# CUDA Programming

Rajat Ghosh

rghosh08.github.io

https://github.com/rghosh08/cuda

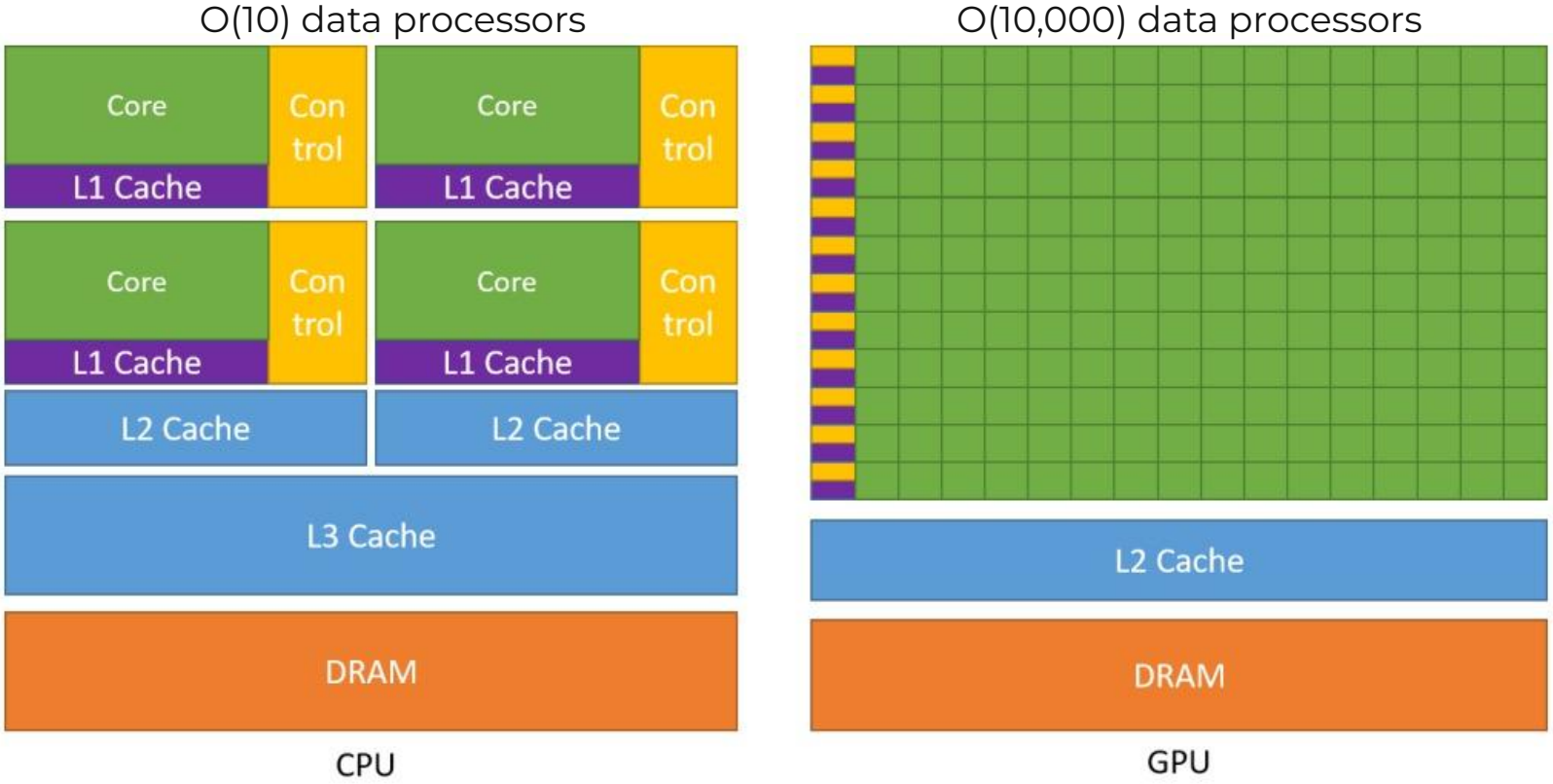
NUTANIX


- Preliminary
- Getting Started
- Matrix Multiplication
- Alexnet
- Map Reduce
- Similarity Search
- Checksum
- LLM Inference Paged Attention

# Compute Unified Device Architecture (CUDA)

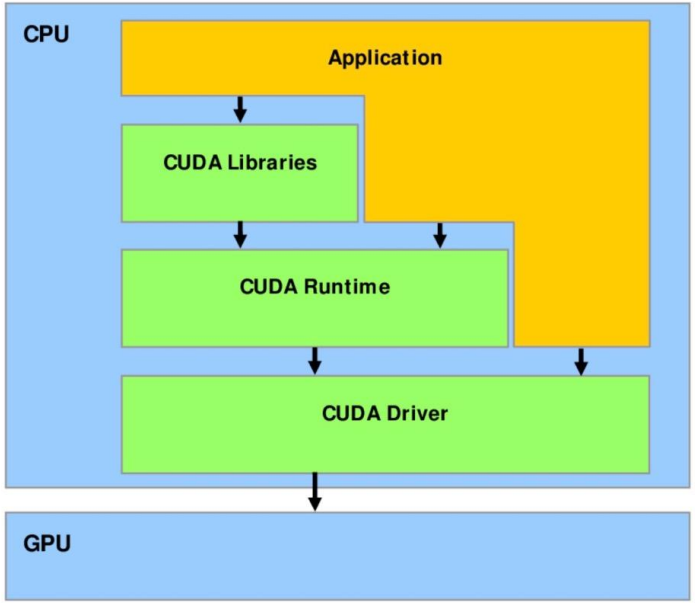
| Before CUDA (Pre-2006) | After CUDA |
| --- | --- |
| GPU for graphics programming with OpenGL/DirectX | GPU for general-purpose parallel computing with C/C++ API |

