0][0].z += reg_a[j % 2][0].x * reg_b[j % 2][0].z;
            c[0][0].w += reg_a[j % 2][0].x * reg_b[j % 2][0].w;
            c[0][1].x += reg_a[j % 2][0].x * reg_b[j % 2][1].x;
            c[0][1].y += reg_a[j % 2][0].x * reg_b[j % 2][1].y;
            c[0][1].z += reg_a[j % 2][0].x * reg_b[j % 2][1].z;
            c[0][1].w += reg_a[j % 2][0].x * reg_b[j % 2][1].w;
            c[1][0].x += reg_a[j % 2][0].y * reg_b[j % 2][0].x;
            c[1][0].y += reg_a[j % 2][0].y * reg_b[j % 2][0].y;
            c[1][0].z += reg_a[j % 2][0].y * reg_b[j % 2][0].z;
            c[1][0].w += reg_a[j % 2][0].y * reg_b[j % 2][0].w;
            c[1][1].x += reg_a[j % 2][0].y * reg_b[j % 2][1].x;
            c[1][1].y += reg_a[j % 2][0].y * reg_b[j % 2][1].y;
            c[1][1].z += reg_a[j % 2][0].y * reg_b[j % 2][1].z;
            c[1][1].w += reg_a[j % 2][0].y * reg_b[j % 2][1].w;
            c[2][0].x += reg_a[j % 2][0].z * reg_b[j % 2][0].x;
            c[2][0].y += reg_a[j % 2][0].z * reg_b[j % 2][0].y;
            c[2][0].z += reg_a[j % 2][0].z * reg_b[j % 2][0].z;
            c[2][0].w += reg_a[j % 2][0].z * reg_b[j % 2][0].w;
```

3. share 共享内存

在同一个 warp 内使用共享内存，这样的话计算的时候就可以分配任务，各个线程执行各个矩阵小块的工作。注意到 GPU 内的共享内存其实是非常的宝贵的，我们这里是 smem 并不能设置的太大。

```
__global__ void gemm_kernel_NN(…
{
    __shared__ float4 smem_a[2][TILE_K * TILE_Y_4];
    __shared__ float4 smem_b[2][TILE_K * TILE_X_4];

    int tx = threadIdx.x % 16;
    int ty = threadIdx.x / 16;

    int tx4 = threadIdx.x % 4;
    int ty4 = threadIdx.x / 4;

    int tx32 = threadIdx.x % 32;
    int ty32 = threadIdx.x / 32;

    //! every thread block read TILE_Y rows of A
    //! every 4 thread read a row of A with TILE_K  elements
    //! every thread read 4 elements
    const float* pA = (A + K * TILE_Y * blockIdx.y + ty4 * K + tx4 * 4);
    //! every thread block read TILE_X columns of B
    //! every 32 thread read a row of B with TILE_X elements
    //! every thread read 4 elements
    const float* pB = (B + TILE_X * blockIdx.x + ty32 * N + tx32 * 4);

    //! every thread block write TILE_Y/4 rows of C, TILE_X_4 * 4(float4)
    //! columns of C
    float4* pC = C + TILE_Y * blockIdx.y * N / 4 + TILE_X_4 * blockIdx.x;

```

4. 连续数据

我们将共享内存中的数据导入到各自的 reg 中，这样可以提升我们的乘法计算速度。

```
376         //! load data from shared memory to register for the next TILE_K
377         //! iteration
378         reg_a[0][0] = smem_a[load_stage_idx ^ 1][0 + ty];
379         reg_a[0][1] = smem_a[load_stage_idx ^ 1][16 + ty];
380         reg_b[0][0] = smem_b[load_stage_idx ^ 1][0 + tx];
381         reg_b[0][1] = smem_b[load_stage_idx ^ 1][16 + tx];
382 
383         //! compute the last TILE_K-1 iteration, the register data is load ahead
384         c[0][0].x += reg_a[1][0].x * reg_b[1][0].x;
385         c[0][0].y += reg_a[1][0].x * reg_b[1][0].y;
386         c[0][0].z += reg_a[1][0].x * reg_b[1][0].z;
387         c[0][0].w += reg_a[1][0].x * reg_b[1][0].w;
388         c[0][1].x += reg_a[1][0].x * reg_b[1][1].x;
389         c[0][1].y += reg_a[1][0].x * reg_b[1][1].y;
390         c[0][1].z += reg_a[1][0].x * reg_b[1][1].z;
391         c[0][1].w += reg_a[1][0].x * reg_b[1][1].w;
392         c[1][0].x += reg_a[1][0].y * reg_b[1][0].x;
393         c[1][0].y += reg_a[1][0].y * reg_b[1][0].y;
394         c[1][0].z += reg_a[1][0].y * reg_b[1][0].z;
395         c[1][0].w += reg_a[1][0].y * reg_b[1][0].w;
396         c[1][1].x += reg_a[1][0].y * reg_b[1][1].x;
397         c[1][1].y += reg_a[1][0].y * reg_b[1][1].y;
```

在这样之后，我们的 gemm 在访存上就被很大地优化了。同时，我们的线程分配也让我们的计算效率基本上最大化。

我这里统计了一下身边可以计算的卡。然后对他们进行一一的测试。但是由于时间限制，我没有测算我们的自己的 cuBLAS 时间。

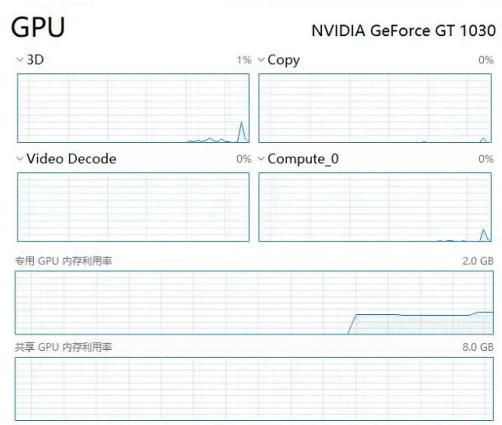
名称	类型	架构	发布日期	CUDA 支持版本	功耗	价格
A100 80G	AI、HPC 数据中心	Ampere	June 26th, 2021	12.4	300W	170000RMB
A100 40G	AI、HPC 数据中心	Ampere	May 15th, 2020	12.4	300W	84000RMB
V100	AI、HPC 数据中心	Volta	May 11th, 2017	12.1		43000RMB
TITAN RTX	图形学、深度学习	Turing	Dec 3rd, 2018	12.3		6000RMB
RTX 2080Ti	游戏	Turing	Oct 8th, 2018	12.4	250W	2600RMB
RTX 3060 Laptop	游戏	Ampere	Feb 2nd, 2021	12.4	<100W	2200RMB
GT 1030	游戏	Pascal	May 17th, 2017	9.1	100W	248RMB
Quadro P2000	图形学	Fermi	Dec 24th, 2010	2.1	80W	93RMB

GPU	cuBLAS 花费时间
RTX 3060 Laptop	150.32
A100 80G	18.09
A100 40G	46.15
V100	41.44
RTX 2080Ti	79.80
Quadro P2000	3603.50
GT 1030	2353.65

```
#!/bin/bash
#BSUB -q 4a100-40 ##队列名
#BSUB -n 4 ##申请的 CPU 总核数
#BSUB -e %J.err
#BSUB -o %J.out
#BSUB -R "span[ptile=4]" ##每个 host 上的 CPU 核数
#BSUB -gpu "num=1/host"
##"num=4/host"参数的含义是每个 host 申请 4 张 GPU 卡，
每
#个 host 上分配 4 张 GPU 卡;也可以使用"num=1/task"参数,
了 1 个
hostfile=`echo $LSB_DJOB_HOSTFILE`
NP=`cat $hostfile | wc -l`
module load cuda/11.8

nvcc he.cu -o He -lcublas

./He >> back.txt
```

Quadro 2000，淘宝 87 块钱的高级货，搭配 E5-2666 v3 神教

GT1030，入门级显卡：



本次的 Project，我想把时间更投入到对 GPU 的研究进展当中，所以我就没有再进行 GEMM 的具体计算和探究。

Part 5: 论文赏析

没活了，给大伙表演一个论文解读吧。通过几篇论文，我们就能对 GPU 研究领域中的一个方向有所了解。接下来这篇会议论文是：GPU 上 MBE 算法的加速。我们将看到 GPU 研究中的一个方向：**复杂算法加速**。

Efficient Maximal Biclique Enumeration on GPUs

Zhe Pan
Zhejiang University
Hangzhou, China
panzhe@zju.edu.cn

Shuibing He*
Zhejiang University
Hangzhou, China
heshuibing@zju.edu.cn

Xu Li
Zhejiang University
Hangzhou, China
fhxu@zju.edu.cn

Xuechen Zhang
Washington State University
Vancouver
Vancouver WA, USA
xuechen.zhang@wsu.edu

Rui Wang
Zhejiang University
Hangzhou, China
rwang21@zju.edu.cn

Gang Chen
Zhejiang University
Hangzhou, China
cg@zju.edu.cn

摘要介绍：

MBE 是当今很重要的算法，但是基本都是在 CPU 上实现。这篇文章里，我们将 MBE 算法引入到 GPU。但是这里边我们遇到了 3 个主要问题：

1. **内存不足**。MBE 图规模大，对内存需求高，GPU 内存不够；
2. **线程分歧多**。MBE 中有很多 if 分支，GPU 在分支上会遇到 Thread divergence，GPU 的运算效率会下降；
3. **负载不均衡**。MBE 中不同的搜索由于图形状不同，运算所需时间也不一样，导致有的线程跑的时间长，有的短。

针对这三个问题，本文提出了对应的 3 个方法：

1. 设计 node reuse approach 降低内存用量。
2. 使用 pro-active method 剪枝，利用节点邻居数，减少线程分歧。
3. 设计 load-aware 任务调度框架实现 GPU warps 内部线程负载均衡。

我们的效果：在 A100 上运行速度比 96 核服务器快 70.6 倍。

Maximal biclique enumeration (MBE) in bipartite graphs is an important problem in data mining with many real-world applications. All existing solutions for MBE are designed for CPUs. Parallel MBE algorithms for GPUs are needed for MBE acceleration leveraging its many computing cores. However, enumerating maximal bicliques using GPUs has three main challenges including large memory requirement, thread divergence, and load imbalance. In this paper, we propose GMBE, the first highly-efficient GPU solution for the MBE problem. To overcome the challenges, we design a node-reuse approach to reduce GPU memory usage, a pro-active pruning method using the vertex's local neighborhood size to alleviate thread divergence, and a load-aware task scheduling framework to achieve load balance among threads within GPU warps and blocks. Our experimental results show that GMBE on an NVIDIA A100 GPU can achieve 70.6× speedup over the state-of-the-art parallel MBE algorithm ParMBE on a 96-core CPU machine.

CCS CONCEPTS

• **Mathematics of computing** → **Graph enumeration**; • **Computer systems organization** → **Multicore architectures**; • **Information systems** → *Data mining*.

KEYWORDS

Maximal biclique enumeration, GPU

什么是 MBE 算法？

MBE 算法用于在给定的图 G(V,E) 中找到所有的最大完全二部子图 maximal bicliques。在离散数学我们学到，二分图就是把一个图的顶点划分为两个不相交子集 ，使得每一条边都分别连接两个集合中的顶点，且集合内顶点不临接。在一个图中寻找最大（点/边最多）完全二部子图是在生物信息学、数据挖掘等应用中的重要算法。

为什么用 GPU？

CPU 方面我们已经有很好的 MBE 算法了，但是随着这几年

1 INTRODUCTION

A *bipartite graph* $G = (U, V, E)$ contains two disjoint vertex sets U and V, where an edge $e \in E$ only occurs between two vertices in U and V, respectively. A *biclique* is a complete bipartite graph, i.e., there exists an edge between two vertices if and only if the two vertices are in different vertex sets. A *maximal biclique* in G is a subgraph of G, which is a biclique and can not be further enlarged to form a larger biclique. Maximal biclique enumeration (MBE) aims to find all maximal bicliques in G.

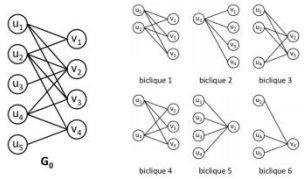


Figure 1: A bipartite graph G_0 containing 6 maximal bicliques.

despread applications, such as anomaly detection in e-commerce networks [30, 35], social recommendation in social networks [29], gene expression analysis in expression datasets [34, 39], and GNN information aggregation [38]. Consider an example in an e-commerce network, where the purchasing relationships can be modeled by a bipartite graph. It is suspicious for a large group of customers to buy a set of products together because online sellers are likely to make fake purchases through illegal platforms to improve their credibility and positive ratings [10, 30, 35]. We can identify all these suspicious groups by enumerating all the maximal bicliques in the network, and then detect them.

在深度学习、数据挖掘等在 GPU 上的应用的更多，大家开始希望有 GPU 上的 MBE 算法。同时，MBE 在社交媒体推荐、基因表达、GNN 信息聚合中应用的更多了，但是这些应用的数据量比以前更大，占用内存更多，CPU 的计算速度开始逐渐跟不上了。

以往有人研究过这个方面吗？有什么缺点？

有的，最早期的 MBE 设计出来后，人们就发现计算开销非常高。于是有人从算法设计上开始用剪枝等操作降低开销，但是他们都是在单核上跑的。后来有人设计了多核的，但是还是在 CPU 上。所以我们需要一个在 GPU 上的。（另外是，也有人设计了 GPU 上 MBE 算法，比如这篇文章[3]，但是他们加速比最高只有 38，那他们为什么没有我们快呢？原因就在下面）

Over the past few decades, many MBE algorithms have been proposed to speed up the enumeration of all maximal bicliques in bipartite graphs [8, 9, 14, 18, 22, 29, 32, 39]. A mainstream approach is to use the set enumeration tree [33] to recursively enumerate all candidate subgraphs, and then judge whether they are maximal bicliques. The enumeration space of this method is a powerset of V or U, so the computational overhead is very high [19], especially for large graphs. Therefore, many efforts are made to reduce the enumeration space using pruning [8, 14] and vertex ordering [14, 39]. However, they only achieve limited speedup for not exploring the parallelism of multi-core CPUs. Other works design parallelization strategies for the multi-core CPUs to speed up the enumeration process [18]. However, their performance speedup is still constrained by the limited parallelism of CPUs. For instance, the state-of-the-art parallel MBE algorithm ParMBE [18] costs over 40 minutes to finish running MBE on a medium-scale bipartite graph Github [25] on a 96-core CPU machine. In contrast, our GPU-based solution can reduce the running time to 132 seconds on an NVIDIA A100 GPU [4] because GPUs offer much higher computational throughput and parallelism than CPUs.

我们的三个困难具体是怎么样的？我们如何解决？

首先是内存问题。直接从 CPU 搬运到 GPU 的算法需要大内存空间，同时它需要经常**分配内存**，这种动态内存分配对 GPU 运算来说也是一种性能损耗。

第二是现有 MBE 算法存在 irregular computation 问题。我们知道假如 GPU 同时执行 1000 条加法指令，它会干的很好。但是 MBE 问题会要求 GPU 在同一个 Warp 内的不同线程都执行不一样的路径。而这会导致动态分支。

这里我简单介绍一下 thread divergence。我们的 CPU 是零售的话，我们 GPU 就是批发。而批发导致我们无法对个别线程很好地处理。比如我们在一个 warp 中有 if(sum[x] < 100){ ... }，如果我们是 CPU，在计组上我们学过，我们会根据结果 flush 掉后面刚刚 fetch 的指令，或者运用分支预测技术选好可能会走下去的分支。

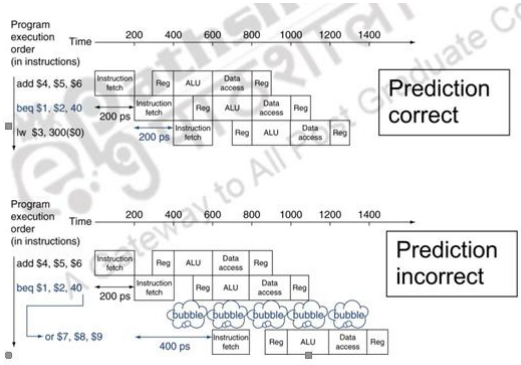


Figure 14.1

但是我们 GPU 如果在 warp 当中，某些数据选择肯定条件，某些选择否定条件，因为我们使用的是 SIMD 模块进行操作，做法跟 CPU 完全不同。

There are three major challenges in the design of a highly efficient MBE algorithm for GPUs. First, the existing MBE algorithms require large memory space and frequent memory allocations to store the intermediate data of each enumerated subgraph, thus they cannot efficiently run on GPUs with limited memory capacity and high dynamic memory allocation overhead [37], due to the severe shortage of memory resources.

Second, the performance of existing MBE algorithms suffers from irregular computation [13] while GPUs are suitable to perform regular computation with high parallelism. Specifically, different GPU threads in the same warp in the MBE algorithm may take different execution paths to access different vertex neighbors, causing the thread divergence problem [15]. GPUs will serialize these diverged thread operations [1], resulting in low thread utilization and poor memory access efficiency.

Lastly, existing MBE algorithms have a serious load imbalance problem on GPUs. The reason is that the maximal biclique sizes are varied significantly with real-world graphs for the power-law distribution of vertex degrees. As a result, threads assigned to each GPU core have different running times. When load imbalance happens, thousands of GPU threads have to wait for the slowest one to complete, causing degraded GPU performance.

Many existing studies have used GPUs to improve the performance of other graph enumeration problems, like maximal clique enumeration [36] and graph pattern mining [15]. The optimizations include data graph partitioning [23], two-level parallelism [17], adaptive buffering [16], hybrid order on GPU [15], etc. However, none of them is efficient for MBE using GPUs. This is because the enumerated subgraphs for MBE generally contain much more vertices than those in other graph enumeration problems. For instance, it may enumerate maximal bicliques comprising several to thousands of vertices, while the triangle counting algorithm only considers subgraphs with three vertices. The larger subgraphs generated at runtime require larger memory and computation costs and lead to more serious problems of large memory requirement, thread divergence, and load imbalance mentioned above.

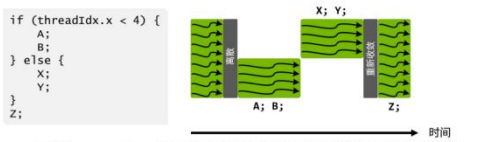
To address all the challenges and achieve highly efficient maximal biclique enumeration on GPUs, we design the first GPU-based MBE algorithm (GMBE) considering the characteristics of GPU architecture and MBE computation pattern. Specifically, first, we replace the existing recursion with a stack-based iteration and reuse the memory of the root node throughout the iteration process. This approach effectively reduces memory usage because it eliminates the need to allocate additional memory space for new nodes. Second, we use the local neighborhood size of a vertex to reduce the enumeration space. The new pruning approach can reduce the number of sub-trees without visiting the nodes. Thus, it can significantly reduce thread divergence. Finally, GMBE carefully manages the size of subtrees assigned to GPU threads and schedules the tasks using two-level queues to achieve load balance within GPU warps and blocks leveraging persistent thread programming models for GPUs.

We adopt the fast CUDA primitives [28] to implement the GMBE prototype and conduct extensive experiments on real-world datasets to demonstrate the efficiency of GMBE. Our experimental results show that GMBE on an NVIDIA A100 GPU can achieve up to 70.6× speedup over the state-of-the-art parallel MBE algorithm ParMBE on a 96-core CPU machine.

[3] Accelerating Maximal Biclique Enumeration on GPUs https://arxiv.org/pdf/2401.05039

NVIDIA 早期 GPU SIMT 模型

Pascal 和早期 NVIDIA GPU 均以 SIMT（单指令多线程）形式执行含 32 个线程的线程组（称为“线程束”）。Pascal 线程束使用所有 32 个线程中共享的单一程序计数器，并结合激活掩码，该掩码可指定某些线程束在特定时间内处于激活状态。这意味着离散的执行路径会让某些线程处于非激活状态，并序列化执行线程束的不同部分，如图 20 所示。原始掩码会存储起来直到线程束重新收敛（通常在离散部分末端），此时掩码恢复，线程再次一起运行。

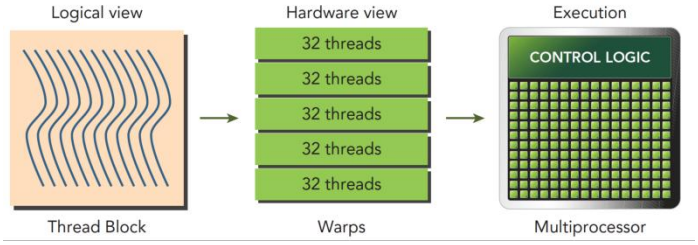


Pascal 和早期 NVIDIA GPU 的 SIMT 线程束执行模型中的线程调度。大写字母代表以程序伪代码编写的语句。线程束中离散的分支经过序列化，因此分支一侧的所有语句一起执行直至完成，然后另一侧的语句才会执行。执行 else 语句后，线程束的线程通常将重新收敛。

在编程层面上，我们常常在核函数中写下这么一句话：