# GPU vs GPU



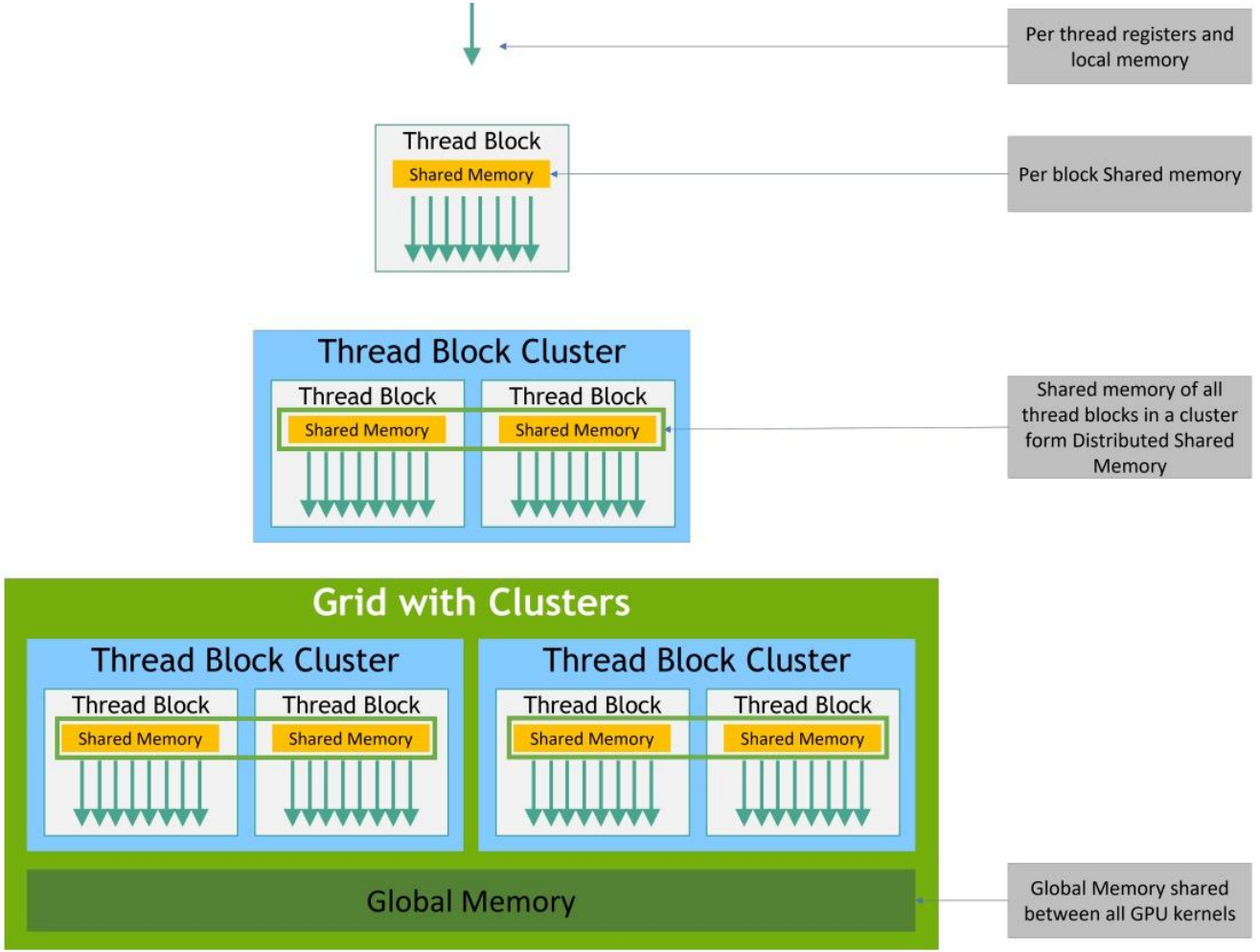

https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

# Programming Model



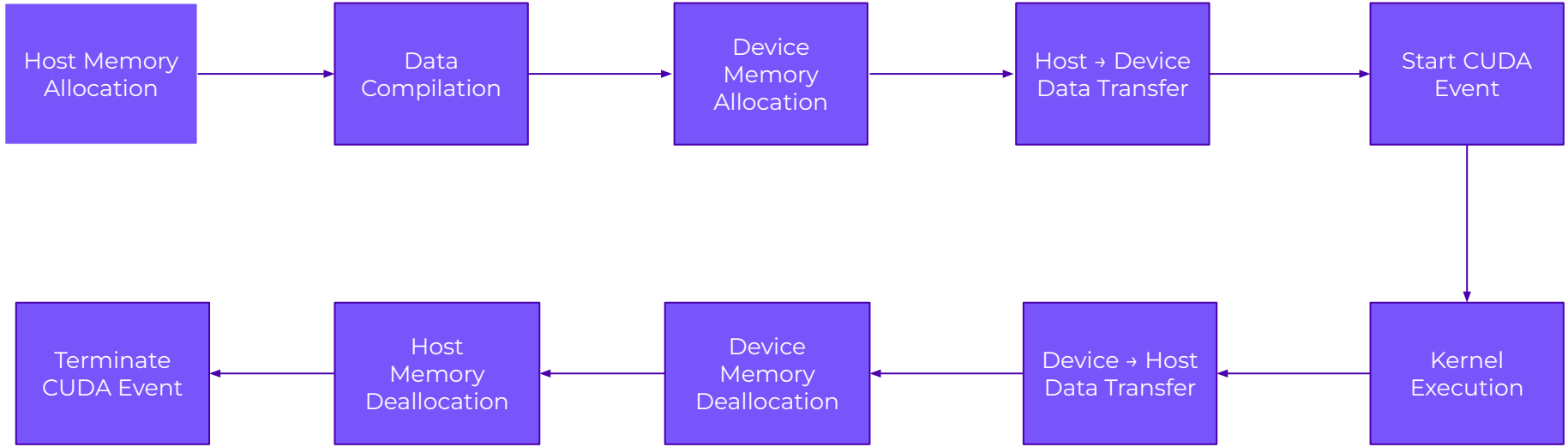

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

# Hierarchy


Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Global Memory
Global Memory shared between all GPU kernels