```
1.      int tid = blockIdx.x* blockDim.x + threadIdx.x;
```

但是，我们的 tid 其实是一种逻辑化的东西，我们认为线程都是独立连续的。但是在物理实现上，GPU 内的线程是以 32 个线程为一组，编成一个线程束 warp。而 GPU 正是由这些线程束统一构成，每一个束基本上是执行命令的一个小单元。

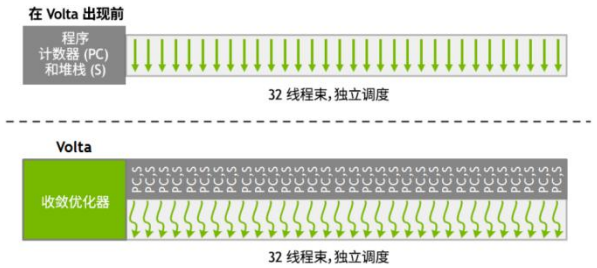


在早期 Pascal 架构中，面对 if 分支指令，一个 warp 会先跑一遍 A,B，随后再跑另一边 else 的 X，Y，所用时间是他们的总和，效果非常的铸币。为什么呢？因为一个 warp 只有一个 PC，这个 PC 只会一次 fetch 一条指令，才导致了这个后果。

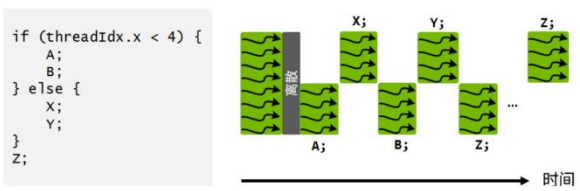
在 Volta 架构开始，线程开始变得精细化，每个线程都有一个自己的 PC，这代表着每个线程可以执行自己想执行的指令了。

Volta SIMT 模型

Volta 通过在所有线程之间实现等效并发（无论线程束为何），成功转变了这一格局。这一点全赖保持每个线程的执行状态（包括程序计数器和调用栈）而实现，如图 21 所示。



相较于 Pascal 和早期架构（顶部），Volta（底部）的独立线程调度架构块状图。Volta 会维持每个线程的调度资源，如程序计数器 (PC) 和调用栈 (S)，而早期架构是按每个线程束来维持这些资源。



通过 Volta 的独立线程调度功能，可交错执行离散分支中的语句。这可帮助执行精细的并行算法，使线程束中的线程实现同步和通信。

虽然这听起来就像南科大任选课程任选专业一样非常的自由，但是在实际执行中我们仍然不能有部分线程执行 A，同时有的线程执行 X。主要原因是害怕 A 和 X 会互相影响。它这么做的目的是可以精细化管理线程。它的力量我们会在后面发现。

不过你这时可能要问了：诶，那这样我们好像执行的时间还比之前早期的用时还长啊！原本时间是 A+B+X+Y+Z，现在变成 A+X+B+Y+Z+Z 了。这里要注意的是，GPU 在执行时有访存时间约束。

这就提到 GPU 的另一个技术：**throughput**。

假设我们现在的任务是光栅渲染。现在，我们使用 GPU 中的 warp 来渲染。我们将取出数据，EX，然后 MEM，这里边，我们会花很多时间进行访存，然后再存储。此时 GPU 的 ALU 不就没事干了，但是闲着也是闲着，GPU 就会像上下文切换一样，换一批程序进行处理。这样，当比如说我们的着色器程序遇到内存读取操作时，如访问纹理，因为 32 个 threads 执行的是相同的程序，所以它们会同时遇到该操作，内存读取意味着该线程组将会阻塞（stall），全部等待内存读取的结果。为了降低延迟，GPU 的 warp 调度器会将当前阻塞的 warp 换出，用另一组包含 32 个线程的 warp 来代替执行。换出操作跟单核的任务调度一样的快，因为在换入换出时，每个线程的数据都没有被触碰，每个线程都有它自己的寄存器，每个 warp 都负责记录它执行到了哪条指令。换入一个新的 warp，不过是将 GPU 的 shader cores 切换到另一组线程上继续执行，除此之外没有其他额外的开销。[4]通过这样精细化线程，我们发现我们可以更好地利用好访存的时间进行别的运算，降低运算总体时间。但是，在这种技术下的运行时间仍然是大于直接运算的时间的。

因此，分支是并行计算里一个比较可怕的事情。因为分支，本来可以一起 add 的指令，现在只能你 add 我 sub，这样的性能下降是我们在把一般的算法用到 CUDA 中必须要考虑的问题。

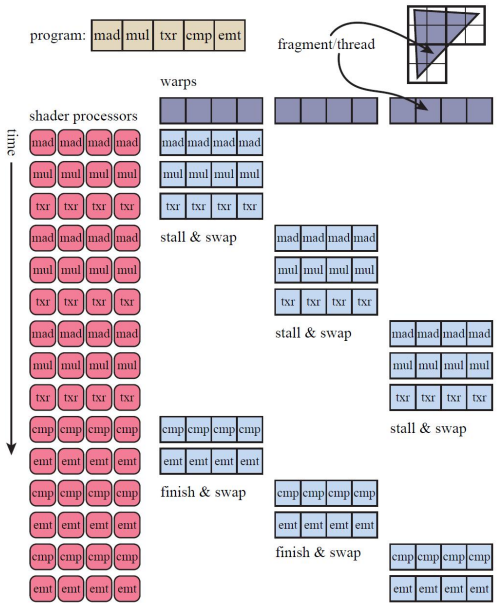
第三个问题是负载均衡问题。可能有的子图非常大，给一个线程算就太慢了。而 GPU 每次就必须等最慢的那个子图算完才能输出，此时其他线程都在围观，这是非常难受的。

以往的 GPU 在图论方面的算法有 maximal clique enumeration, graph pattern mining，优化方法有 data graph partitioning, two-level parallelism, adaptive buffering, hybird order。但是他们都只适用于小点数的图，面对大点数的 MBE 无能为力。

但是我们就比较牛。第一，面对内存问题，我们将递归变回 for+栈（递归本质），同时重复利用 root node 的 Memory 而不是自己 copy 新的一份。第二面对分支问题，我们使用 local neighborhood size 来减少搜索空间，从而减少 if 的次数。第三面对负载问题，我们会根据 subtree 的具体大小来动态分配线程任务。

我们用的操作是使用 fast CUDA primitives 线程束级原语来实现我们的 GPU 上 MBE（后称 GMBE）算法。"线程束级原语" 指的是 CUDA 中针对线程束（thread warp）级别的操作或功能。线程束是 CUDA 中最小的并行执行单元，通常包含 32 个线程。线程束级原语允许开发者直接操作线程束，以实现更细粒度的控制和优化。

我们拿下面这个例子进行介绍，在一个带选择分支的语句中，我们先算出这个线程的 tid（thread id），然后根据线程 id 进行判断。但是正如我们刚才所说，这会导致一个 warp 内 32 个线程，有 16 个走左边，16 个走右



[4] https://www.cnblogs.com/hellobb/p/10695915.html

边的情况发生。而我们希望的是，一个 warp 32 线程走左边，另一个 warp 32 线程走右边。

```
__global__ void mathKernel1(float *c)
{
 int tid = blockIdx.x* blockDim.x + threadIdx.x;

 float a = 0.0;
 float b = 0.0;
 if (tid % 2 == 0)
 {
  a = 100.0f;
 }
 else
 {
  b = 200.0f;
 }
 c[tid] = a + b;
}
```

那么我们就可以这样写我们的代码：在判断时加上 warpSize，第一个线程束内的线程编号 tid 从 0 到 31，tid/warpSize 都等于 0，那么就都执行 if 语句。第二个线程束内的线程编号 tid 从 32 到 63，tid/warpSize 都等于 1，执行 else 语句。

这样，我们的线程束内就没有了分支，提高了我们的效率[5]。 线程束级原语的设计是目前学界感兴趣的一项研究项目。文章[7]探讨了不同的线程束级原语的设计及对效率的影响。文章[8]使用了原语来实施最小生成树算法（如右图）。文章[9]利用了原语中的 Scan（类似于 MPI Scan，将各个线程的结果发至一个线程），实现了 GPU 上的高效快速排序和稀疏矩阵乘法，并将其应用到图形浅水流体模拟上。文章[16]则是一篇古老的文章，它是最早将渲染算法用更精细的原语实现的文章。

原语中有很多精细的操作。比如当 warp 中的线程需要执行比数据交换原语所提供的更复杂的通信或选择操作时，可以使用该__syncwarp()原语来同步 warp 中的线程。它类似于__syncthreads()原语（同步线程块中的所有线程），但粒度更细。

void __syncwarp(unsigned mask=FULL_MASK);

__syncwarp()原语使正在执行的线程等待，直到指定的所有线程 mask 都已执行了__syncwarp()（具有相同的 mask），然后才恢复执行。它还提供了一个 内存屏障， 以允许线程在调用原语之前和之后通过内存进行通信。

每个“同步数据交换”原语在线程束中的一组线程之间执行集合操作。例如，表 2 显示了其中的三个。每个线程在同一 warp 中从一个线程调用__shfl_sync()或 __shfl_down_sync()接收数据，并且每个调用的线程都__ballot_sync()接收一个位掩码，该掩码表示 warp 中所有传递谓词参数真值的线程。

int __shfl_sync(unsigned mask, int val, int src_line, int width=warpSize);

int __shfl_down_sync(unsigned mask, int var, unsigned detla,
int width=warpSize);

int __ballot_sync(unsigned mask, int predicate);

---

[5]https://face2ai.com/CUDA-F-3-2-%E7%90%86%E8%A7%A3%E7%BA%BF%E7%A8%8B%E6%9D%9F%E6%89%A7%E8%A1%8C%E7%9A%84%E6%9C%AC%E8%B4%A8-P1/

参与调用每个原语的线程集使用 32 位掩码指定，这是这些原语的第一个参数。必须使所有参与线程同步，以使选择操作正常工作。因此，如果这些原语尚未同步，则它们首先将同步线程。

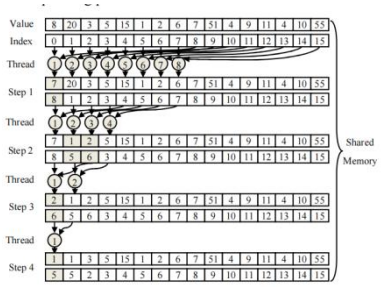


Figure 3. Shared memory search

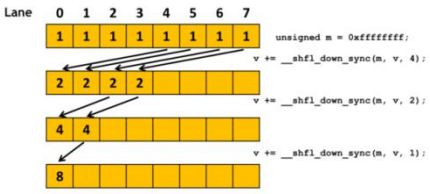
```
__global__ void mathKernel2(float *c)
{
 int tid = blockIdx.x* blockDim.x + threadIdx.x;
 float a = 0.0;
 float b = 0.0;
 if ((tid/warpSize) % 2 == 0)
 {
  a = 100.0f;
 }
 else
 {
  b = 200.0f;
 }
 c[tid] = a + b;
}
```

另外，我也看到可以使用下面的命令查看 CUDA 运行时的 branch efficiency：

```
nvprof --metrics branch_efficiency ./divergence
```

这个 branch efficiency 是由下面这个计算公式得来的：

$$\text{Branch Efficiency} = \frac{\text{Branches} - \text{DivergentBranches}}{\text{Branches}}$$



有趣的是，文章中还提到，这种分支效率有时候会被 nvcc 编译器优化改进。也就是说，我们的 nvcc 还是会做一些优化的，只不过不是 CPU 上的优化，而是在 GPU 上的。

接下来就是论文里的具体操作了。原始的方法是使用递归进行操作，然后每一个线程负责不同的节点底下不同的分支，在 DFS 下找到新的节点，判断构建的二部图是否可以继续扩大。

那么我们首先进行一下内存的分析。首先按照这样的算法的操作，我们根据测试案例，给到每一个 SM 的子图大概需要 3.67GB。而如果想利用上 A100 内的 108 个 SM，我们需要 397GB。

那么我们自然会想到，诶既然你给每一个 SM 都来一个子图，你为什么不造一个更大的图，然后几个 SM 一起共用，这样子不就节省了一些内存吗？确实，他们就是这么做的。请看算法。

如图所示，将递归更改成栈之后，我们的 node buffer 就可以继续利用下去，从而使得 GPU 内的资源就可以重复利用。虽然这样存储单个子树的大小提高，但是整体的大小变小。在选用的数据集上，因为每个过程只需要（3×13,601 + 2×53,915）×sizeof(int) = 595 KB。与 3.1 节中讨论的简单实现需要 13,601×（13,601+53,915）×sizeof(int)= 3.67 GB 相比，这种节点重用方法极大节省了内存空间。

下一个优化是剪枝。由于这里跟文章的算法紧密相关，我就简单介绍一下。我们这个 GMBE 算法里最大的 if 分支是右图的第 10-13 行。而在这之中作者通过证明发现，我们将一个顶点 v∈V 的 local neighbors 定义为 v 连接的其他节点。比如图 5 中的 v4，连着 u4,u5 两个点，local neighbors 数量为 2。V3 连着 u1,u2,u4，数量为 3。接着，我们发现如果跳出遍历子节点到别的节点，它们当中的点的 local neighbors 大小不改变，我们就可以不用去检验大小不改变的节点。这样的剪枝方法具有较低的线程散度，因为不同的线程总是检查同一候选集中的元素。

Algorithm 1: Recursive MBE Algorithm

```
Data: Bipartite graph G(U, V, E)
Input: Set L ⊆ U, disjoint sets R, C ⊆ V
Output: All maximal bicliques
1  procedure recursively_search(L, R, C):
2      foreach v' ∈ C do
3          L' ← L ∩ N(v'); R' ← R; C' ← ∅;
4          foreach v_c ∈ C do                    // Node generation
5              if L' ∩ N(v_c) == L' then
6                  R' ← R' ∪ {v_c};
7              else if L' ∩ N(v_c) ≠ ∅ then
8                  C' ← C' ∪ {v_c};
9          if R' == Γ(L') then                   // Maximality check
10             Output(L', R') as a maximal biclique;
11             recursively_search(L', R', C');   // Recursion
12         C ← C \ {v'};
```

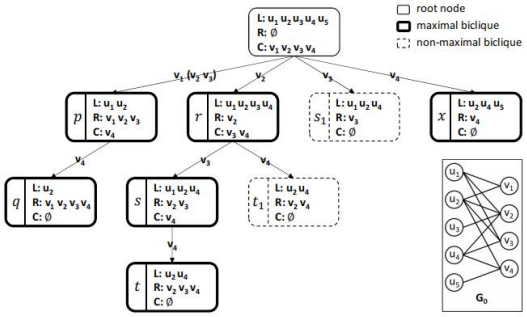


Figure 2: An enumeration tree for bipartite graph G_0.

GPU usage guidelines. There are several notable guidelines for improving the efficiency of GPU programs. First, dynamic memory allocations on GPUs are expensive because many threads may allocate new memory at the same time, causing problems such as thread contention, synchronization overhead, and memory fragmentation [37]. Therefore, we should carefully manage the valuable GPU memory and avoid frequent memory allocations. Second, we should minimize the thread divergence, i.e., threads in a warp take different execution paths on the GPU. Because GPU has to serialize the different execution paths, thread divergence can severely degrade the execution performance [1]. Third, we should pay more attention to the load balancing among multiple cores on the GPU, because thousands of cores have to wait for the slowest thread to run on a lightweight core, which is costly.

Recent optimizations. To reduce the computational overhead, researchers mainly focus on reducing the enumeration space, using various vertex ordering and pruning approaches [8, 14, 39]. To achieve further speedup, other works parallelized MBE algorithms on multicore CPUs [18] or distributed architectures [32]. Specifically, existing parallel MBE algorithms distribute all vertices v in V across CPU threads, and each thread generates a subtree correspondingly using 1-hop and 2-hop neighbors of v. However, their performance speedup is constrained by the limited parallelism of CPUs. For instance, the state-of-the-art parallel MBE algorithm ParMBE [18] costs over 40 minutes to enumerate all maximal bicliques on a medium-scale bipartite graph Github (180k vertices and 440k edges) on a 96-core CPU machine, as shown in Section 6.2.

4.1 Stack-Based Iteration with Node Reuse

To reduce memory usage, we propose a stack-based iteration with node reuse. The key idea is that we can reuse a node x to represent its child nodes with additional metadata stored in the node x. By doing this, we save the memory space allocated for the child nodes of x. We can efficiently support node reuse because during enumeration the $L \cup R \cup C$ of the child nodes of x is always a subset of the $L \cup R \cup C$ of its parent node x. For instance, vertices $\{u_2, v_1, v_2, v_3, v_4\}$ in child node q ($\{u_2\}, \{v_1, v_2, v_3, v_4\}, \emptyset$) is the subset of vertices $\{u_1, u_2, v_1, v_2, v_3, v_4\}$ in its parent node p ($\{u_1, u_2\}, \{v_1, v_2, v_3\}, \{v_4\}$) as shown in Figure 2. We describe this stack-based MBE algorithm with node reuse in Algorithm 2.

Algorithm 2: Stack-based Iterative MBE Algorithm

```
Data: Bipartite graph G(U, V, E)
Input: Set L_r ⊆ U, disjoint sets R_r, C_r ⊆ V
Output: All maximal bicliques
1  procedure iteratively_search(L_r, R_r, C_r):
2      node_buf.init_and_push((L_r, R_r, C_r));
3      while node_buf is not empty do            // Iteration
4          (L_p, R_p, C_p) ← node_buf.pop();
5          if C_p is not empty then
6              v' ← the smallest vertex in C_p;
7              node_buf.push((L_p, R_p, C_p \ {v'}));
8              L' ← L_p ∩ N(v'); R' ← R_p; C' ← ∅;
9              foreach v_c ∈ C_p do              // Node generation
10                 if L' ∩ N(v_c) == L' then
11                     R' ← R' ∪ {v_c};
12                 else if L' ∩ N(v_c) ≠ ∅ then
13                     C' ← C' ∪ {v_c};
14             if R' == Γ(L') then               // Maximality check
15                 Output(L', R') as a maximal biclique;
16                 node_buf.push((L', R', C'));
```

Specifically, instead of dynamically creating and freeing nodes for recursions in Algorithm 1, we **replace recursions** (line #11 in Algorithm 1) **with iterations** (lines #2-5, 7, 16 in Algorithm 2) and explicitly manage nodes with a stack-like structure node_buf. The node_buf structure and its update process can refer to Figure 5. A node_buf consists of a root node (L_r, R_r, C_r), the attribute *depth* of each vertex in $L_r \cup R_r \cup C_r$, and the *traversed vertices* from the root node to the current node. The *depth* of each vertex is updated according to the depth of the current node (i.e., the number of ancestor nodes of the current node). We can apply the node reuse

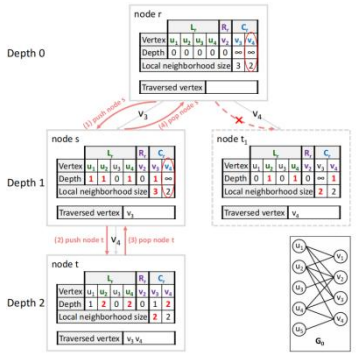
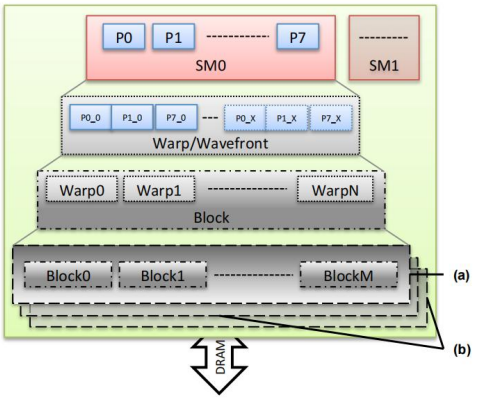


Figure 5: Illustration of GMBE with pruning using local neighborhood size.

面对下一个负载均衡的问题，研究人员使用了一个叫 persistent thread 持久线程的东西。什么是持久性线程？



This programming style (which we shall refer to as 'nonPT') forces the developer to abstract units of work to virtual threads. As the number of blocks is dependent on the number of work units, in most scenarios there are several hundreds or thousands more blocks to run on the hardware than can be initiated at kernel launch. In the traditional programming style, these extra blocks are scheduled at runtime. The switching of blocks is managed entirely by a hardware scheduler, with the programmer having no means of influencing how blocks are scheduled onto the SM. So while these abstractions provide an easy-to-program model by presenting a low barrier to entry for developers from a wide variety of application domains, it gets in the way of seasoned programmers working on highly irregular workloads that are already hard to parallelize. This exposes a significant limitation of the current SPMD programming style that neither guarantees order, location and timing, nor does it explicitly allow developers to influence the above three parameters without

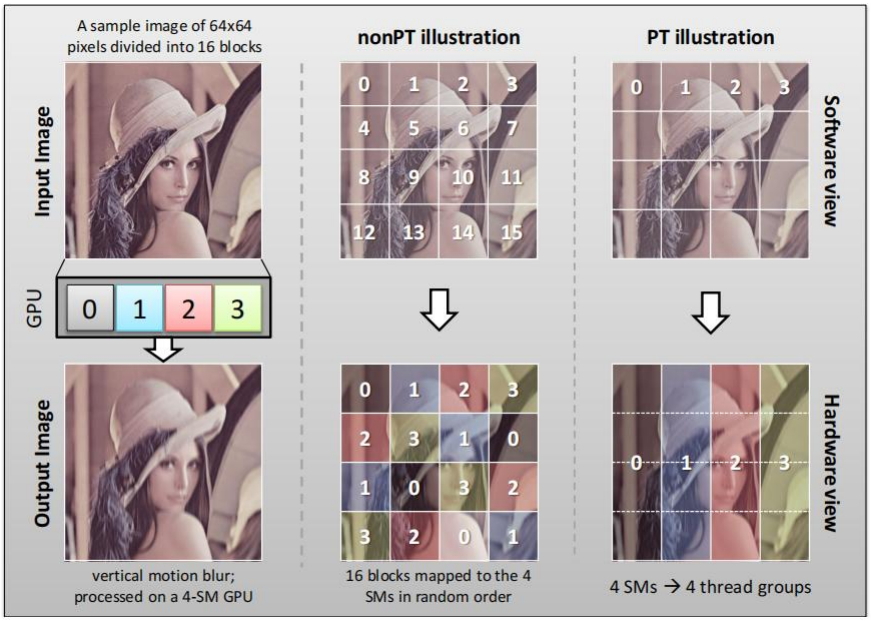
这是一个更加底层并且提升效率的线程设计。文章[5]重点分析了这个设计。GPU 的硬件和编程风格的设计使得它们严重依赖于单指令多线程（SIMT）和单程序多数据（SPMD）编程范例。这两种范式都**虚拟化了多层次的底层硬件**，从实际的硬件操作中抽象出软件程序员的视图。物理 SIMD 流多处理器（SM）的每个通道都被虚拟化成更大批量的线程，这些线程是 SIMD 通道宽度的倍数，称为扭曲或波前（SIMT 处理）。就像在 SIMD 处理中一样，每个 warp 都以锁步的方式操作，并执行相同的指令，在硬件处理器上进行时间多路复用。多个线程被组合在一起，形成一个更高的抽象，称为线程块 blocks，每个块内的线程允许在运行时通过 L1 缓存/共享内存或寄存器进行通信和共享数据。该过程被进一步虚拟化，多个线程块被同时调度到每个 SM 上，每个块在不同数据上的不同程序实例上独立运行（SPMD 处理）。

这种编程风格（我们将称之为“non-PT”）迫使开发人员将工作单元抽象到虚拟线程中。由于块的数量取决于工作单元的数量，在大多数情况下，在硬件上运行比内核启动时启动的块多几百或数千个块。在传统的编程风格中，这些额外的块在运行时被调度。块的切换完全由一个硬件调度器来管理，而**程序员则没有办法影响如何将块调度到 SM 上**。因此，虽然这些抽象通过为来自各种应用程序领域的开发人员提供了一个低的程序进入模型，但**它阻碍了经验丰富的程序员工作已经难以并行化的高度不规则的工作负载**。这暴露了当前 SPMD 编程风格的一个重大限制，它既不保证顺序、位置和时间，也不明确允许开发人员不影响上述三个参数。

另外一些可怕的性质，比如，1.主从设备导致 GPU 只能听从 CPU 发送具体代码，2.运行时决策使得我们不能保证将在何时何地调度块。3.块状态设计让 GPU 的一个新的块被映射到一个特定的 SM 时，该 SM 上的旧状态（寄存器和共享内存）会被认为是过时的，不允许块之间的任何通信，即使在同一 SM 上运行时也是如此。4.块间通信的唯一机制是全局内存，块作为独立内容，

这样的通信可能性能不足。5.生产消费结构，内核只能在运行到完成时生成数据。在这个内核产生的 GPU 上的数据需要另一个内核。6.内核独立单一，即内核不能调用自身的另一个副本（递归），不能生成其他内核，或者动态添加更多块。在调用之间存在数据重用的情况下，这种成本尤其昂贵。

为了规避这些限制，PT 诞生了，它可以涉及较低层次的抽象化，通过直接控制调度来提升性能。全文中最直白地解释持久化线程的便是下面这幅图：



在这幅图片里，假设我们的 GPU 一共有 4 个 SM 来执行对 Lenna 图像的处理。如果是 nonPT 的普通线程，GPU 内的硬件调度器就会启动 16 个 block 来执行处理操作，而这些 block 会在物理层面上对应到相应的 SM 上（SM0,SM1,SM2,SM3），但是，我们可以看到这样的操作就有点不是很好，1.调度器要重新分配线程和数据；2.如果是对图像变暗变亮，那还好，但是如果是卷积或者跟相邻块要共用的操作，那我们要花一部分时间重新搬运回数据。

但是我们看右边的 PT 线程，他们从始至终只有 4 个，对应 4 个 SM。首先从内存访问上来说，我们可以更好地安排存储，Cache 在导入 n-way 数据后，我们的 SM 访问时的 Cache Hit Rate 就可以更高（又是你，计组）。同时，数据在像素跨垂直块边界共享时进行重用，这对卷积什么的操作也挺不错的。

因此，编程的持久线程风格改变了虚拟软件线程的生命周期的概念，使它们更接近物理硬件线程的执行寿命，也就是说，让程序员能够控制线程在内核的持续时间。

那么，PT 线程有什么样的功用呢？文章[5]给了 4 个例子：

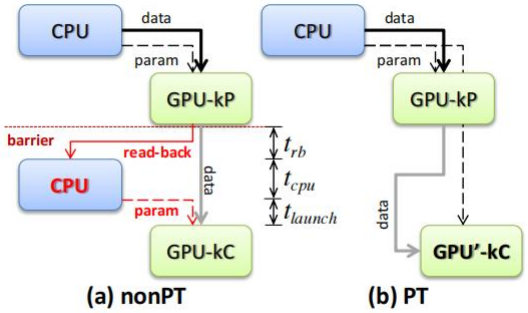
1. CPU-GPU 同步
2. 负载均衡（也是我们的前面文章用到的）
3. 本地化生产者-消费者
4. 全局同步

Use Case	Scenario	Advantage of Persistent Threads
CPU-GPU Synchronization	Kernel A produces a variable amount of data that must be consumed by Kernel B	nonPT implementations require a round-trip communication to the host to launch Kernel B with the exact number of blocks corresponding to work items produced by Kernel A.
Load Balancing	Traversing an irregularly-structured, hierarchical data structure	PT implementations build an efficient queue to allow a single kernel to produce a variable amount of output per thread and load balance those outputs onto threads for further processing.
Maintaining Active State	A kernel accumulates a single value across a large number of threads, or Kernel A wants to pass data to Kernel B through shared memory or registers	Because a PT kernel processes many more items per block than a nonPT kernel, it can effectively leverage shared memory across a larger block size for an application like a global reduction.
Global Synchronization	Global synchronization within a kernel across thread blocks	In a nonPT kernel, synchronizing across blocks within a kernel is not possible because blocks run to completion and cannot wait for blocks that have not yet been scheduled. The PT model ensures that all blocks are resident and thus allows global synchronization.

Table 2: Summary of Persistent Threads use cases

CPU-GPU 同步

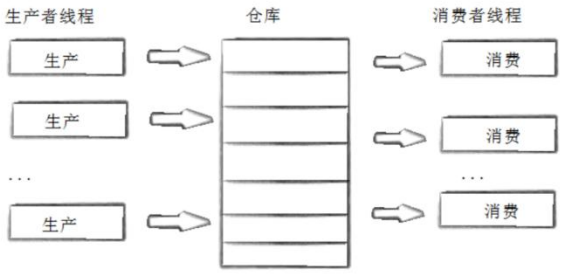
对于第一个功用，由于 CPU（主机）和 GPU（设备）作为主设备和从设备耦合在一起，因此设备缺乏向自己提交工作的能力。除了内核中内置的功能之外，GPU 还依赖于主机来发出所有的数据移动、执行和同步命令。如果生产者内核为消费者内核在运行时处理生成可变数量的项目，主机必须发出回读消息，确定消耗中间数据所需的块数，然后启动新内核。读取备份有显著的开销，特别是在主机和设备不共享同一内存空间或不在同一芯片上的系统中。



#1: CPU-GPU Synchronization

但是，在有持久性线程后，数据一旦从 CPU 来到 GPU-kP（生产者），之后就能一直计算到 GPU-kC，数据不会因为线程结束而自动消失，GPU 会保持运算一直到需要再次访问数据的时刻。这样，我们就节省了重新向 CPU 申请大量数据的时间。

这里涉及到一个生产者消费者模型的概念。我推测它跟 OS 的生成者消费者模型是差不多的。在我们的 CPU 多线程中，我们将程序分成生产者和消费者。生产者生产数据，消费者使用数据。



如果生产者生产数据的速度很快，而消费者消费数据的速度很慢，那么生产者就必须等待消费者消费完了数据才能够继续生产数据，同理如果消费者的速度大于生产者那么消费者就会经常处理等待状态，所以为了达到生产者和消费者生产数据和消费数据之间的平衡，那么就需要一个

缓冲区 buffer 用来存储生产者生产的数据。

生产者消费者模式就是通过一个仓库 buffer 来解决生产者和消费者的速度不匹配问题。生产者和消费者彼此之间不直接通讯，他们通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。

这样，我们就可以增加生产者线程或者消费者线程，而不会对整个程序的正确性产生影响，同时我们也可以根据不同硬件访问速度的不同动态地控制两者的比例。

在 GPU 上，我们通过将生产者和消费者 PT 化，让他们持久地运输数据或计算数据，就可以避免开启线程以及重新搬运数据的时间。

负载均衡

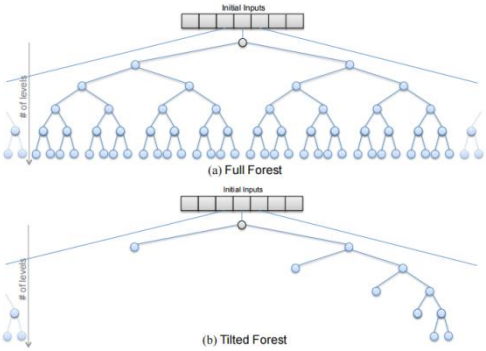
文章[12]针对光线射线追踪，使用了 PT 进行加速。

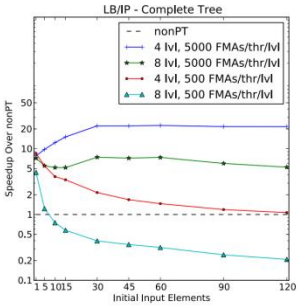
在文章中，PT 的主要作用是通过保持一些长时间运行的射线线程，避免了频繁创建和销毁线程的开销，从而提高了 SIMD 的效率和利用率。通过让每条射线处理时所有的遍历都保持在一个块中，绕过硬件调度器，从而可以提高线程对数据的重复利用。而在短时间内完成渲染的 PT，可以接着拿到下一个分配任务。如果我们不使用 PT，我们就会重开一个线程，刷新掉缓存里的数据，开销就更大了。

因此，PT 对于不规则的数据处理，比如树、图，是比较有效的。

```
DEFINITIONS
B = Box (xmin,ymin,zmin,xmax,ymax,zmax);
O = ray origin (x,y,z);
D = ray direction (x,y,z);
invD = (1/D.x,1/D.y,1/D.z);
OoD  = (O.x/D.x,O.y/D.y,O.z/D.z);
tminray = ray segment's minimum t value; ≥0
tmaxray = ray segment's maximum t value; ≥0

RAY vs. AXIS-ALIGNED BOX
// Plane intersections (6 x FMA)
float x0 = B.xmin*invD[x] - OoD[x]; [−∞,∞]
float y0 = B.ymin*invD[y] - OoD[y]; [−∞,∞]
float z0 = B.zmin*invD[z] - OoD[z]; [−∞,∞]
float x1 = B.xmax*invD[x] - OoD[x]; [−∞,∞]
float y1 = B.ymax*invD[y] - OoD[y]; [−∞,∞]
float z1 = B.zmax*invD[z] - OoD[z]; [−∞,∞]

// Span intersection (12 x 2-way MIN/MAX)
float tminbox = max4(tminray, min2(x0,x1),
                min2(y0,y1), min2(z0,z1));
float tmaxbox = min4(tmaxray, max2(x0,x1),
                max2(y0,y1), max2(z0,z1));

// Overlap test (1 x SETP)
bool intersect = (tminbox<=tmaxbox);
```

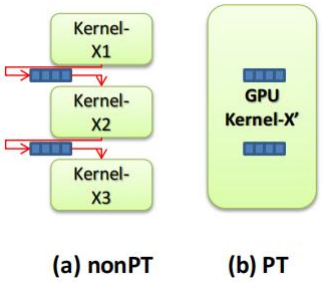
Figure 2: *Pseudocode for ray-box intersection.*

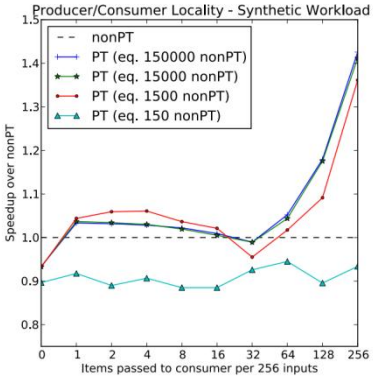




本地化生产者-消费者

在我们了解了 CPU 内的生产者消费者模型后，这个标题我们自然也很好理解了。

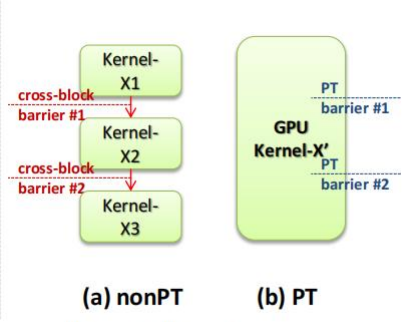




对于一个生产者消费者，如果我们不使用 PT，我们的 Kernel 之间想实现这个模型，就只能是一个生产者模型，一个消费者模型。他们通过 GPU L2 Cache 或者 DRAM 通信。这样的成本就非常大了。但是，如果我们现在只使用一个 Kernel，它经历两个阶段：阶段 A 是生产者，将数据拉到 GPU 并尝试填满内存；如果内存满了，Kernel 进入阶段 B：消费者，Kernel 开始搬运数据试图清空内存。此时 Kernel 由于是在 PT 情况下执行，这些内存始终是在离 Kernel 物理执行的 SM 处最近的地方，因此此时访存的成本就可以大大降低了。

全局同步

GPU 为在同一个块中的线程同步提供了硬件支持，但 GPU 对在同一个 SM 中的不同 block 乃至整个 GPU 中，没有提供同步。那么我们同样是可以使用一个大 PT 来解决问题的。我们将不同的 Kernel 合并成一个 Kernel-X'，当每个 blocks 完成一个计算时，它就开始处理下一个阶段的 blocks。这个阶段就是对块进行全局同步的阶段。



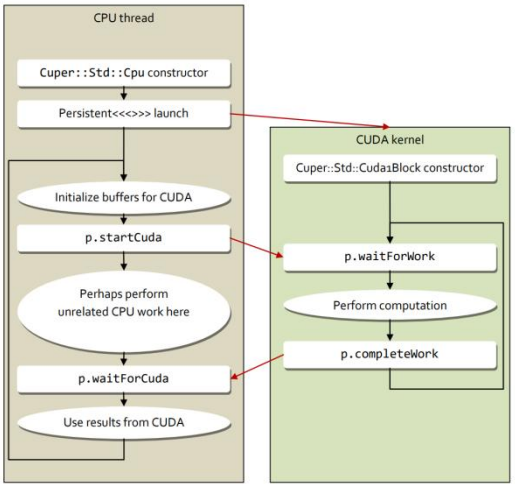
#4: Global Synchronization

所以我们可以看到，PT 作为一个持久化的线程，让我们更好地操作我们的 GPU。它的语法格式也还比较简单，具体类似如下：

RedHawk CUDA Persistent Threads (CuPer) API

The standard interfaces are provided in the <cuper.h> header file. All elements are declared within the Cuper::Std namespace. There are three classes defined therein: Cpu, Cuda1Block, and CudaMultiBlock. An object of the Cpu class is created in CPU source code. A typical usage would have a form similar to this:

```
void cpuFunction (…)
{
      Cuper::Std::Cpu  p;
      cudaHostGetDevicePointer(&d_A, h_A);
      Persistent<<<blocksPerGrid, threadsPerBlock>>>(p.token(), d_A);
      for (…) {
            … initialize h_A …
            p.startCuda();
            … possibly do unrelated CPU work …
            p.waitForCuda();
            … use results in h_A …
      }
      p.terminateCuda();
}
```




```
__global__ void Persistent (Cuper::Std::Token  token, float*  A)
{
      Cuper::Std::Cuda1Block  p(token);
      for (…) {
            p.waitForWork();
            if (p.isTerminated()) break;
            … perform workload …
            p.completeWork();
      }
}
```

CPU 在 lunch 线程后，CUDA 会接收说哪个线程现在处于启动状态（有点像 CPU 的线程启动模式），然后 CPU 说线程启动，CUDA 就开始计算，CPU 会在 waitforCUDA 那边等待 GPU 线程完

成的信号。

现在我们回到论文，通过在 PT 上进行改进，我们就可以对特定的搜索给定特定的线程，从而加速搜索的速度。

随后文章就到了实施细节和测试环节。实施环节提到了写算法时要特地注意的部分，测试则针对不同的数据集，不同的 GPU，多 GPU 进行了测试，并比较了三个优化产生的效果。这里我们就不再详细查看。但是总结一下的话，这篇文章里我们学到了下面 3 个技术以及研究方向：

1. 线程束级原语——更精细化的核函数设计

2. PT 持久线程——核函数间更负载均衡

3. 内存复用——更好的内存设计

Algorithm 4: Load-aware task-centric scheme in GMBE

```
processing_v is a global variable initialized as 0 ;
SM_task_queue is a global concurrent queue for load balance ;
// For each warp
1  procedure warp_kernel:
2      while true do
3          if SM_task_queue is not empty then
4              (L, R, C) ← SM_task_queue.dequeue() ;
5          else
6              v_s = atomicInc(processing_v) ;
7              if v_s ∈ V then
8                  L ← N(v_s); R ← {v_s}; C ← ∅;
9                  foreach v_c ∈ N_2(v_s) do
10                     if L ∩ N(v_c) == L then
11                         R ← R ∪ {v_c};
12                     else if v_c is with later order than v_s then
13                         C ← C ∪ {v_c};
14             else // All tasks on GPU have been processed
15                 return;
16         if R == Γ(L) then                    // Maximality check
17             if min{|L|, |C|} × |C| > bound_size and
                min{|L|, |C|} > bound_height then
18                 foreach v_t ∈ C do           // Node generation
19                     L_t ← N(v_t); R_t ← R; C_t ← ∅;
20                     foreach v_c ∈ C do
21                         if L_t ∩ N(v_c) == L_t then
22                             R_t ← R_t ∪ {v_c};
23                         else if L_t ∩ N(v_c) ≠ ∅ then
24                             C_t ← C_t ∪ {v_c};
25                     SM_task_queue.enqueue((L_t, R_t, C_t)) ;
26                     C ← C \ {v_t};
27             else
28                 iteratively_search(L, R, C);
```

从这篇文章中我们可以看到，1.内存是我们开始加速计算的一个起点，解决不了内存问题，加速计算就无从谈起。2.线程计算的精细化与复杂化是目前加速计算研究的一个方向，如何让线程计算更复杂的算法，让原本单核运行的算法多核化，才能让加速计算的应用方向更加广泛。

在 GPU 上复杂算法的实现是目前 GPU 研究中的一个重要方向。在矩阵计算之外，我们还有很多算法支撑着我们的生活，而要想更快，我们的移植是无法阻挡的。在这之中，想实现一个好的移植算法，我们就要操控好线程，进行精细化的设计。

“

有个周末我带女儿 Madison 去书店，然后就看到了这本书 OpenGL 手册，定义了硅谷

图形的计算机图形处理方式。一本 68 美元，我带了几百块钱，买了三本。

我把书带回办公室，对大家说：「我找到了咱们的未来。」我把三本书分发下去传阅，中间有大幅的折叠插页，这个插页就是 OpenGL 流水线计算机图形处理流水线。我把它交给了与我共同创办公司的那些天才手中。

我们以前所未有的方式实现了 OpenGL 流水线，构建出了世界从未见过的东西。其中有很多经验教训。对我们公司来说，那一刻给了我们极大的信心：即使对所做的事情一无所知，也能成功创造出未来。

现在这就是我对任何事情的态度。当有人跟我说我没听过的事情，或者听说过但不懂原理，我的想法总是：能有多难呢？可能看本书就搞定了，可能找一篇论文就能搞清楚原理。

我确实花了很多时间阅读论文，这是真的。当然，你不能照搬别人的做法，指望会有不同的结果。但你可以了解某件事情的实现原理，然后回归问题的本质，扪心自问：基于现有的条件、动机、手段和工具，以及一切如今的变革，我会怎么去重做这件事？我会如何重新发明它？我会如何设计它？

如果今天造一辆车，我会沿用过去的方式吗？如果今天让我创造一台计算机，我会采用怎样的方式？如果今天让我来编写软件呢？

这么想有道理吗？即使是今天的公司，我也经常回归本质，从头思考。这是因为世界已经变了。**过去编写软件的方式是单一的，是为超级计算机设计的，但现在软件架构已经解耦等等。我们今天思考软件、计算机的方式一直在改变。经常促使公司和自己回归问题本质，会创造出大量的机会。**

”

GPU 通信

看完这个方向后，我们再往一个系统一点的方向看看：**GPU 通信**。虽然我们在 Project4 有所了解，但是我们可以再深入了解一下。

Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect

Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker

Abstract—High performance multi-GPU computing becomes an inevitable trend due to the ever-increasing demand on computation capability in emerging domains such as deep learning, big data and planet-scale simulations. However, the lack of deep understanding on how modern GPUs can be connected and the real impact of state-of-the-art interconnect technology on multi-GPU application performance become a hurdle. In this paper, we fill the gap by conducting a thorough evaluation on five latest types of modern GPU interconnects: PCIe, NVLink-V1, NVLink-V2, NVLink-SLI and NVSwitch, from six high-end servers and HPC platforms: NVIDIA P100-DGX-1, V100-DGX-1, DGX-2, OLCF's SummitDev and Summit supercomputers, as well as an SLI-linked system with two NVIDIA Turing RTX-2080 GPUs. Based on the empirical evaluation, we have observed four new types of GPU communication network NUMA effects: three are triggered by NVLink's topology, connectivity and routing, while one is caused by PCIe chipset design issue. These observations indicate that, for an application running in a multi-GPU node, choosing the right GPU combination can impose considerable impact on GPU communication efficiency, as well as the application's overall performance. Our evaluation can be leveraged in building practical multi-GPU performance models, which are vital for GPU task allocation, scheduling and migration in a shared environment (e.g., AI cloud and HPC centers), as well as communication-oriented performance tuning.

Index Terms—Performance evaluation, GPU, interconnect, NUMA, PCIe, NVLink, NVSwitch, SLI, GPUDirect, RDMA, NCCL

我们从这篇 2020 年的综述性的文章开始，它评估了现代 GPU 通信方法：PCIe,NVLink,NV-SLI,NVSwithc, GPUDirect。

PCIe(Peripheral-Component-Interconnect-Express-Bus)是高速串行计算机扩展总线的标准，我们在 Project4 里详细介绍了 PCIe 与 GPUDirect 技术的点滴。然而，这对于需要大量数据的 GPU 来说，还是太慢了。传统的基于 PCIe 的 CPU-GPU 节点连接是一个树结构，CPU 与 GPU 之间使用 PCIe 交换机连接，CPU 内部使用 QPI 总线连接。但是，这意味着当 GPU 想跨树进行通信时，就得经过 CPU。

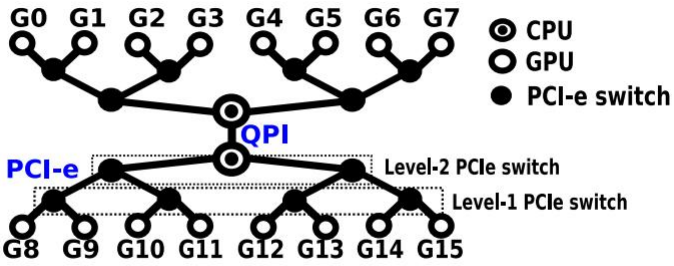


Fig. 4. PCIe interconnect topology in DGX-2.

于是，英伟达提出了 NVLink 技术。对于一机 8 卡，除了传统的 PCIe 连接 CPU 与 GPU 以外，GPU-GPU 通信使用 NVLink 技术，即图中的立方体的黑线。

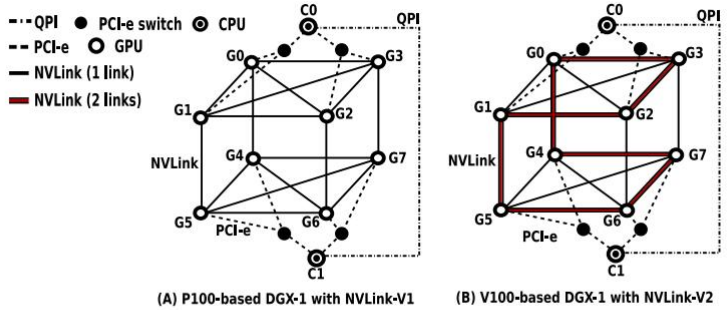
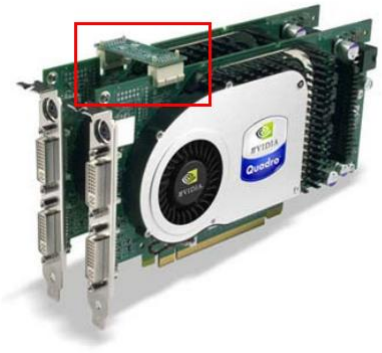


Fig. 1. PCIe and NVLink-V1/V2 topology for P100-DGX-1 and V100-DGX-1. The thin black connection between GPUs means they are interconnected by a single NVLink. The thick red connection in (B) means two GPUs are interconnected by two NVLink-V2s.

这种设计结构的特点在于，对于上平面和下平面，对角线也是连接的，形成两个完全连通的子集。这种拓扑设计是平衡的，在平面内具有更强的连通性。换句话说，在一个平面内访问是 UMA，而跨平面访问节点会导致 NUMA（当它们没有直接连接时，例如，从 GPU-0 到 GPU-7）。我们平常说的 NUMA 性，意思拓展为节点内通信时长与成本并不相同。

NVLink2 技术并没有选择进一步加强平面内的连通性，而是在超立方体-网格网络内形成了一个快速的骨干环。与其他链路相比，这个环中的每个连接都支持 2 倍的带宽，速度提升了两倍。NVLink2 技术的目的不是消除 NUMA 性，而是提高集体沟通的效率。



Scalable Link Interface (SLI)可伸缩的链接接口将两个 GPU 配对，这样它们就可以相互通信和代码游戏，共同运行 GPGPU 任务，或共同共享 GPU 内存空间。这是给游戏玩家想用两张卡搭建的便携方案，让玩家低成本地玩到 NVLink。

随着深度学习对 GPU 的需求大幅提高，NVSwitch 终于到来。NVSwitch 是一种基于 NVLinkV2 的交换机节点内通信芯片，每个交换机有 18 个 NVLink 端口。如下图所示，我们的单个节点内有两个基板，每个基板包含 6 个 NVSwitchs，节点内共有 8 个 GPU。一个 V100 GPU

包含 6 个 NVLink 插槽，8 个 GPU 一共 48 个口，正好对应 6 个交换机的 8 个口。每个 NVSwitch 的带宽在 50 GB/s，并且还是双工通信（天啊，这比 UART 高到不知道哪里去了）。

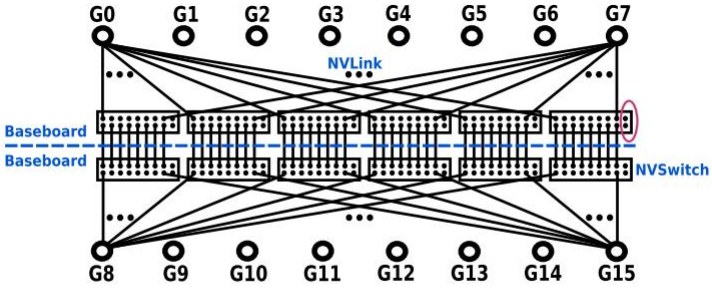
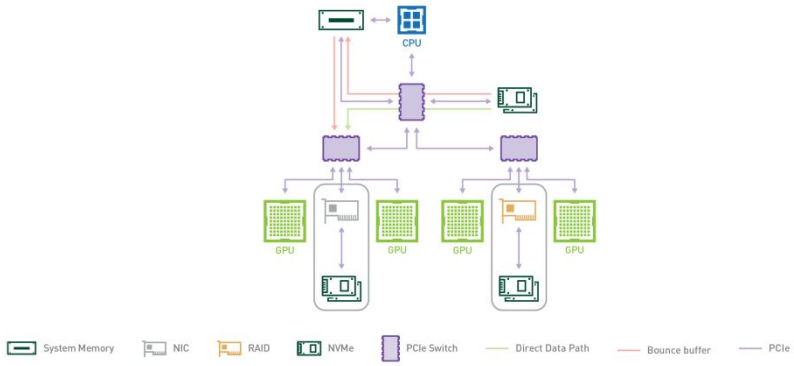


Fig. 3. NVSwitch interconnect topology in DGX-2.

自 Kepler 架构以来，NVIDIA GPU 引入了 GPUDirect-RDMA（AMD 其实也提出了 ROCn-RDMA ）。它使第三方 PCIe 设备，特别是 IB 主机通道适配器（即 HCA）可以通过 PCIe 直接访问 GPU 设备内存，无需 CPU 或主存分段，显著提高了节点间 GPU 通信的效率。为了实现 IB RDMA，GPU 供应商提供了一个 OS 内核扩展，以返回一个针对 GPU 设备内存的 DMA 总线映射。当用户创建一个 IB 区域时，它用 GPU 设备内存的目标地址向 IB 驱动器发出信号。IB 驱动程序然后调用一个例程来获取 DMA 映射。最后，一个正常的 IB 虚拟内存结构被返回给用户程序，就像它针对正常的 CPU 内存一样。



上面我们介绍了这么多的通信方法，这些方法可以让我们的 GPU 访问存储、互相通信地更快。但是，如果没有一个好模型，写多机多卡的 CUDA 程序员是会骂娘的。正如文章所说：有效地实现 CL 通信是具有挑战性的，因为：

(a)需要理解底层硬件网络拓扑，以实现协调映射；

(b)需要处理同步、重叠和死锁的问题；

(c)性能指标可能因应用程序特性而不同（例如，小传输面向延迟，但大传输面向带宽）。

为了减轻用户的这些负担，NVIDIA 提供了集体通信库（NCCL），使用与 MPI 集体类似的原语，（AMD 提供了 RCCL）。NCCL 目前支持五种 CL 模式：broadcast, all-gather, reduce, all-reduce, and reduce-scatter。

为了提供最大的带宽，NCCL 在通信的 GPU，CPU 之间构造环状网络，通过将数据分割成小块，并以管道的方式沿着环状数据进行传输，可以有效地实现广播和减少。NVIDIA 声称，该环算法可以为大多数标准的 NCCL 库内的操作提供接近最优的带宽，并且可以很容易地应用于各种网络拓扑。

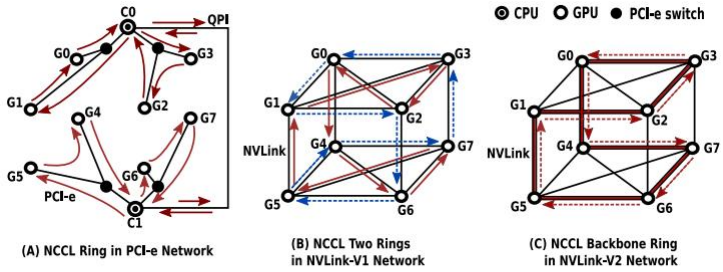
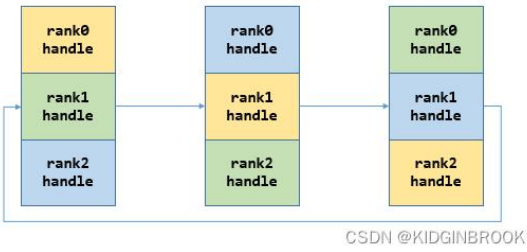


Fig. 18. NCCL Rings for PCIe, NVLink-V1 and NVLink-V2 interconnect. (A) for PCIe, the ring is to traverse the binary-tree network; (B) for NVLink-V1, there are two independent rings, marked in red-solid line and blue-dash line; and (C) for NVLink-V2, the lines with 2 links form a fast ring (i.e., the backbone network) while the lines with 1 link form a slow ring.

比较可惜的是，NCCL 库只开源了旧版本。我们可以从博客[7]中简单看看 NCCL 会做什么。这里一共有 14 篇文章，把他们总结下来就是：

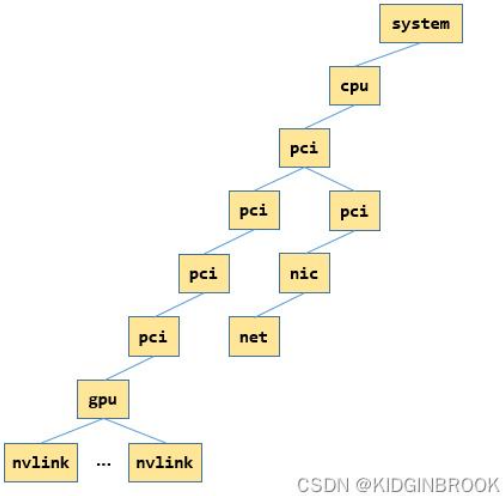
1. 在初始化的过程，NCCL 获取了当前机器上所有可用的 GPU、对应的 IB 网卡和普通以太网卡的信息（如 ip，连接方法），并保存下来，给每一个机器和端口生成 ncclUniqueId。

2. 在每一个 GPU 都有自己的编号后，我们就能做到上图 18 所想要的环形算法了。NCCL 会创建 bootstrap 环形网络连接，并保存到 ncclComm 里。而这里边的连接还是使用 MPI，MPI 给每一个 GPU 发送信号，给到他们的 rank 数。接着 rank n 的 GPU 会给 rank n+1 的 GPU 信号，构建环网。

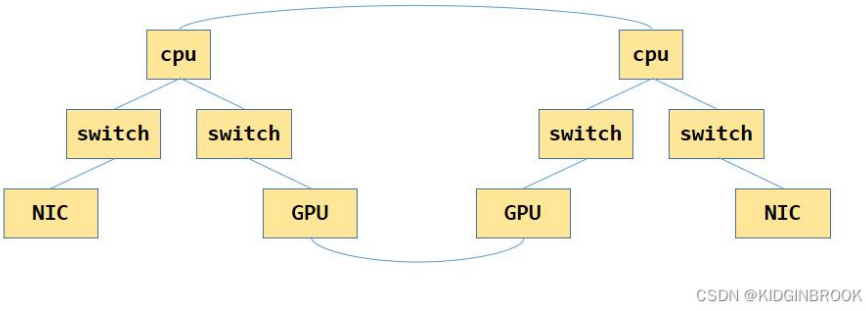