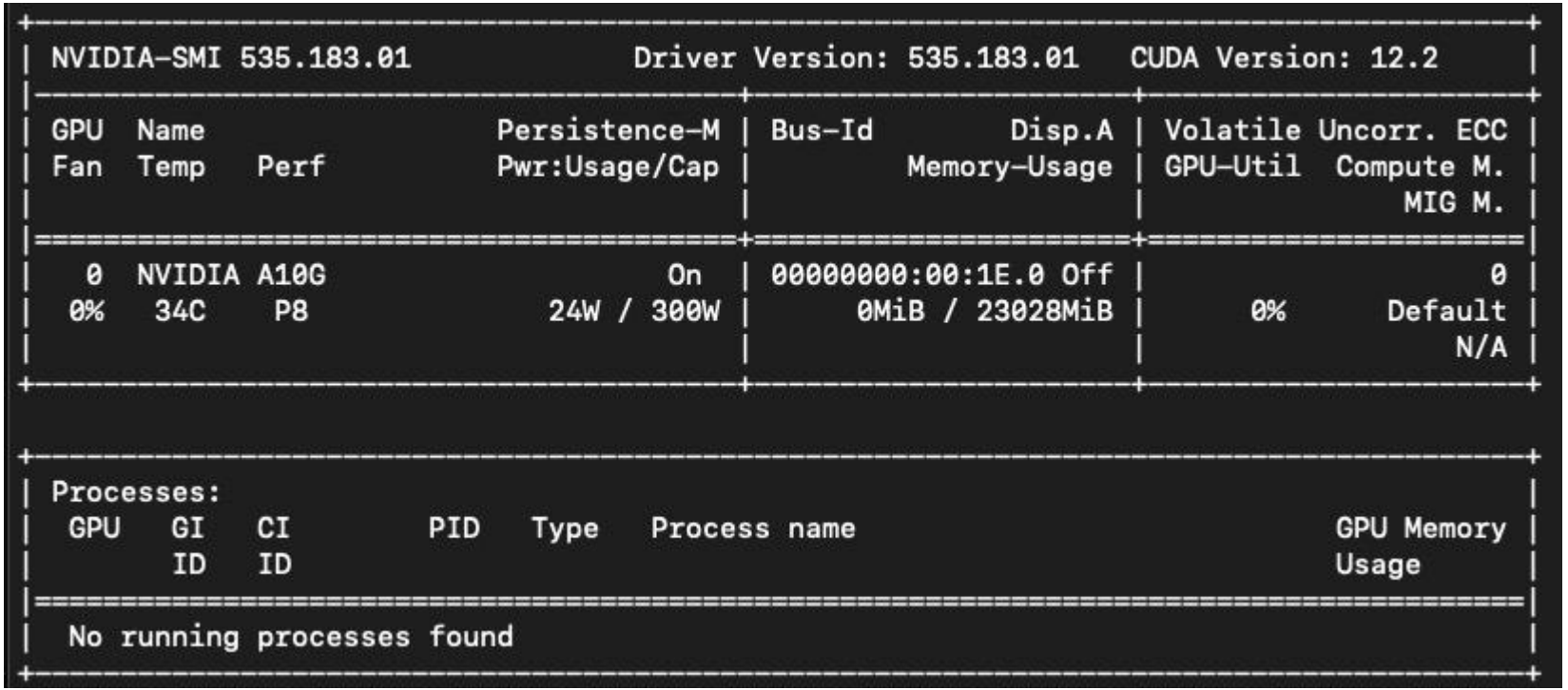
# CUDA Operations Model

- Break a task into tiny operations
- Run each tiny operations on a single GPU thread
- Coalesce the results from single threads into a final answer
- Host CPU as an orchestrator



# Experimental Setup

# CUDA Device Info



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version: 12.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name                     Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf              Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A10G                        On | 00000000:00:1E.0 Off |                    0 |
|  0%   34C    P8               24W / 300W |      0MiB / 23028MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

# CUDA Host Info

| Specification | Value |
| --- | --- |
| Physical CPU Cores | 8 |
| Total Threads | 16 |
| Memory Size | 64 GB |
| Disk Space | 1.1 TB |
| OS | Ubuntu 20.04.6 LTS |

# Getting Started

# Synchronization Issue

```cpp
#include <stdio.h>

// Kernel function to generate a markdown-formatted table showing thread indices
__global__ void generateThreadTable() {
    // Calculate global thread ID
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;

    // Print table header only from the first thread to avoid duplicate headers
    if (threadId == 0) {
        printf("| blockIdx.x | threadIdx.x | blockDim.x | threadId |\n");
        printf("|------------|-------------|------------|----------|\n");
    }

    // Synchronize all threads to ensure the header prints before any thread data
    __syncthreads();

    // Each thread prints its own indices and global ID in table format
    printf("| %-10d | %-11d | %-10d | %-8d |\n",
           blockIdx.x, threadIdx.x, blockDim.x, threadId);
}

int main() {
    // Launch kernel with 2 blocks, each containing 4 threads
    generateThreadTable<<<2, 4>>>();

    // Wait for all GPU tasks to complete before ending the program
    cudaDeviceSynchronize();

    return 0;
}
```

```
| 1          | 0           | 4          | 4        |
| 1          | 1           | 4          | 5        |
| 1          | 2           | 4          | 6        |
| 1          | 3           | 4          | 7        |
| blockIdx.x | threadIdx.x | blockDim.x | threadId |
|------------|-------------|------------|----------|
| 0          | 0           | 4          | 0        |
| 0          | 1           | 4          | 1        |
| 0          | 2           | 4          | 2        |
| 0          | 3           | 4          | 3        |
```