```
21  // Listen will return the local address via info (specify interface type 'findSubne
22  state->dev = idFromEnv ? findSubnetIf : 0;
23  void* extBstrapListenCommRoot;
24  NCCLCHECK(bootstrapNetListen(state->dev, &info.extHandleListen, &state->extBstrapLi
25  NCCLCHECK(bootstrapNetListen(state->dev, &info.extHandleListenRoot, &extBstrapListe
26
27  // stagger connection times to avoid an overload of the root at very high rank coun
28  if (nranks > 128) {
29    long msec = rank;
30    struct timespec tv;
31    tv.tv_sec = msec / 1000;
32    tv.tv_nsec = 1000000 * (msec % 1000);
33    TRACE(NCCL_INIT, "rank %d delaying connection to root by %ld msec", rank, msec);
34    (void) nanosleep(&tv, NULL);
35  }
36
37  // send info on my listening socket to root
38  NCCLCHECK(bootstrapNetConnect(state->dev, netHandle, &tmpSendComm));
39  NCCLCHECK(bootstrapNetSend(tmpSendComm, &info, sizeof(info)));
40  NCCLCHECK(bootstrapNetCloseSend(tmpSendComm));
41
42  // get info on my "next" rank in the bootstrap ring from root
43  }
```

3. 然而，这里 NCCL 还不知道我们集群的拓扑结构是怎么样的，它只是找到了所有人然后给他们轮成一圈。这里边每个人牵着谁的线还不知道呢。不过找结构的方法也没有想象的那么难，NCCL 会根据每个 rank GPU 的 hosthash 进行判断，如果相等的话说明在同一个机器。接着，它会建一个树结构：



4. 在拿到节点树后，NCCL 会把它变成 UML 格式。接着就可以建成我们之前看到的三维图了。具体其实跟 DFS 差不多，先从根节点递归遍历下去，直到遇到 nvlink xml 节点，然后拿到 nvlink 的父节点，即 gpu 节点，然后通过 tclass 获取对端 PCI 设备类型，如果是 gpu 或者 cpu，直接返回对端 node，如果是 nvswitch，那就先创建 nvswitch 节点，然后创建当前 gpu 节点和对端的双向边。然后通过 ncclTopoConnectCpus 将 cpu 两两连接。建好的图就像下面这样：



5. 接下来 NCCL 会先计算 GPU 和 NIC 节点到其他任意节点之间的最优路径，以及对应的带宽，即**最优路径上所有边的带宽的最小值**。算法设计一下，相当于给定一个无向图，每条边有一个权值，给定查询(u, v)，求节点 u 到节点 v 的路径，使得路径上的权值最小的边的权值尽可能大。

这其实就是个网络流问题，对应到 ADAH 就是 Lab10。我们看洛谷 P1396，用二分+SPFA 可以通过，或者使用有的人说的 MST+LCA。（哦，算法！）

题目描述

妈妈下班回家，街坊邻居说小明被一群陌生人强行押上了警车！妈妈丰富的经验告诉她小明被带到了 t 区，而自己在 s 区。

该市有 m 条大道连接 n 个区，一条大道将两个区相连接，每个大道有一个拥挤度。小明的妈妈虽然很着急，但是不愿意拥挤的人潮冲乱了她优雅的步伐。所以请你帮她规划一条从 s 至 t 的路线，使得经过道路的拥挤度最大值最小。

输入格式

第一行有四个用空格隔开的 n，m，s，t，其含义见【题目描述】。

接下来 m 行，每行三个整数 u, v, w，表示有一条大道连接区 u 和区 v，且拥挤度为 w。

两个区之间可能存在多条大道。

不过博客介绍说，NCCL 用的是并查集，在最后生成的图中，将不需要的网卡路径剪枝删除。嘛反正能达到目的就行。

6. 虽然建立了最小路径，但是，我们接下来要做的是建立 NCCL Channels。这是什么呢？一个 github issue[6]是这么回答的："Channels map to GPU SMs, so using more channels means using more GPU compute resources. NCCL tries to minimize the number of SMs it uses, while delivering the best performance. For TCP/IP sockets, the speed we're trying to achieve is usually less than 10GB/s, which we should be able to achieve with the minimal number of channels (2 currently, to ensure proper overlap and no bubbles)."

也就是说，我们可以把 Channel 理解为 NCCL 动态调配的一个通信网络口。它可以让 GPU 与别的 GPU 在 SM 中通信。如果 Channel 或者通信 SM 过少，我们的不能及时完成任务，如果 Channel 过多，对 SM 的资源占用也就过多。NCCL 的目标是利用必要的最小 SM 数量以实现最佳性能。

这里有几个有用的名词：

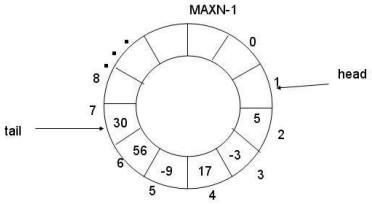
NCCL_NSOCKS_PERTHREAD：此参数允许将单个通道的传输分散到多个套接字上。通过在多个套接字上分配工作量，它有助于减轻创建额外通道（从而利用更多 GPU 资源）的需要。

NCCL_SOCKET_NTHREADS：此参数用于将网络 TCP/IP 封装工作分配到多个 CPU 核心上，使用多线程。它解决了单 CPU 核心可能无法实现最大网络带宽的情况，特别是在像 100GbE 卡这样的高速网络适配器的情况下。

NCCL_MIN_NCHANNELS：此参数指定 NCCL 使用的最小通道数。增加此参数可以帮助克服网络限制、解决 CPU 瓶颈或增加 GPU 并行性。NCCL 会根据系统的需求动态调整通道数量。

那么怎么去具体实现说想要几个 Channel 呢？我们仔细想想，一个人收数据，一个人发数据，其实跟我们之前学习的生产者消费者很像。我们能不能用生产者消费者确定要几个生产者消费者的算法，来确定 NCCL Channel 呢？

博客提到，NCCL 内的 channel 数据结果 collectives，就是一个类似于生产者消费者模型里的环形队列。这个环形队列就是我们之前 buffer 的具体实现。当环形队列为空或为满时，生产者和消费者才会相遇。消费者不能超过生产者，生产者不能超过消费者一个圈。生产者关心的是还剩多少剩余空间，消费者关心的是现有多少数据。生产者和消费者访问下标的行为互斥的，我们需要用一个 char，把它称为锁，来解决判定互斥问题。




```
1.      static ncclResult_t setupChannel(struct ncclComm* comm, int channelId, int rank, int nranks,
    int* ringRanks) {
2.        TRACE(NCCL_INIT, "rank %d nranks %d", rank, nranks);
3.        NCCLCHECK(initChannel(comm, channelId));
4.        struct ncclRing* ring = &comm->channels[channelId].ring;
5.        // Reorganize ranks to start with rank.
6.        int shift;
7.        for (shift = 0; shift<nranks; shift++) {
8.          if (ringRanks[shift] == rank) {
9.            break;
```

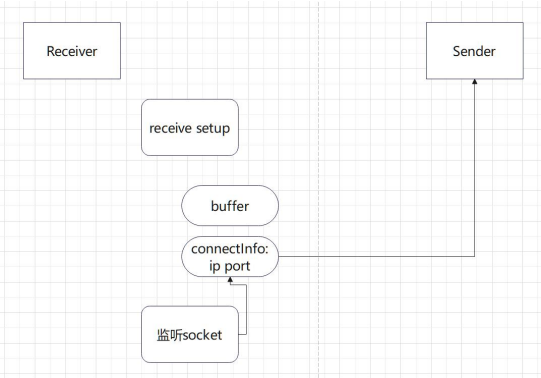
[6] How to decide the right number of NCCL channels? • Issue #438 • NVIDIA/nccl (github.com)

```
      }
    }
    for (int i=0; i<nranks; i++) {
      ring->userRanks[i] = ringRanks[(i+shift)%nranks];
    }
```

那么在这里，生产者就是发送方 SM，消费者就是接收方 SM。在多 GPU 同时计算且它们需要数据交换和同步时，这样的 NCCL 通道提供了一种有效的机制来管理数据传输。面对不同的 Channel，有多个不一样的生产者消费者，NCCL 就会基于之前找到的机器结构，搜索出多组 channel 用于之后的数据通信。这些 Channel 会被记录到 ncclTopoGraph 中。

7. 接下来是建立通讯链路的过程。博客写的很抽象，我也没学过计网，所以基于他给的总结尽量画图解释。

7.1 接收端执行 recv setup，创建 buffer，它将作为我们数据的临时集散地（因此这个 buffer 也自然是一个 FIFO 数据结构）。将相关信息记录到 connectIndo，启动一个监听 socket，ip port 同样记录到 connectInfo，通过 bootstrap 发送 connectInfo 到发送端。



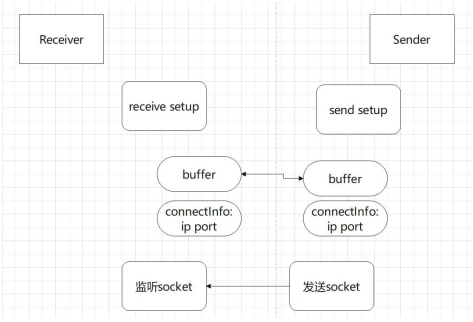
简单记录一下 Socket 是什么。Socket 是一种在计算机网络通信中使用的编程接口，用于在不同主机之间进行通信。它提供了一种抽象层，使得应用程序能够通过网络发送和接收数据。Socket 通常用于实现客户端-服务器模型，其中一个程序充当服务器，等待来自客户端的连接请求，而另一个程序充当客户端，向服务器发送请求并接收响应。

7.2 发送端执行 send setup，创建 buffer 等，将相关信息记录到 connectInfo，然后发送给接收端。这一步如果是 RDMA 场景，我们就不用 connectInfo（Project4 探索了部分 RDMA 的连接操作，RDMA 的通信记录会跟 OS 内的 RDMA 驱动程序和相关库深度绑定，我们 NCCL 就不用再来管理这方面的记录）。

7.3 发送端接受到步骤 1 中接收端的信息，然后建立发送端到接收端的链接，p2p 场景的话只是简单记录对端 buffer，RDMA 场景的话需要初始化 QP 到 INIT 状态。RDMA 中的 QP（Queue Pair，队列对）是 RDMA 通信中的一个重要概念，用于定义通信的端点和消息队列。每个 QP 都与一个本地端点和一个远程端点相关联，其中本地端点表示 QP 所在主机的地址和端口信息，远程端点表示与之通信的远程主机的地址和端口信息。

一个 QP 包含两个消息队列：发送队列（Send Queue）和接收队列（Receive Queue）。发送队列用于存储待发送的数据和 RDMA 操作请求，接收队列用于存储接收到的数据和 RDMA 操作完成事件。RDMA 适配器在进行 RDMA 操作时会自动处理这些队列，从而实现数据传输和通信的目

的。因此，这里 QP 其实就充当 Buffer 的作用，NCCL 会根据系统状态自动判定是否需要再启用 buffer。



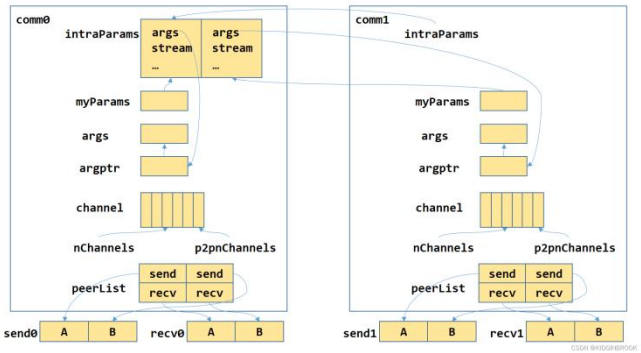
7.4 接收端 接受到步骤 2 中 send 发送的信息，然后建立接收端到发送端的链接，p2p 场景还是记录对端 buffer，RDMA 场景需要初始化 QP 到 RTS 状态，将本端的 QP 信息发送回对端。（RTS 状态表示发送端已经准备好发送数据，请求发送数据帧）

7.5 如果 RDMA 场景的话，发送端还需接收对端的 QP 状态初始化本端的 QP 到 RTS 状态。这是 RDMA 的 QP 是一个双向队列所需的额外确认。

在这之后，不同的接收端发送端就可以进行通信了。

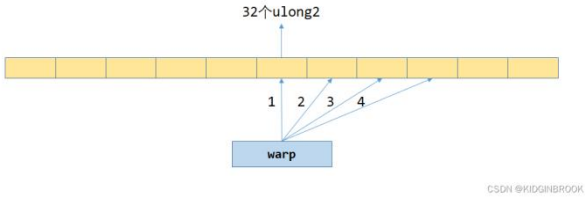
8. 这里博主介绍了单节点两个 GPU P2P 通信的情况。

这里 ncclSend/ncclRecv 的过程，主要就是两步，先通过 peerlist 将用户的操作记录下来，根据记录生成 kernel 所需要的参数，然后启动 kernel 执行拷贝即可。



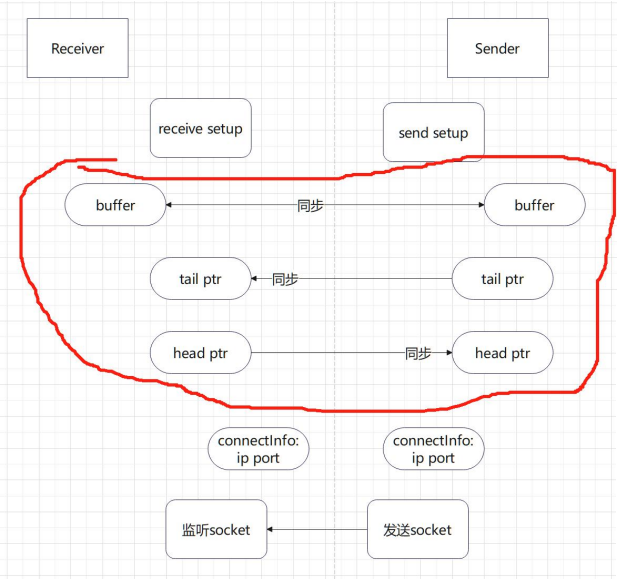
放到文中的图来讲，应该就是我们的数据 args stream 被打包好后，由一个 ptr 带队（跟 Project4 里的 data*挺像的），然后选择我们 GPU 内的一个 Channel，填写好发送方，随后发送。发送和接收的过程会根据我们之前确定的 Channel，启动对应的 CUDA 核函数，在对应的 SM 上执行通信。每个 ncclColl 会执行 ncclSendRecvKernel<4, FuncSum<int8_t>, int8_t>，有的线程负责发送，有的负责接收，还有一部分负责同步。在接收到数据后，还有一个核函数 ReduceOrCopyMulti，它执行从 buffer 或者 QP 里数据的拷贝，每次拷贝长度为 blockSize。

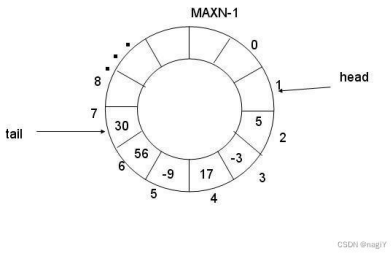
```cpp
template<int UNROLL, class FUNC, typename T>
__device__ void ncclSendRecvKernel(struct CollectiveArgs* args) {
  const int tid = threadIdx.x;
  const int nthreads = args->p2p.nThreads-2*WARP_SIZE;