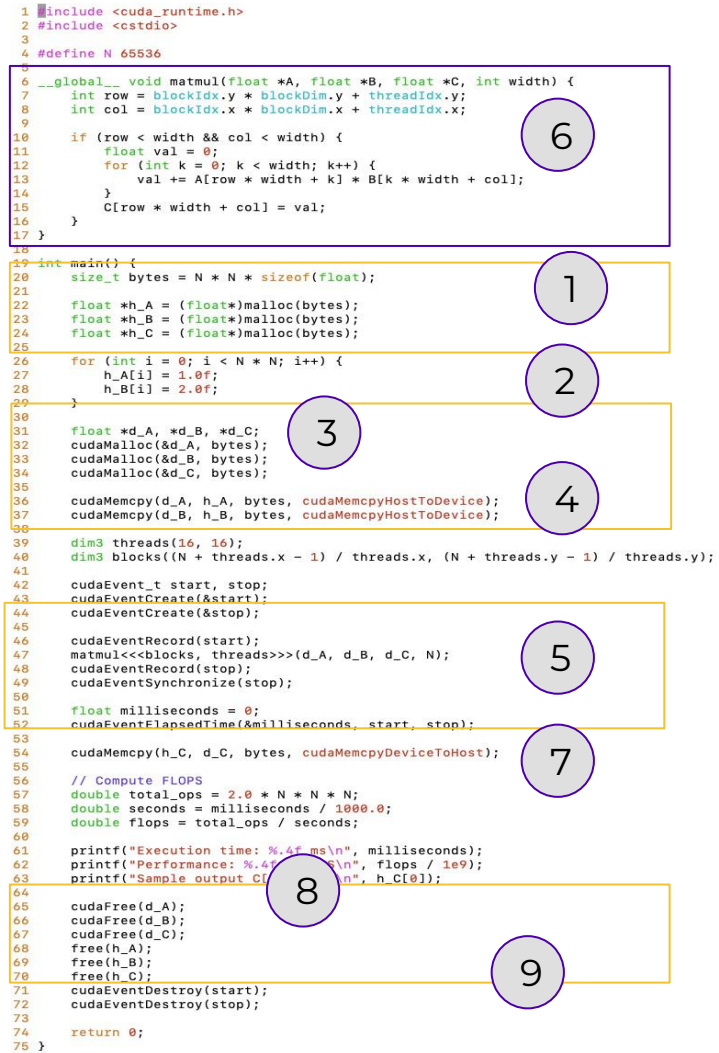
# Fixed by Changing Execution Order

```cpp
#include <stdio.h>

// Kernel function to generate thread data (no header printing here)
__global__ void generateThreadTable() {
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;

    printf("| %-10d | %-11d | %-10d | %-8d |\n",
           blockIdx.x, threadIdx.x, blockDim.x, threadId);
}

int main() {
    // Print table header from CPU (host)
    printf("| blockIdx.x | threadIdx.x | blockDim.x | threadId |\n");
    printf("|------------|-------------|------------|----------|\n");

    // Launch kernel
    generateThreadTable<<<2, 4>>>();

    // Wait for kernel to complete
    cudaDeviceSynchronize();

    return 0;
}
```

| blockIdx.x | threadIdx.x | blockDim.x | threadId |
|---|---|---|---|
| 1 | 0 | 4 | 4 |
| 1 | 1 | 4 | 5 |
| 1 | 2 | 4 | 6 |
| 1 | 3 | 4 | 7 |
| 0 | 0 | 4 | 0 |
| 0 | 1 | 4 | 1 |
| 0 | 2 | 4 | 2 |
| 0 | 3 | 4 | 3 |

# Matrix Multiplication

Task: Benchmark CUDA for Massive Scale Matrix Multiplication




| ID | Description |
|---|---|
| 1 | Host Memory Allocation |
| 2 | Data Compilation |
| 3 | Device Memory Allocation |
| 4 | Host→Device Memory Copy |
| 5 | Execution Invocation |
| 6 | Kernel Execution |
| 7 | Device→Host Memory Copy |
| 8 | Device Memory Deallocation |
| 9 | Host Memory Deallocation |

# Result

| Matrix Size | CUDA | CPP | CPP-OpenMP | CUDA Speedup |
|---|---|---|---|---|
| 1024x1024 | 1.1 ms | 6.9 s | 79 ms | ~70x |
| 2048x2048 | 8.3 ms | 181.9 s | 7.2 s | ~800x |
| 4096x4096 | 66 ms | 2391.27 s | 117 s | ~1772x |
| 8192x8192 | 527 ms | RIP | 2,130 s | ~4042x |
| 16384x16384 | 4.3 s | RIP | RIP | CUDA survives 268M |
| 32768x32768 | 36 s | RIP | RIP | CUDA survives 1B |
| 65536x65536 | OOM on 24GB VRAM | RIP | RIP | Memory Bound |

# Deep Neural Network Training with Alexnet

Task: Replicate Alexnet Training on CUDA

Link

# AlexNet with 60M Parameters (SOTA in 2012)

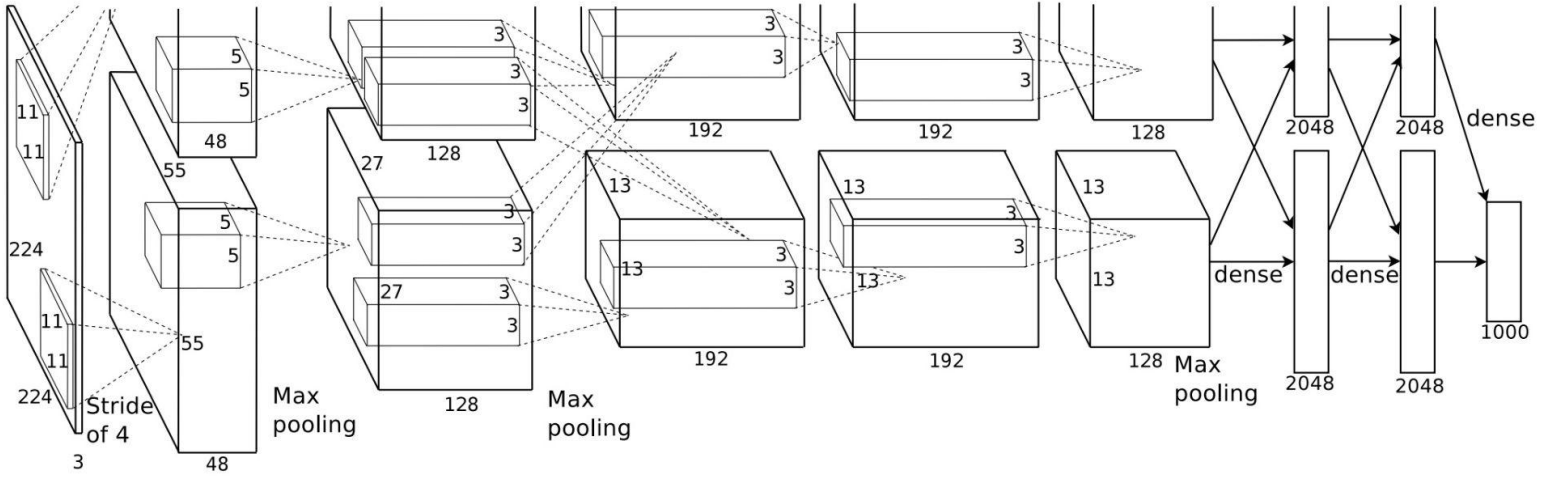



Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

# Architecture Implemented

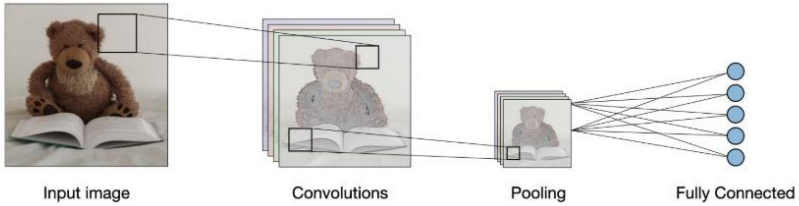
| Parameters | Original | Implemented |
|---|---|---|
| Input Size | 227 x 227 x 3 | 227 x 227 x 3 |
| No of Classes | 1000 | 1000 |
| Batch Size | 128 | 512 |
| Epoch | 90 | 10 |
| Training Samples | 1,281,167 | 5120 |
| Interactions per Epoch | 10,009 | 10 |
| Set Up | 2x NVIDIA GTX 580 | 1x NVIDIA A10 |
| Architecture | 5 Conv + 3 FC | 3 Conv + 3 FC |

# Data Generator on Host

```cpp
// Simple data loader
class SimpleDataLoader {
private:
    std::vector<float> batch_input;
    std::vector<int> batch_labels;
    std::mt19937 rng;
    int current_batch;
    int total_batches;

public:
    SimpleDataLoader() : rng(std::random_device{}()), current_batch(0) {
        total_batches = Config::SAMPLES_PER_EPOCH / Config::BATCH_SIZE;

        size_t input_size = Config::BATCH_SIZE * Config::INPUT_CHANNELS *
                            Config::INPUT_HEIGHT * Config::INPUT_WIDTH;
        batch_input.resize(input_size);
        batch_labels.resize(Config::BATCH_SIZE);

        std::cout << "DataLoader initialized: " << total_batches << " batches per epoch" << std::endl;
    }

    bool hasNextBatch() { return current_batch < total_batches; }
    void resetEpoch() { current_batch = 0; }
    int getCurrentBatch() const { return current_batch; }
    int getTotalBatches() const { return total_batches; }

    void generateBatch(float** d_input, int** d_labels) {
        if (!hasNextBatch()) return;

        // Generate random data
        std::uniform_real_distribution<float> input_dist(-1.0f, 1.0f);
        std::uniform_int_distribution<int> label_dist(0, Config::NUM_CLASSES - 1);

        for (size_t i = 0; i < batch_input.size(); ++i) {
            batch_input[i] = input_dist(rng);
        }

        for (size_t i = 0; i < batch_labels.size(); ++i) {
            batch_labels[i] = label_dist(rng);
        }

        // Copy to GPU
        CHECK_CUDA(cudaMemcpy(*d_input, batch_input.data(),
                              batch_input.size() * sizeof(float), cudaMemcpyHostToDevice));
        CHECK_CUDA(cudaMemcpy(*d_labels, batch_labels.data(),
                              batch_labels.size() * sizeof(int), cudaMemcpyHostToDevice));

        current_batch++;
    }
};
```

# Convolution on CUDA

```
// CUDA kernels
__global__ void simple_conv2d_kernel(float* input, float* weight, float* bias, float* output,
                                     int batch_size, int in_channels, int out_channels,
                                     int input_h, int input_w, int kernel_size, int stride, int padding) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int output_h = (input_h + 2 * padding - kernel_size) / stride + 1;
    int output_w = (input_w + 2 * padding - kernel_size) / stride + 1;
    int total_outputs = batch_size * out_channels * output_h * output_w;

    if (idx < total_outputs) {
        int b = idx / (out_channels * output_h * output_w);
        int oc = (idx / (output_h * output_w)) % out_channels;
        int oh = (idx / output_w) % output_h;
        int ow = idx % output_w;

        float sum = bias[oc];

        // Simple convolution (unoptimized but functional)
        for (int ic = 0; ic < in_channels; ic++) {
            for (int kh = 0; kh < kernel_size; kh++) {
                for (int kw = 0; kw < kernel_size; kw++) {
                    int ih = oh * stride - padding + kh;
                    int iw = ow * stride - padding + kw;

                    if (ih >= 0 && ih < input_h && iw >= 0 && iw < input_w) {
                        int input_idx = b * (in_channels * input_h * input_w) +
                                        ic * (input_h * input_w) + ih * input_w + iw;
                        int weight_idx = oc * (in_channels * kernel_size * kernel_size) +
                                         ic * (kernel_size * kernel_size) + kh * kernel_size + kw;
                        sum += input[input_idx] * weight[weight_idx];
                    }
                }
            }
        }

        output[idx] = fmaxf(0.0f, sum); // ReLU activation
    }
}
```




$$Y[i,j] = \sum_m \sum_n X[i \times s + m - p, j \times s + n - p] \cdot W[m,n] + b$$