  // Compute pointers
  const T* sendbuff = (const T*)args->sendbuff;
  T* recvbuff = (T*)args->recvbuff;

  if (args->p2p.delta < 0 ) return; // No-op

  if (args->p2p.delta == 0) {
    if (tid < nthreads && sendbuff != recvbuff) {
      // Local copy : ReduceOrCopyMulti takes an int as number of elements,
      // so we split it in blocks of 1G elements.
      int blockSize = 1<<30;
      for (size_t offset=0; offset<args->p2p.sendCount; offset += blockSize) {
        size_t remaining = args->p2p.sendCount - offset;
        if (remaining < blockSize) blockSize = remaining;
        ReduceOrCopyMulti<UNROLL, FUNC, T, 1, 1, 1, 1>(tid, nthreads, 1, &sendbuff, 1
        sendbuff += blockSize; recvbuff += blockSize;
      }
    }
```



对于不同卡的情况，send 将数据从用户指定的 sendbuff 拷贝到 NCCL P2P transport 的 buff，recv 将数据从 buff 拷贝到用户指定的 recvbuff，NCCL 通过 head，tail 指针来完成对发送和接收过程的协调；对于同卡的情况直接通过 kernel 将数据从 sendbuff 拷贝到 recvbuff 即可。

那么这里边 buffer 有一个重要的事情就是，记录数据传输到哪里了。这里是我看代码的理解，如果错了还请指出：就是我们建立连接的两端 buffer 是同步的，但是我们需要两个 ptr，对准我们发送的数据。第一个 ptr 是 tailptr，它是负责记录我们接收方接收数据到哪里了。第二个是 send ptr，它是负责记录发送方发到哪里了。每次这些 ptr 都会前进 block 个数据（因为发送和接收就是这个速度），这样两边的连接就可以根据发送与接收的速度动态调整自己的线程接收情况，他们还要判断 buffer 是否是满的，这样才能保证数据被完整发送接收。

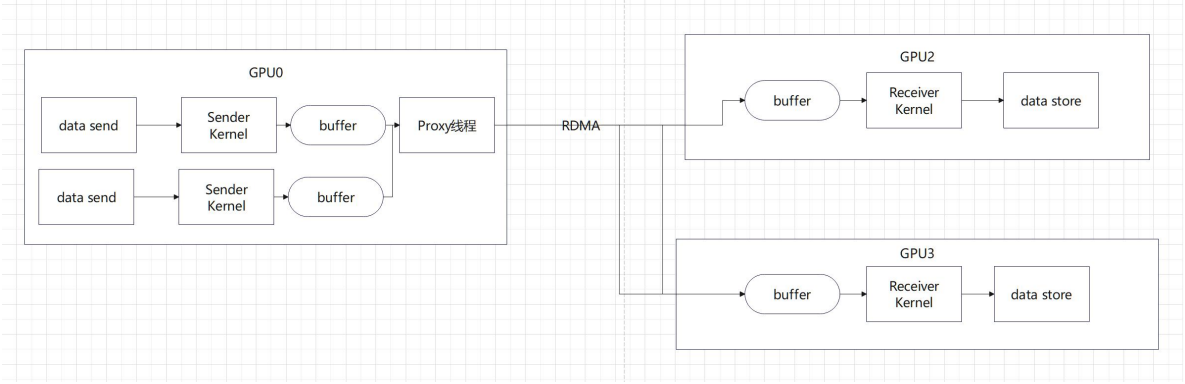




诶，这不就是我们的 OS 生产者消费者模型吗？

9. 多机多卡通信

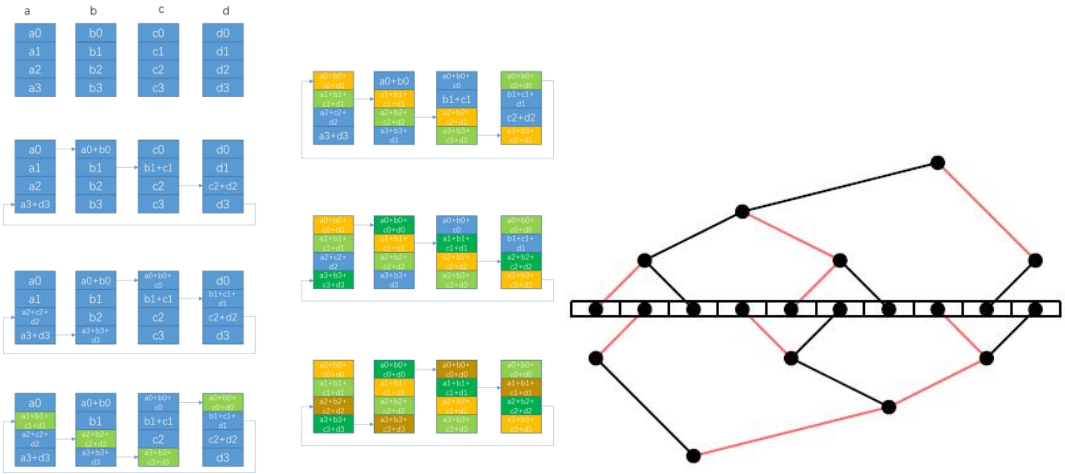
刚刚我们看完了 P2P 通信，那么，怎么样 PnP 呢？我们开启 n 个 Kernel，n 个 buffer，然后再在 Kernel 中每个都来一次上面的 setup。但是，这样的话，好像一个 Kernel 管理的事情有点多，能不能 Kernel 就负责发数据，然后另一个人送数据呢？所以，我们采用了一个经典的 proxy 的思路。Proxy 就像一个代理，它负责专门将一个 GPU 内不同的 SenderKernel 的 buffer 的数据送给不同的机器，GPU kernel 负责计算所需传输的数据的地址以及数据量大小，而 Proxy 线程负责完成实际的数据传输。



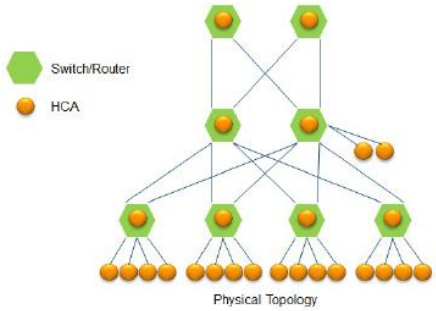
通信由 kernel 和 proxy 线程协调完成，send 端 kernel 负责将数据从 input 搬运到 buf，proxy 线程负责将 buf 中数据通过网络发送给 recv 端。kernel 和 proxy 间通过队列实现生产者消费者模式。

send 端通过 rdma send 发送数据，和 recv 端通过队列实现生产者消费者模式，队列位于 send 端，recv 端每次下发一个 wr 到 rq 之后会执行 rdma write 通知 send 端。

实现了多机多卡以后，后面的事情就好说了。比如文章 10 提到的 ring allreduce 环通信，文章 11 的 double binary tree 树通信（父亲节点发信号给子节点，上传下达），我们只要管理 proxy 里下一个数据发给谁就好了，自从把 proxy 独立出来后，后面的协议都能比较简洁地实现。



这也包括后面交换机的引入。通过相关的树算法，我们就能建立 GPU 与 Switch 的网络拓扑树，实现在 IB 下的 IB SHARP 和 NVLink 下的 NVLink SHARP。SHARP 技术是 IB 交换机的计算卸载（Compute Offload）技术。它的核心思想是将计算任务的一部分从 CPU 卸载到网络交换机中进行处理，以减轻 CPU 的负担，并提高网络通信的效率和性能。同样的，这个技术也减轻了 GPU 通信负担，并在 NVLink SHARP 中得到了专门针对 GPU 的优化操作。



(a) Physical Network Topology

在上完这节计网课后，再回过头看看文章中的测试结果，是不是觉得很多都可以进行解释了？比如说，我们的数据 latency，它有一个拐点，过了一定的数据规模后，latency 就会突然疯狂上涨。我们就有理由怀疑，是不是我们的生产者消费者模型中的 buffer 填满了，导致接收端接收数据的速度慢下来了？

我们可以看到，latency 的拐点出现处其实跟我们的通信 BW（GB/s）其实不是完全对齐的，比如在 Fig16 里的 A 图，PCIe 拐点在 14 次方，但是 B 图 PCIe 带宽拐点在 18 次方。NV-SLI 的 latency 拐点在 18 次方，而 B 图里大概是 18 多一点，相比 PCIe 还是有显著的改进。Local address 似乎在 18 次方-22 次方里就有所涨幅，但是它的带宽一直到 20 次方才满。不过为什么 local address 会有一个山峰的趋势呢？我猜可能是 Kernel 的数量导致的，从而引起了 SM 的竞争。不过这只是推测。

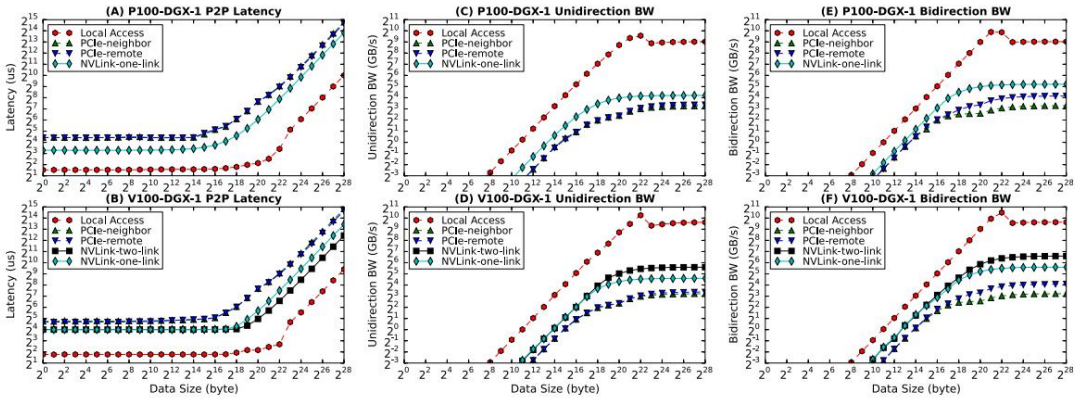


Fig. 15. P2P communication latency, unidirection and bidirection bandwidth with increased message size via PCIe and NVLink for DGX-1.

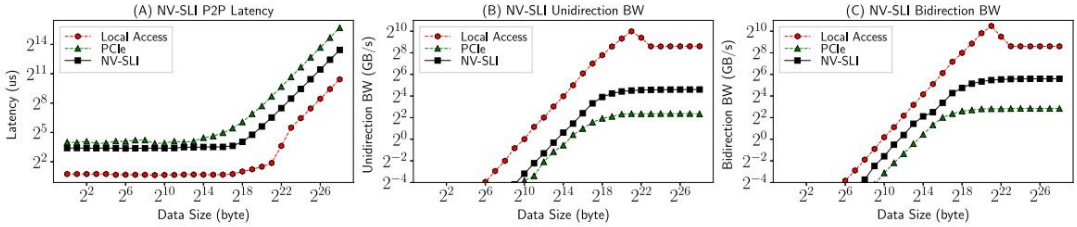


Fig. 16. P2P communication latency, unidirection and bidirection bandwidth with increased message size via PCIe and NV-SLI for the SLI-system.

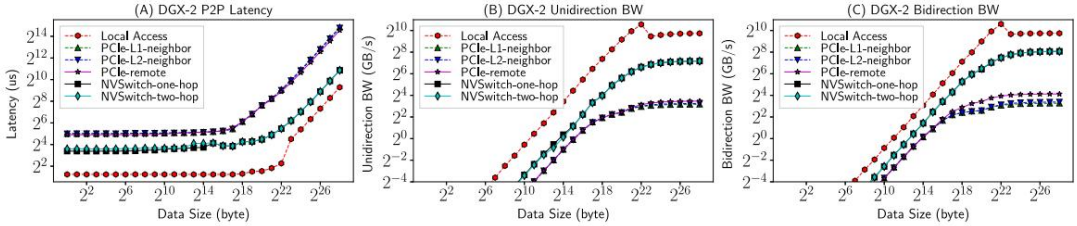
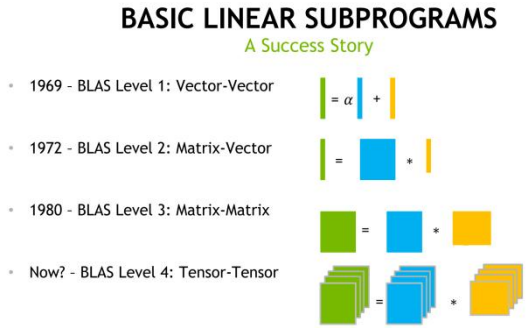
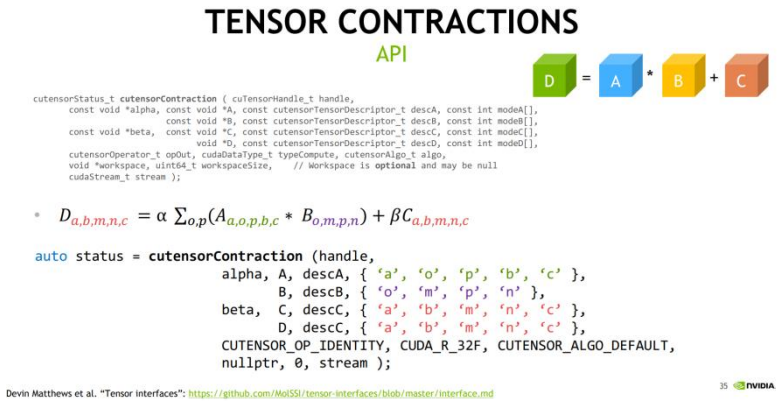


Fig. 17. P2P communication latency, unidirection and bidirection bandwidth with increased message size via PCIe and NVSwitch for DGX-2.

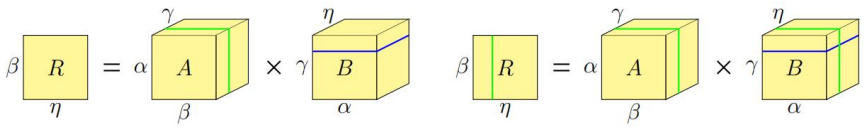
下一代 BLAS：Tensor-Tensor

在科学计算的历史上，BLAS 的成功是毋庸置疑的，它让无数计算得到了更快更好的匹配和优化。不过，“随着生成式人工智能时代的到来”，英伟达说，下一代 BLAS 应该是高维矩阵的乘加指令，也就是 Tensor 的运算，对此，他们也提供了新的 API。

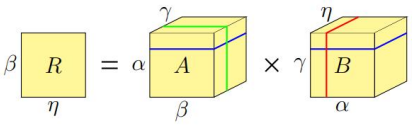




Tensor 的乘法跟 GEMM 相比，又多了许多新的优化方向，比如说，我们之前说的数据重排的操作，这回随着高维矩阵的引入，可能要重新思考了。同时，我们可能要把 SM 们分层 n 维，每一个维度的 SM 做那个维度的乘法。文章[17]试图将这种高维的方法规约回 GEMM。不过，新的算法的到来应该会继续加速这个算法。



(a) Slicing $\gamma \rightarrow$ COPY + GEMM　　(b) Slicing γ and $\eta \rightarrow$ GEMV



(c) Slicing γ and $\alpha \rightarrow$ GER

“

我认为我们正处于生成式人工智能革命的开端。如今，世界上进行的大部分计算仍然基于检索。检索意味着您触摸手机上的某些内容，它会向云端发送信号以检索一条信息。它可

能会用一些不同的东西组成一个响应，并使用 Java 将其呈现在您的手机的漂亮屏幕上。未来，计算将更加基于 RAG（Retrieval-augmented generation：检索增强生成，这是一个框架，允许大型语言模型从其通常参数之外提取数据），它的检索部分会更少，而个性化生成部分会高得多。

那一代将由 GPU 完成。所以我认为我们正处于这场检索增强的生成计算革命的开端，生成人工智能将成为几乎所有事物不可或缺的一部分。

”

CUDA 汇编：PTX

我们知道对于不同的体系结构的 CPU，比如 RISC-V，他们都有一套自己的指令集。那么，GPU 有没有自己的一套 ISA 呢？

这就是 PTX，并行线程执行（Parallel Thread eXecution，PTX），它有点像 Java 的字节码，是一个 a low-level parallel thread execution virtual machine and instruction set architecture (ISA)，这个指令会被发送到 GPU 上，由线程调度器来执行具体的 SIMT 操作。博客[9]探索了在 nvcc 编译器编译后的 cuda 中间字节码是什么样的。

当然，我们可以用 nvcc-gdb 或者 Nsight 来看看具体执行的情况。Nsight 就像我们之前看的 VTune 一样。

```
    84: __global__ void __launch_bounds__(num_threads_per_block) GPUBlackScholesCallPut(ir
0x0000000b00f93700               IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]
0x0000000b00f93710         @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ
    87:     int pos = blockIdx.x * blockDim.x + threadIdx.x;
0x0000000b00f93720               S2R R2, SR_CTAID.X
0x0000000b00f93730               S2R R3, SR_TID.X
0x0000000b00f93740               IMAD R2, R2, c[0x0][0x0], R3
    90:     while (pos < num_options)
0x0000000b00f93750               ISETP.GE.AND P0, PT, R2, c[0x0][0x160], PT
0x0000000b00f93760          @P0  EXIT
    87:     int pos = blockIdx.x * blockDim.x + threadIdx.x;
0x0000000b00f93770               MOV R3, 0x4
0x0000000b00f93780               IMAD.WIDE R2, R2, R3, c[0x0][0x178]
0x0000000b00f93790               LDG.E.SYS R2, [R2]
   117:         float expRT = ExpWrapper(-RISKFREE * T);
0x0000000b00f937a0               MOV R0, 0x14800000
0x0000000b00f937b0               MOV R7, 0xb
0x0000000b00f937c0               FMUL R5, R2, -0.019999999552965164185
0x0000000b00f937d0               IMAD.MOV.U32 R2, RZ, RZ, R0
0x0000000b00f937e0               MOV R3, R7
    69:     for (int i = 0; ; i+=10)
0x0000000b00f937f0               IADD3 R0, P0, R0, 0x28, RZ
0x0000000b00f93800               IMAD.X R7, RZ, RZ, R7, P0
    71:         g_arr[i] = val;
0x0000000b00f93810               STG.E.SYS [R2], R5
```

下面是 CUDA 官方教程的一个例子。程序的前两行跟我们的 asm 文件中.text 差不多，就是定义变量 r1 r2 (32bit，可以是 int)，array[N]（32 位 float）。然后就是.main，将我们的 thread id 赋值给 r1，然后将 r1 向左移动 2 位；接着将全局的 array[idx/4]赋值给 r2，然后 r2+=0.5；看起来在全局 Kernel 里这可能是给数组内的一部分施行加法操作。

Examples

```
        .reg     .b32 r1, r2;
        .global  .f32  array[N];

start:  mov.b32   r1, %tid.x;
        shl.b32   r1, r1, 2;          // shift thread id by 2 bits
        ld.global.b32 r2, array[r1];  // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;        // add 1/2
```

跟 CPU ISA 不一样的是，GPU 的 ISA 有 Vector 形式。其实我们也可以去翻翻 RISC-V 的 V 拓展指令集，这里边就实现了 32 位的寄存器集体 SIMD 操作，里边也有 Vector 的操作。Vector 的操作基本和数组差不多。不过，很快教程又介绍了高维矩阵 Tensor 的方法。

5.4.2. Vectors

Limited-length vector types are supported. Vectors of length 2 and 4 of any non-predicate fundamental type can be declared by prefixing the type with `.v2` or `.v4`. Vectors must be based on a fundamental type, and they may reside in the register space. Vectors cannot exceed 128-bits in length; for example, `.v4 .f64` is not allowed. Three-element vectors may be handled by using a `.v4` vector, where the fourth element provides padding. This is a common case for three-dimensional grids, textures, etc.

Examples