# Simple Max Pooling Kernel

```cpp
__global__ void simple_maxpool_kernel(float* input, float* output,
                                      int batch_size, int channels, int input_h, int input_w,
                                      int kernel_size, int stride) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int output_h = (input_h - kernel_size) / stride + 1;
    int output_w = (input_w - kernel_size) / stride + 1;
    int total_outputs = batch_size * channels * output_h * output_w;

    if (idx < total_outputs) {
        int b = idx / (channels * output_h * output_w);
        int c = (idx / (output_h * output_w)) % channels;
        int oh = (idx / output_w) % output_h;
        int ow = idx % output_w;

        float max_val = -INFINITY;

        for (int kh = 0; kh < kernel_size; kh++) {
            for (int kw = 0; kw < kernel_size; kw++) {
                int ih = oh * stride + kh;
                int iw = ow * stride + kw;
                int input_idx = b * (channels * input_h * input_w) +
                                c * (input_h * input_w) + ih * input_w + iw;
                max_val = fmaxf(max_val, input[input_idx]);
            }
        }

        output[idx] = max_val;
    }
}
```

$$Y[i,j] = \max_{m,n}(X[i \times s + m,\ j \times s + n])$$

$$\text{ReLU}(x) = \max(0, x)$$

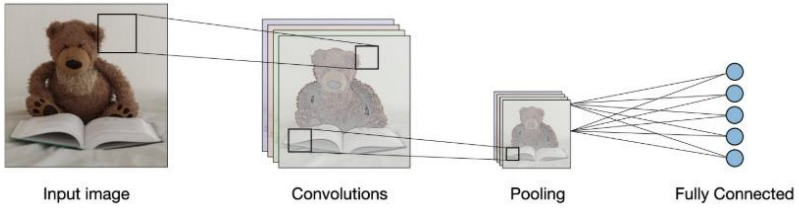
$$y = x \odot m$$

$$L = -\sum_{i=1} y_i \log(\hat{y}_i)$$

$$\theta_{t+1} = \theta_t - \eta/m \sum_{i=1} \nabla_\theta L(\theta_t; x^{(i)}, y^{(i)})$$

```cuda
__global__ void relu_kernel(float* input, float* output, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        output[idx] = fmaxf(0.0f, input[idx]);
    }
}

__global__ void dropout_kernel(float* input, float* output, int size, unsigned long long seed) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        // Simple deterministic dropout for demo
        if ((idx + (int)seed) % 2 == 0) {
            output[idx] = input[idx] * 2.0f;
        } else {
            output[idx] = 0.0f;
        }
    }
}

__global__ void cross_entropy_loss_kernel(float* logits, int* labels, float* loss,
                                          float* grad_output, int batch_size, int num_classes) {
    int batch_idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (batch_idx < batch_size) {
        int label = labels[batch_idx];
        int offset = batch_idx * num_classes;

        // Find max for numerical stability
        float max_val = logits[offset];
        for (int i = 1; i < num_classes; i++) {
            max_val = fmaxf(max_val, logits[offset + i]);
        }

        // Compute softmax denominator
        float sum_exp = 0.0f;
        for (int i = 0; i < num_classes; i++) {
            sum_exp += expf(logits[offset + i] - max_val);
        }

        // Loss
        float log_prob = logits[offset + label] - max_val - logf(sum_exp);
        atomicAdd(loss, -log_prob / batch_size);

        // Gradient
        for (int i = 0; i < num_classes; i++) {
            float softmax_val = expf(logits[offset + i] - max_val) / sum_exp;
            grad_output[offset + i] = (softmax_val - (i == label ? 1.0f : 0.0f)) / batch_size;
        }
    }
}

__global__ void sgd_update_kernel(float* weights, float* gradients, float learning_rate, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        weights[idx] -= learning_rate * gradients[idx];
    }
}
```

# Forward Pass and Loss Function




```
float computeLoss() {
    float* d_loss;
    CHECK_CUDA(cudaMalloc(&d_loss, sizeof(float)));
    CHECK_CUDA(cudaMemset(d_loss, 0, sizeof(float)));

    dim3 block(256);
    dim3 grid((Config::BATCH_SIZE + block.x - 1) / block.x);

    cross_entropy_loss_kernel<<<grid, block>>>(
        d_fc3_out, d_labels, d_loss, d_grad_fc3_out, Config::BATCH_SIZE, 1000);

    float host_loss;
    CHECK_CUDA(cudaMemcpy(&host_loss, d_loss, sizeof(float), cudaMemcpyDeviceToHost));
    CHECK_CUDA(cudaFree(d_loss));

    return host_loss;
}
```

```
public:
    void forward() {
        const float alpha = 1.0f, beta = 0.0f;

        // Conv1 + ReLU + Pool1 (simplified using custom kernels)
        dim3 block(256);
        int conv1_outputs = Config::BATCH_SIZE * 96 * 55 * 55;
        dim3 grid_conv1((conv1_outputs + block.x - 1) / block.x);

        simple_conv2d_kernel<<<grid_conv1, block>>>(
            d_input, d_conv1_w, d_conv1_b, d_conv1_out,
            Config::BATCH_SIZE, 3, 96, 227, 227, 11, 4, 2);

        // Pool1
        int pool1_outputs = Config::BATCH_SIZE * 96 * 27 * 27;
        dim3 grid_pool1((pool1_outputs + block.x - 1) / block.x);
        simple_maxpool_kernel<<<grid_pool1, block>>>(
            d_conv1_out, d_pool1_out, Config::BATCH_SIZE, 96, 55, 55, 3, 2);

        // FC1 (flatten pool1 and use cuBLAS)
        int fc1_input_size = 96 * 27 * 27;
        CHECK_CUBLAS(cublasSgemm(cublas, CUBLAS_OP_T, CUBLAS_OP_N,
            4096, Config::BATCH_SIZE, fc1_input_size,
            &alpha, d_fc1_w, fc1_input_size, d_pool1_out, fc1_input_size,
            &beta, d_fc1_out, 4096));

        // Add bias and apply ReLU + Dropout
        for (int b = 0; b < Config::BATCH_SIZE; ++b) {
            CHECK_CUBLAS(cublasSaxpy(cublas, 4096, &alpha, d_fc1_b, 1, d_fc1_out + b * 4096, 1));
        }

        dim3 grid_fc1((Config::BATCH_SIZE * 4096 + block.x - 1) / block.x);
        relu_kernel<<<grid_fc1, block>>>(d_fc1_out, d_fc1_out, Config::BATCH_SIZE * 4096);
        dropout_kernel<<<grid_fc1, block>>>(d_fc1_out, d_fc1_out, Config::BATCH_SIZE * 4096, dropout_seed++);

        // FC2
        CHECK_CUBLAS(cublasSgemm(cublas, CUBLAS_OP_T, CUBLAS_OP_N,
            4096, Config::BATCH_SIZE, 4096,
            &alpha, d_fc2_w, 4096, d_fc1_out, 4096,
            &beta, d_fc2_out, 4096));

        for (int b = 0; b < Config::BATCH_SIZE; ++b) {
            CHECK_CUBLAS(cublasSaxpy(cublas, 4096, &alpha, d_fc2_b, 1, d_fc2_out + b * 4096, 1));
        }

        relu_kernel<<<grid_fc1, block>>>(d_fc2_out, d_fc2_out, Config::BATCH_SIZE * 4096);
        dropout_kernel<<<grid_fc1, block>>>(d_fc2_out, d_fc2_out, Config::BATCH_SIZE * 4096, dropout_seed++);

        // FC3
        CHECK_CUBLAS(cublasSgemm(cublas, CUBLAS_OP_T, CUBLAS_OP_N,
            1000, Config::BATCH_SIZE, 4096,
            &alpha, d_fc3_w, 4096, d_fc2_out, 4096,
            &beta, d_fc3_out, 1000));

        for (int b = 0; b < Config::BATCH_SIZE; ++b) {
            CHECK_CUBLAS(cublasSaxpy(cublas, 1000, &alpha, d_fc3_b, 1, d_fc3_out + b * 1000, 1));
        }
    }
```

# Backward Pass

```
void backward() {
    // Simplified backward pass - only update FC3 weights
    const float alpha = 1.0f, beta = 0.0f;

    // FC3 weight gradients
    CHECK_CUBLAS(cublasSgemm(cublas, CUBLAS_OP_N, CUBLAS_OP_T,
        4096, 1000, Config::BATCH_SIZE,
        &alpha, d_fc2_out, 4096, d_grad_fc3_out, 1000,
        &beta, d_grad_fc3_w, 4096));

    // FC3 bias gradients
    CHECK_CUBLAS(cublasSgemv(cublas, CUBLAS_OP_T,
        Config::BATCH_SIZE, 1000,
        &alpha, d_grad_fc3_out, Config::BATCH_SIZE,
        d_ones, 1, &beta, d_grad_fc3_b, 1));
}

void updateWeights() {
    dim3 block(256);

    // Update FC3 weights
    dim3 grid_w((1000 * 4096 + block.x - 1) / block.x);
    sgd_update_kernel<<<grid_w, block>>>(d_fc3_w, d_grad_fc3_w, learning_rate, 1000 * 4096);

    dim3 grid_b((1000 + block.x - 1) / block.x);
    sgd_update_kernel<<<grid_b, block>>>(d_fc3_b, d_grad_fc3_b, learning_rate, 1000);
}
```

# Cuda vs CPP

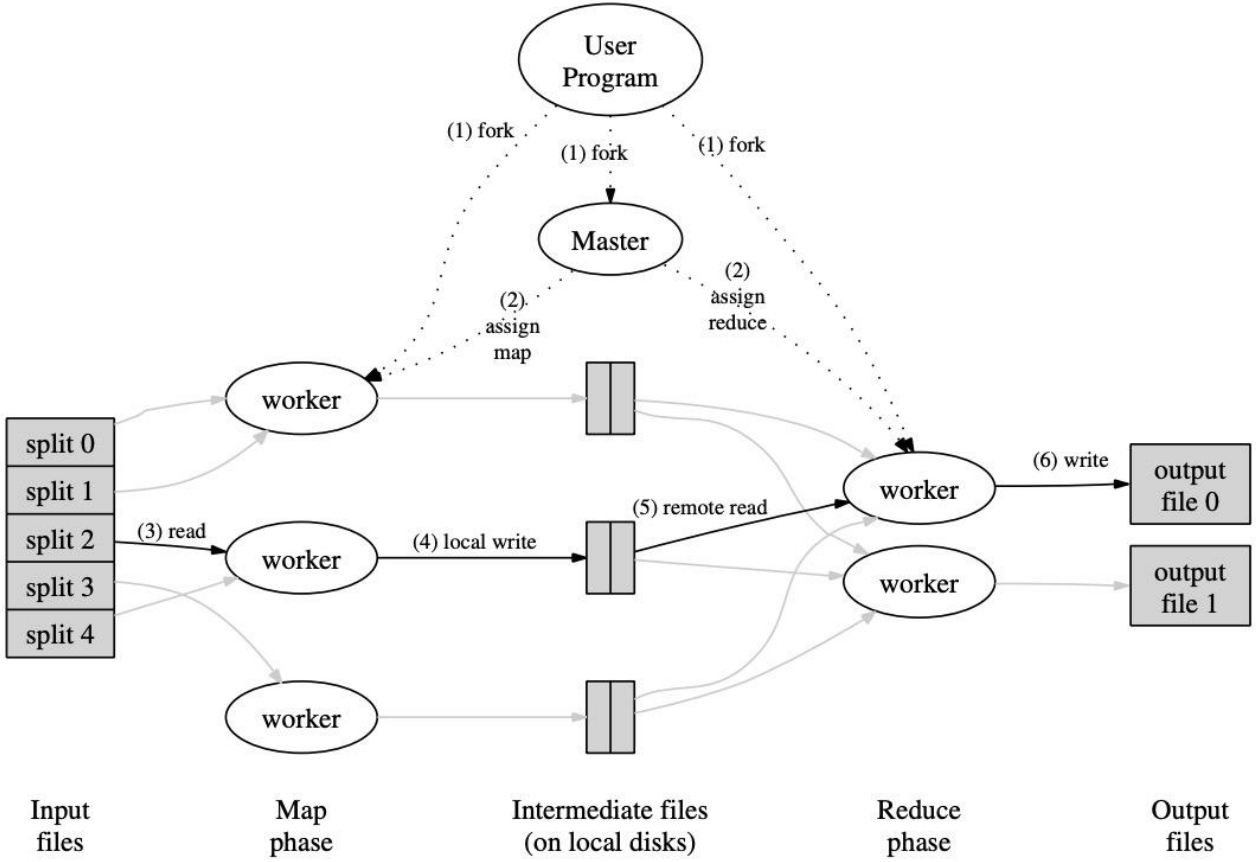
| CUDA | CPP-OpenMP |
| --- | --- |
| === Training Summary ===<br>Total Training Time: 5 minutes<br>Average Time per Epoch: 0.5 minutes<br>=== Loss Progression ===<br>Epoch  1: 9.5615 (33.0000s)<br>Epoch  2: 8.1356 (33.0000s)<br>Epoch  3: 7.8586 (33.0000s)<br>Epoch  4: 7.8395 (33.0000s)<br>Epoch  5: 7.7629 (33.0000s)<br>Epoch  6: 7.7826 (33.0000s)<br>Epoch  7: 7.7677 (33.0000s)<br>Epoch  8: 7.7502 (33.0000s)<br>Epoch  9: 7.7596 (33.0000s)<br>Epoch 10: 7.7541 (33.0000s)<br>=== Performance Metrics ===<br>Initial Loss: 9.5615<br>Final Loss: 7.7541<br>Peak GPU Memory: 2486 MB | === Training Summary ===<br>Total Training Time: 20 minutes<br>Average Time per Epoch: 2.0 minutes<br>=== Loss Progression ===<br>Epoch  1: 23.0079 (125.0000s)<br>Epoch  2: 9.5457 (118.0000s)<br>Epoch  3: 6.9092 (125.0000s)<br>Epoch  4: 6.9085 (125.0000s)<br>Epoch  5: 6.9091 (125.0000s)<br>Epoch  6: 6.9095 (118.0000s)<br>Epoch  7: 6.9080 (125.0000s)<br>Epoch  8: 6.9094 (127.0000s)<br>Epoch  9: 6.9234 (125.0000s)<br>Epoch 10: 6.9079 (126.0000s)<br>=== Performance Metrics ===<br>Initial Loss: 23.0079<br>Final Loss: 6.9079<br>OpenMP Threads Used: 16<br>CPU Architecture: Multi-core parallel training<br>=== Performance Comparison ===<br>CPU Training (this run): 20 minutes<br>Expected GPU Training: 4-6 minutes (A10 GPU)<br>CPU vs GPU Speedup: ~4x faster on GPU<br>=== OpenMP Optimization Analysis ===<br>• Convolution parallelized across output elements |