```
.global .v4 .f32 V;   // a length-4 vector of floats
.shared .v2 .u16 uv;  // a length-2 vector of unsigned ints
.global .v4 .b8  v;   // a length-4 vector of bytes
```

教程这样说道“PTX Tensor instructions treat the tensor data in the global memory as a multi-dimensional structure and treat the data in the shared memory as a linear data.”对于高维矩阵的操作，底层的 PTX 最高只支持到 5 维数据，这里边怎么对数据降维是一个有趣的研究方向。另外，教程额外花了很多篇幅描述数组越界 OOB 的处理情况。

Table 6: State Spaces

Name	Description
`.reg`	Registers, fast.
`.sreg`	Special registers. Read-only; pre-defined; platform-specific.
`.const`	Shared, read-only memory.
`.global`	Global memory, shared by all threads.
`.local`	Local memory, private to each thread.
`.param`	Kernel parameters, defined per-grid; or Function or local parameters, defined per-thread.
`.shared`	Addressable memory, defined per CTA, accessible to all threads in the cluster throughout the lifetime of the CTA that defines it.
`.tex`	Global texture memory (deprecated).

在 CUDA 内就像 CPU 的状态寄存器 PSW 一样，在块中（在指令集里，CUDA 称他们为 CTA），有%tid 寄存器，记录当前是哪个线程，有%ntid，多少个线程；%warpid，warp 的 id；%ctaid，我们的块 CTA 的 id；%smid，SM 的 id；%clusterid，grid 中存储的集群的 id 等等。

GPU 内也有特殊指令。比如 NVIDIA 发家的视频图像处理：

All scalar video instructions operate on 32-bit register operands. The scalar video instructions are:

- vadd
- vsub
- vabsdiff
- vmin
- vmax
- vshl
- vshr
- vmad
- vset

另外一个有趣的指令叫 DPX，这是 Hopper 架构里的指令。我们看文章[7]。这个指令专门用来加速 DP 动态规划算法。

在 DP 里边，最重要的事情是状态转移方程。写出这个方程，DP 基本上就完成 80%了。而这个状态转移方程也是 TPX 想加速的地方。比如这个指令 __viaddmin_s16x2_relu，这样的指令就可以很好地实现我们的 DP 转移方程。TPX 加速了 Smith-Waterman、Needleman-Wunsch 等 DP 算法。这感觉真不戳，以后算法课再也不怕卡常了。

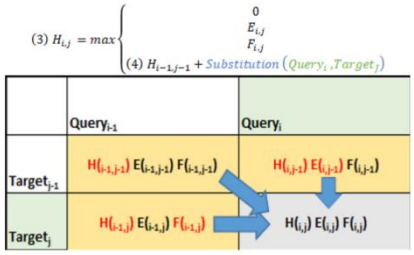


图 6。在 Smith-Waterman 实施步骤中更新插入惩罚、删除惩罚和分数

NVIDIA Hopper 架构 math API 为此类计算提供了巨大的加速。 API 将 NVIDIA Hopper 流式多处理器提供的加速作为融合运算（例如，__viaddmin_s16x2_relu，一种按半字执行的内在函数 $max(min(a+b,c),0)$）进行加法运算，然后进行最小或最大加法运算。

看懂 GPU 的 ISA，可以帮我们在日后更好地调程序。

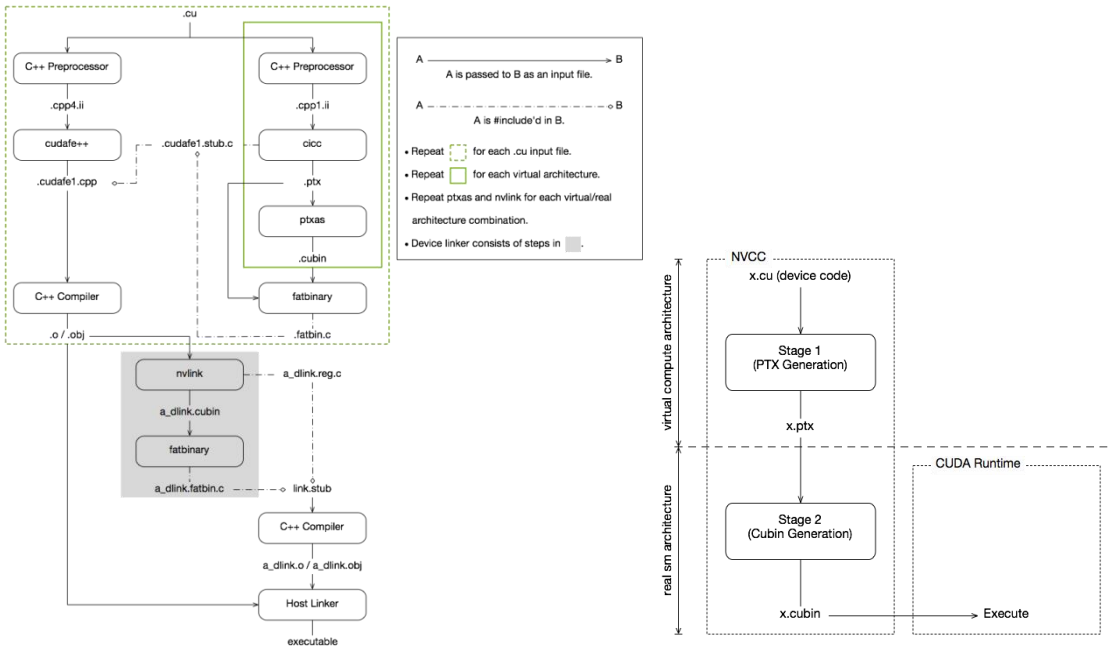
对了，顺带一提，nvcc 是怎么编译我们的 CUDA 代码的呢？我们看看官方教程[11]是怎么说的。

CUDA 编译工作原理如下：输入程序经过预处理后进行设备编译，并编译成 CUDA 二进制（cubin）和/或 PTX 中间代码，然后将其放入二进制文件（fatbinary）中。输入程序再次进行预处理，以进行主机编译，并进行合成以嵌入二进制文件，并将 CUDA 特定 C++ 扩展转换为标准 C++ 结构。然后，C++ 主机编译器将合成的主机代码与嵌入的二进制文件一起编译成主机对象。实现这一目标的具体步骤如下图所示。

每当宿主程序启动设备代码时，CUDA 运行时系统都会检查嵌入的二进制文件，以便为当前的 GPU 获取合适的二进制文件。

CUDA 程序默认以整个程序编译模式编译，即设备代码不能引用来自单独文件的实体。在整个程序编译模式下，设备链接步骤不起作用。

[7] 使用 NVIDIA Hopper GPU DPX 指令提高动态编程性能 - NVIDIA 技术博客



可以看到，编译过程其实分两部分，一部分是主机端和普通 c++一样的编译，另一部分是针对 CUDA 中扩展的 C++程序的编译，GPU 设备端的编译最终的结果文件为 fatbinary 文件，GPU（的驱动）通过 fatbinary 文件来执行 GPU 功能。

那么为什么要有个 PTX 的中间层？其实和 Java 的想法是一样的：一次编译，到处运行。GPU 到目前迭代了那么多代，如果像 gcc 一样只出那么几代的话，那么 nvcc 的底层要疯狂地写，这是非常不现实的。我们能做的就是转换成 PTX，它作为一个虚拟 GPU 的程序集。虚拟 GPU 提供的这个通用的 PTX 指令集下，不同的硬件只要实现了，就可以运行，在此二进制指令编码是一个无关紧要的问题。

因此，nvcc 编译命令总是使用两种体系结构：虚拟中间体系结构，以及真实的 GPU 体系结构来指定要执行的目标处理器。要使这样的 nvcc 命令有效，真正的体系结构必须是虚拟体系结构的实现。这些虚拟框架由 compute_开头，比如 sm_50。

由此可见，如果我们在写程序时始终尽可能小的用虚拟架构中的 ISA，我们就能最大限度地提高运行的实际 GPU。可是这样就会继续带来问题：我们的新指令假如效果更高，受到通用程序的影响，我们又不敢用新指令了。这可怎么办？

在我们的老朋友即时编译（JIT）的帮助下，GPU 设备有能力知道它应该用哪个 PTX。

通过指定虚拟代码架构而非真实 GPU，nvcc 可将 PTX 代码的组装推迟到应用程序运行时，因为此时目标 GPU 已完全确定。例如，当应用程序在 sm_50 或更高架构上启动时，下面的命令允许生成完全匹配的 GPU 二进制代码。

```
1.      nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50
```

即时编译的缺点是会增加应用程序的启动延迟，但这可以通过让 CUDA 驱动程序使用编译缓存来缓解（请参阅 "第 3.1.1.2 节：即时编译"，《CUDA C++ 编程指南》），该缓存可在应用程序的多次运行中持续存在。

当然，一个更普适性的想法是，既然我们的 LLVM 下的 JIT 还要制定代码来执行，那我为什么

不在编译时就把 compute_50，compute_40 等等等等全部编译出来，到时候执行的时候哪个中我就用哪个呢？

这就是刚刚那副图中 fatbinary 的来历。它存储了多个计算框架，这些代码能够保存相同 GPU 源代码的多个翻译。 在运行时，CUDA 驱动程序将在设备功能启动时选择最合适的翻译。

按照博客[12]里的总结如下：

1. CUDA 程序的编译必须经历两个过程，即虚拟框架和真实框架，虚拟框架决定了程序最小的可运行 GPU 框架，而真实框架决定了程序可运行的最小的实际 GPU。例如-arch=compute_30;-code=sm_30 表示计算能力 3.0 及以上的 GPU 都可以运行编译的程序。但计算能力 2.0 的 GPU 就不能运行了。

2. 即时编译（Just-In-Time）机制让程序可以在大的 GPU 框架内动态选择与电脑 GPU 最合适的小代。

3. –generate-code 保证用户 GPU 可以动态选择最适合的 GPU 框架（最适合 GPU 的大代和小代）。

所以，我们之前用的 O3 优化，在 CUDA 这里可能起的作用就不是非常的大。博客[13]对此做出了一些解释。我们的 O3 是在 CPU 端的优化，而想优化 GPU 上的，我们就得加上-Xptxas，这个东西会选择最好的 PTX 指令。但是，更好的优化地区还是在我们自己程序员上。

最佳答案

警告:编译 `nvcc -O3 filename.cu` 将仅将 -O3 选项传递给主机代码。

为了优化 CUDA 内核代码，您必须将优化标志传递给 PTX 编译器，例如:

```
nvcc -Xptxas -O3,-v filename.cu
```

将要求优化级别 3 到 cuda 代码(这是默认设置)，而 `-v` 要求详细编译，它报告了非常有用的信息，我们可以考虑进一步优化技术(稍后会详细介绍)。

另一个可用于 nvcc 编译器的速度优化标志是 `-use_fast_math` 它将以牺牲浮点精度为代价使用内部函数(请参阅 Options for Steering GPU code generation)。

无论如何，根据我的经验，这种自动编译器优化选项通常不会实现很大的提升。通过显式编码优化可以实现最佳性能，例如:

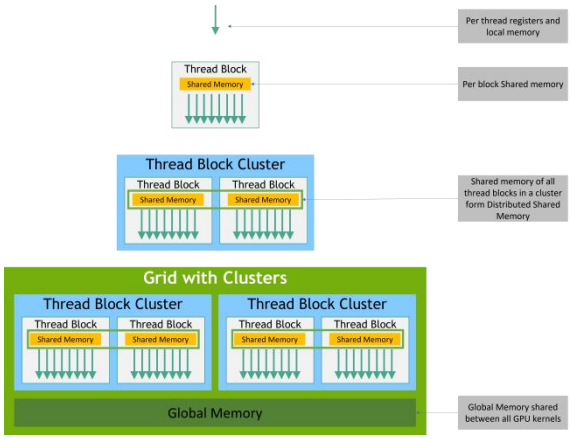
- **指令级并行 (ILP)** :让每个 CUDA 线程在多个元素上执行其任务 - 这种方法将保持管道加载并最大化吞吐量。例如，假设您要处理 NxN tile 的元素，您可以使用 2 级 TLP 启动 NxM 线程块(其中 M=N/2)，并让 threadIdx.y 在 2 个不同的元素行上循环。
- **寄存器溢出控制** :控制每个内核使用的寄存器数量并使用 `-maxrrregcount=N` 进行实验选项。内核需要的寄存器越少，有资格并发运行的块就越多(直到寄存器溢出将接管)。
- **循环展开:**尝试添加 `#pragma unroll N` 在 CUDA 内核中的任何独立循环(如果有)之前。 N 可以是2、3、4。当您在套准压力和达到的展开水平之间达到良好的平衡时，就会达到最佳效果。毕竟，这种方法属于 ILP 技术。
- **数据打包** : 有时你可以加入不同的相关缓冲区数据，比如 `float A[N],B[N]` , 到 `float2 AB[N]` 的一个缓冲区中数据。这将转化为更少的加载/存储单元操作和总线使用效率。

当然，始终，始终，始终检查您的代码以将内存访问合并到全局内存并避免共享内存中的存储库冲突。使用 nVIDIA Visual Profiler 更深入地了解此类问题。

GPU 安全&虚拟化

训练 AI 的数据是用户的隐私。但是人们发现，我们可以从 GPU 上进行攻击，绕过 CPU 与 OS 搭建的安全防御系统。比如我们想象一下之前的持久化线程，假如用户 A 先在 SM1 训练了一堆数据，接下来用户 B 的任务要在这个 SM 上执行任务。如果是为了效率，我们可以不刷新数据，让

B 的任务直接跑起来。但是这就意味着 B 可能可以拿到 A 的计算结果。这可能是 SHA256 或者什么密码，造成攻击。



同时，多个用户使用的多个程序部署到 GPU 上也是一个问题。GPU 就像以前的大型机一样，要有一个时分复用以及调度系统。怎么样调配不同的 SM 给到不同的用户，让 GPU 也可以多路复用，这是 GPU 要考虑的一个重要问题。

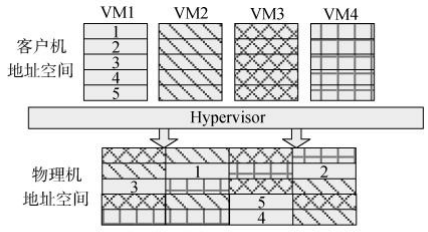


图 7　虚拟机内存与物理机内存的关系

Figure 7　Relationship between virtual memory and physical machine memory

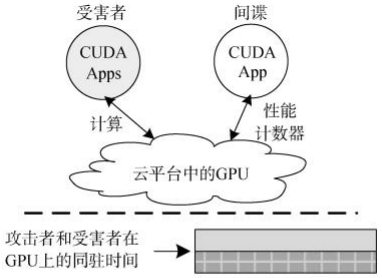
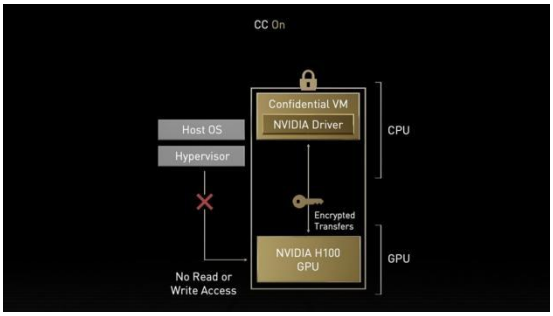


图 24　基于资源度量的侧信道攻击场景

Figure 24　Attack scenario of profile-based side channel

在 H100 中，工程师设计了一个 Confidential VM，它将 CPU 内的驱动锁住。新的机密计算支持保护用户数据，抵御硬件和软件攻击，并在虚拟化和 MIG 环境中更好地隔离和保护虚拟机（VM）。H100 实现了世界上第一个本机机密计算 GPU，并以全 PCIe 线速率使用 CPU 扩展了可信执行环境（ TEE ）。



另外，在多路复用中，GPU 也开始注意性能的。如何更好地切换以及多路复用[27]。

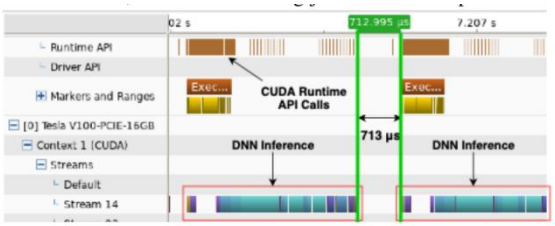


Fig. 5. Alexnet inference with single CUDA stream

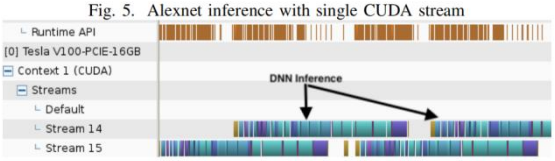
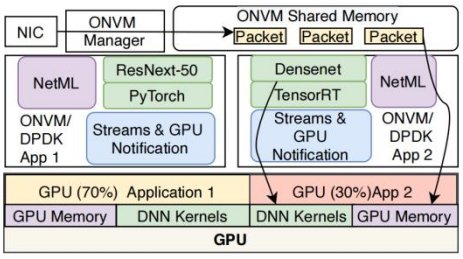


Fig. 6. Alexnet inference with two CUDA streams

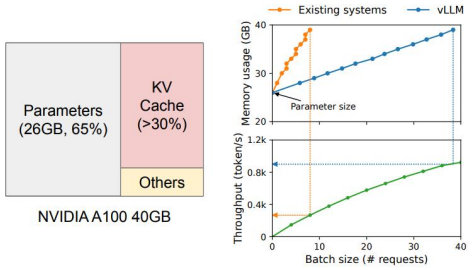
IV. A First Step to Improving Multiplexing



vLLM:降低 GPU 训练 LLM 的内存需求

随着 LLM 训练数据的增加，KV 缓存的大小也在迅速扩大，这在处理大规模数据时尤为明显。

KV 缓存是指关键-值缓存，它是在大型语言模型中用于存储上下文信息的一种数据结构。在语言模型中，关键-值缓存用于存储先前生成的令牌的上下文信息，以便在生成下一个令牌时使用。关键-值缓存中的关键是输入的令牌，而值是与该令牌相关的上下文信息。通过使用 KV 缓存，语言模型可以利用先前生成的令牌的上下文信息来生成更准确的下一个令牌，从而加速 LLM 的训练速度。



以一个含有 13B 参数的 OPT 模型为例，每一个 token 的 KV 缓存就需要占用 800 KB 的空间。而 OPT 模型能生成的 token 序列最多可达 2048 个，因此在处理一个请求时，KV 缓存所需的内存空间可能高达 1.6 GB。这种情况在当前 GPU 的资源稀缺环境下尤为突出，因为即便是主流 GPU 的内存容量也只有几十个 GB，如果将所有可用内存都分配给 KV 缓存，那么也仅能处理几十个请求。而且，如果内存管理不够高效，还会进一步降低批处理的大小，导致资源利用率进一步降低。与此同时，GPU 的计算速度的增长速度是超过内存容量的，这让我们相信，随着时间的推进，内存的瓶颈问题将变得越来越明显，可能会严重影响数据处理和模型训练的效率。

vLLM 这个系统采用一个集中式的调度器来协调分布式的 GPU 工作节点。其中的 KV 缓存管理器能够以“分页”的方式有效地管理 KV 缓存，这正是得益于 PagedAttention 算法。具体来说，KV 缓存管理器通过中央调度器发送的指令来管理 GPU 工作节点上的物理 KV 缓存内存。

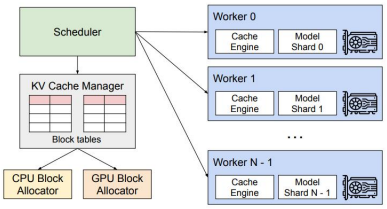


Figure 4. vLLM system overview.

Although compaction [54] has been proposed as a potential solution to fragmentation, performing compaction in a performance-sensitive LLM serving system is impractical due to the massive KV cache. Even with compaction, the pre-allocated chunk space for each request prevents memory sharing specific to decoding algorithms in existing memory management systems.

传统的注意力算法往往要求在连续的内存空间中存储键和值，但 PagedAttention 允许在非连续的内存空间中存储连续的键和值。1. PagedAttention 将每个序列的 KV 缓存分割成多个 KV 块。2. 每个块包含固定数量 tokens 的键和值向量，这个固定的数量被称为 KV 块大小。3. 公式部分给出了如何将传统的注意力计算转换为基于块的计算[16]。

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t \mathbf{1} / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \qquad (4)$$

在这种页注意力机制下，不同的 page 被分配到不同的内存空间中。vLLM 通过动态地为逻辑块分配新的物理块，随着更多 tokens 及其 KV 缓存的生成，优化了内存的利用。在这种机制下，所有的块都是从左到右填充的，仅当所有之前的块都已满时，才会分配一个新的物理块。这种设计帮助将一个请求的所有内存浪费限制在一个块内，从而可以更有效地利用所有的内存。

这种内存管理策略不仅有助于减少内存浪费，还通过允许更多的请求适应于内存中的批处理，提高了系统的吞吐量。每个请求的处理变得更为高效，因为现有的内存资源得到了更好的利用。当一个请求完成其生成过程后，其占用的 KV 块可以被释放，从而为其他请求的 KV 缓存提供存储空间。

这种动态分配和释放物理块的机制，为 LLM 服务中的内存管理提供了一种解决方案。

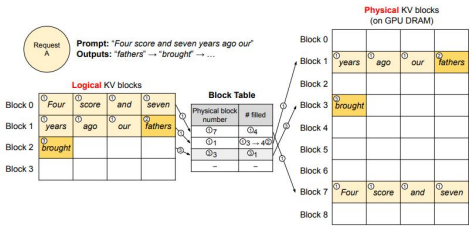


Figure 6. Block table translation in vLLM.

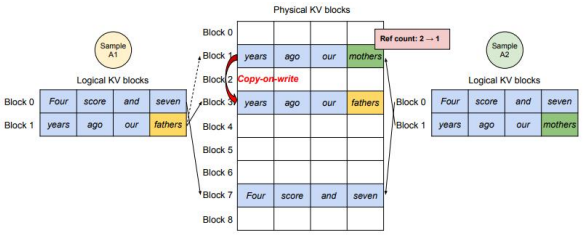


Figure 8. Parallel sampling example.

流水线

我们在 Project4 里分析了一部分关于 GPU 内存中统一内存的发展。但是，我们还没有仔细思考过如何使用内存。比如说，模型是 one-hot 编码进去，还是用什么编码方式进入 GPU。

我们接下来以 Megatron-LM 这一经典的流水线训练模型为例子（文章[29]）。通过 pipeline 并行，一个模型可以被拆解为多份放到不同节点上，以 transformer 为例，在 transformer 中 block 多次重复，所以每个 device 会分别处理相同个数的 layer，对于非对称结构不好切分，这里不做考

虑。如下图，不同 TransformerLayer 做为不同的 pipeline 的 stage，也称为 partition。

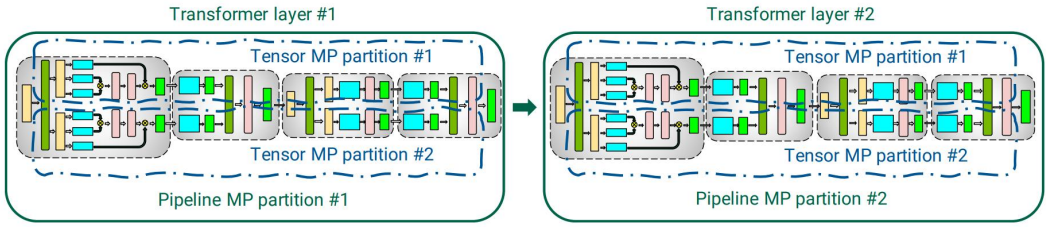
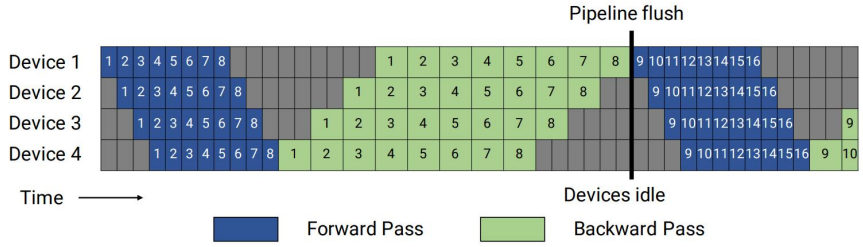
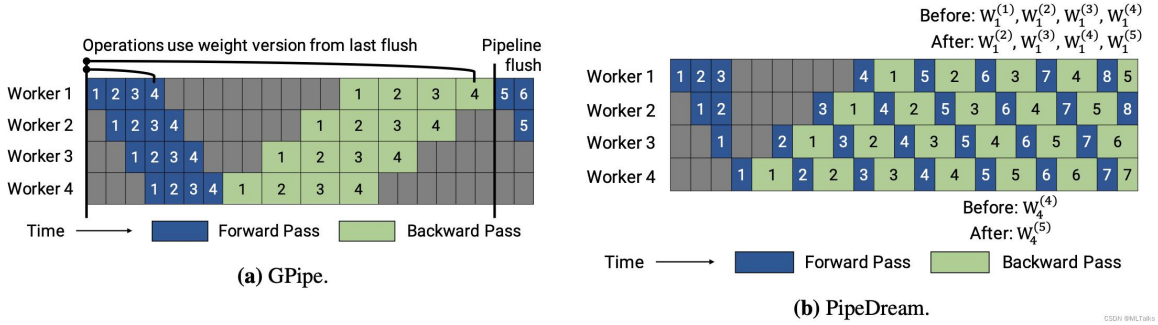


Figure 2: Combination of tensor and pipeline model parallelism (MP) used in this work for transformer-based models.

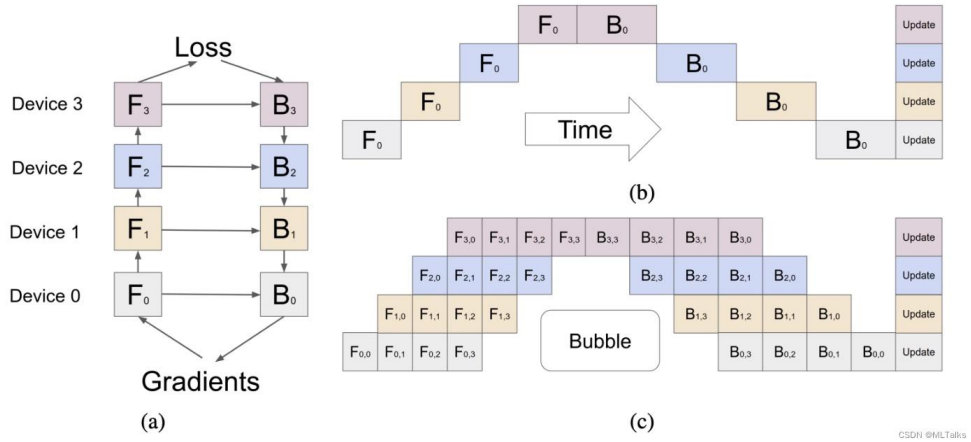


这就像 CPU 里的流水线，我们的模型训练也可以使用 pipeline 进行操作。当然，还有很多的操作和优化的地方，博客[15]进行了更宽泛的综述。



(a) GPipe.

(b) PipeDream.



(a)

(b)

(c)

下面这个图，是一项针对 GPU 中训练 AI 的时间分配的研究结果。

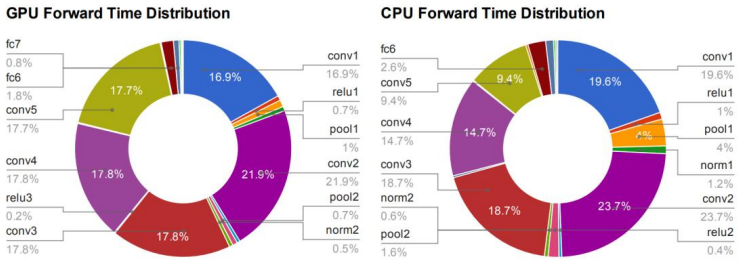
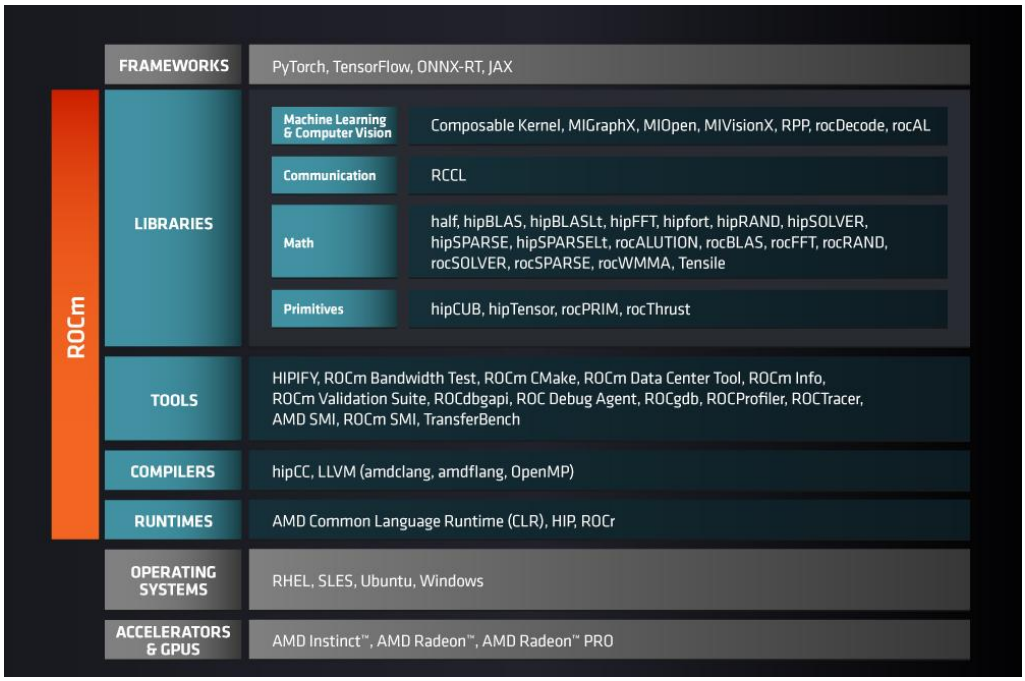


Figure 6.1: Computation time distribution of individual layers, on both GPUs and CPUs for the forward pass.

AMD：你好，我也要恰饭的

看了这么多 CUDA，AMD 有没有任何动作呢？其实也有的，那就是他们的 ROCm。但是我不喜欢 AMD，他们的软件栈文档又得重新学习，遇到的问题社区里很少有回答，并且很多旧 A 卡是使用不了 ROCm 的。但是我们也可以看一眼他们的架构，跟 CUDA 生态还是比较类似的。



总结：

本次 Project 我们简单学习了一下 CUDA 编程，比较了几个计算参数。同时我们读了几篇文章，深入了解了几个 GPU 的技术，了解了 GPU 目前的一些研究方向。虽然时间不足，但是毋庸置疑地是将来我还会跟它打交道。这次的 Project 学习的东西无疑是一个好基础。

主持人：对于计算机或者工程学专业的学生，你会给他们什么建议，来提高成功的机会？

黄仁勋：

我认为我的一大优势是，我期望值很低。我认为大多数斯坦福毕业生期望值很高。

期望值很高的人通常韧性很低。不幸的是，韧性在成功中很重要。我不知道如何教你们，除了我希望痛苦发生在你们身上。

我很幸运，我成长的环境中，我的父母为我们提供了成功的条件，但同时，也有足够的挫折和痛苦的机会。

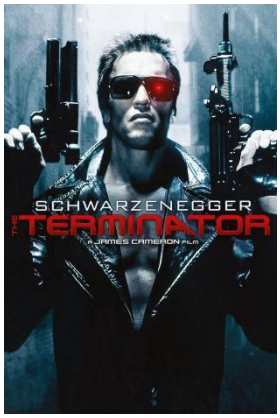
直到今天，我在我们公司里常常使用“痛苦和折磨”这个词。

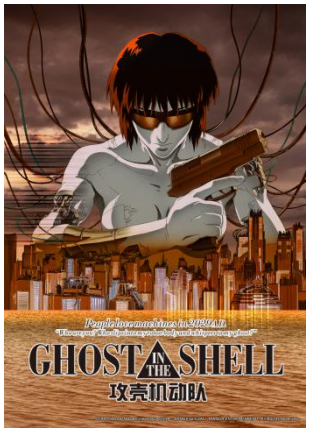
伟大不是智力，伟大来自于性格。聪明人需要经历痛苦才能打造出这样的性格。

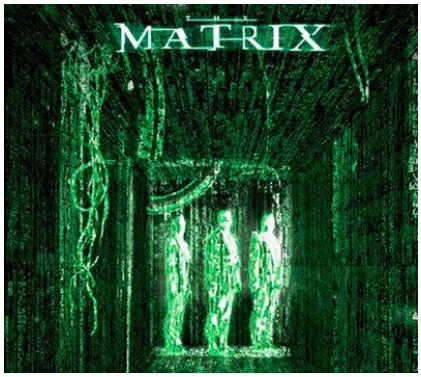
主持人：如果您能分享一条简短的建议给斯坦福，会是什么呢？

黄仁勋：拥有一个核心信念。每天都发自内心地检视目标竭尽全力追求、持之以恒地追求。和您爱的人一起，携手踏上正途。这就是 NVIDIA 的故事。

Is it over?
No, it is just beginning.







Reference

引用博客/文章

[1] CUDA 基础】3.2 理解线程束执行的本质(Part I) | 谭升的博客 (face2ai.com)
[2] CUDA Warp-Level 级原语 - 吴建明 wujianming - 博客园 (cnblogs.com)
[3] Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf (concurrent-rt.com)
[4] 【Linux】深入理解生产者消费者模型-阿里云开发者社区 (aliyun.com)
[5] Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf (concurrent-rt.com)
[6] Overview of NCCL — NCCL 2.21.5 documentation (nvidia.com)
[7] NVIDIA NCCL 源码学习（一）- 初始化及 ncclUniqueId 的产生-CSDN 博客
[8] Using CUDA Warp-Level Primitives | NVIDIA Technical Blog
[9] CUDA 进阶第二篇：巧用 PTX cuda ptx-CSDN 博客
[10] 用 NVIDIA CUDA 11 . 2 C ++编译器提高生产率和性能 - NVIDIA 技术博客
[11] NVIDIA CUDA Compiler Driver
[12] CUDA：NVCC 编译过程和兼容性详解 nvcc 把 cuda 代码转换成什么-CSDN 博客
[13] cuda - 如何让 nvcc CUDA 编译器进行更多优化？ - IT 工具网 (coder.work)
[14] What is ROCm? — ROCm Documentation (amd.com)
[15] 详解 MegatronLM 流水线模型并行训练(Pipeline Parallel) efficient large-scale language model training on g-CSDN 博客
[16] 解锁 vLLM：大语言模型推理的速度与效率双提升-腾讯云开发者社区-腾讯云 (tencent.com)
[17] njuhope/cuda sgemm (github.com)

引用学术文章

[18] K. Gupta, J. A. Stuart and J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads," 2012 Innovative Parallel Computing (InPar), San Jose, CA, USA, 2012, pp. 1-14, doi: 10.1109/InPar.2012.6339596.

[19] Stuart, J.A., & Owens, J.D. (2011). Efficient Synchronization Primitives for GPUs. ArXiv, abs/1110.4623.

[20] W. Wang, S. Guo, F. Yang and J. Chen, "GPU-Based Fast Minimum Spanning Tree Using Data Parallel Primitives," 2010 2nd International Conference on Information Engineering and Computer Science, Wuhan, China, 2010, pp. 1-4, doi: 10.1109/ICIECS.2010.5678261.

[21] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan primitives for GPU computing. In Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (GH '07). Eurographics Association, Goslar, DEU, 97–106.

[22] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Trans. Parallel Distrib. Syst. 31, 1 (Jan. 2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[23] He, J., & Zhai, J. (2024). FastDecode: High-Throughput GPU-Efficient LLM Serving using Heterogeneous Pipelines. [Preprint]. arXiv. Retrieved from https://arxiv.org/abs/2403.11421

[24] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, May 2009.

[25] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In Proceedings of High Performance Graphics 2009, pages 145–149, August 2009.

[26] Toledo, R.D., & Lévy, B. (2005). Extending the graphic pipeline with new GPU-accelerated primitives.

[27] Di Napoli, E., Fabregat-Traver, D., Quintana-Ortí, G., & Bientinesi, P. (2013, July 22). Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions.

[28] Shi, Y., Niranjan, U. N., Anandkumar, A., & Cecka, C. (2016, December). Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In 2016 IEEE 23rd International Conference on High Performance Computing (HiPC) (pp. 1-10). IEEE. doi:10.1109/hipc.2016.031

[29] 吴再龙,王利明,徐震,等.GPU 虚拟化技术及其安全问题综述[J].信息安全学报,2022,7(02):30-58. DOI:10.19363/J.cnki.cn10-1380/tn.2022.03.03.

[30] A. Dhakal, S. G. Kulkarni and K. K. Ramakrishnan, "Machine Learning at the Edge: Efficient Utilization of Limited CPU/GPU Resources by Multiplexing," 2020 IEEE 28th International Conference on Network Protocols (ICNP), Madrid, Spain, 2020, pp. 1-6, doi: 10.1109/ICNP49622.2020.9259361.

[31] Saumya, C., Sundararajah, K., & Kulkarni, M. (2021). CFM: SIMT Thread Divergence Reduction by Melding Similar Control-Flow Regions in GPGPU Programs. CoRR, abs/2107.05681. https://arxiv.org/abs/2107.05681

[32] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2020). Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv preprint arXiv:1909.08053.

[33] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv preprint arXiv:2309.06180.