# Energy-Aware Training

| Epoch | Temperature ($^0$C) |
|---|---|
| 10 | 46 |
| 30 | 48 |
| 100 | 52 |
| 300 | 66 |
| 1000 | 86 |

# Map Reduce

Task: Benchmark massive scale term frequency computation on CUDA




https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

# Kernel Execution for Term Frequency Computation

```
// Custom MapReduce kernels
__global__ void map_kernel(int* input, KeyValue* mapped, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        mapped[idx].key = input[idx];
        mapped[idx].value = 1;
    }
}

__global__ void reduce_kernel(KeyValue* input, KeyValue* output, int* counts, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n) {
        int key = input[idx].key;
        atomicAdd(&counts[key], 1);
    }
}

__global__ void compact_results(int* counts, KeyValue* output, int max_keys, int* result_count) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < max_keys && counts[idx] > 0) {
        int pos = atomicAdd(result_count, 1);
        output[pos].key = idx;
        output[pos].value = counts[idx];
    }
}
```

Generate key value pairs {‘integer:1’}

Generate value sum for each integer

Skipping zero frequency integers

# Host Orchestration in Term Frequency Computation

```
void custom_mapreduce(int* data, int n, KeyValue* results, int* num_results) {
    KeyValue *d_mapped, *d_results;
    int *d_data, *d_counts, *d_result_count;

    // Allocate memory
    CUDA_CHECK(cudaMalloc(&d_mapped, n * sizeof(KeyValue)));
    CUDA_CHECK(cudaMalloc(&d_results, MAX_WORDS * sizeof(KeyValue)));
    CUDA_CHECK(cudaMalloc(&d_data, n * sizeof(int)));
    CUDA_CHECK(cudaMalloc(&d_counts, MAX_WORDS * sizeof(int)));
    CUDA_CHECK(cudaMalloc(&d_result_count, sizeof(int)));

    // Copy data to device
    CUDA_CHECK(cudaMemcpy(d_data, data, n * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemset(d_counts, 0, MAX_WORDS * sizeof(int)));
    CUDA_CHECK(cudaMemset(d_result_count, 0, sizeof(int)));

    // Map phase
    int grid_size = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
    map_kernel<<<grid_size, BLOCK_SIZE>>>(d_data, d_mapped, n);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Reduce phase
    reduce_kernel<<<grid_size, BLOCK_SIZE>>>(d_mapped, d_results, d_counts, n);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Compact results
    int count_grid = (MAX_WORDS + BLOCK_SIZE - 1) / BLOCK_SIZE;
    compact_results<<<count_grid, BLOCK_SIZE>>>(d_counts, d_results, MAX_WORDS, d_result_count);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Copy results back
    CUDA_CHECK(cudaMemcpy(num_results, d_result_count, sizeof(int), cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(results, d_results, (*num_results) * sizeof(KeyValue), cudaMemcpyDeviceToHost));

    // Cleanup
    cudaFree(d_mapped);
    cudaFree(d_results);
    cudaFree(d_data);
    cudaFree(d_counts);
    cudaFree(d_result_count);
}
```

# Host Orchestration with Thrust


```
void thrust_mapreduce(int* data, int n, KeyValue* results, int* num_results) {
    thrust::device_vector<int> d_words(data, data + n);
    thrust::device_vector<int> d_counts(d_words.size(), 1);

    // Sort by key for grouping
    thrust::sort_by_key(d_words.begin(), d_words.end(), d_counts.begin());

    // Reduce by key
    thrust::device_vector<int> unique_words(n);
    thrust::device_vector<int> word_counts(n);

    auto end = thrust::reduce_by_key(d_words.begin(), d_words.end(), d_counts.begin(),
                                     unique_words.begin(), word_counts.begin());

    *num_results = end.first - unique_words.begin();

    // Copy results
    for (int i = 0; i < *num_results; i++) {
        results[i].key = unique_words[i];
        results[i].value = word_counts[i];
    }
}
```

# Result

| Size | Time | Memory Usage |
|---|---|---|
| 1M | 149 ms | 0.0037 GB \| 21.98 GB |
| 10M | 159 ms | 0.037 GB \| 21.98 GB |
| 100M | 250 ms | 0.372 GB \| 21.98 GB |
| 1B | 1.2 s | 3.72 GB \| 21.98 GB |
| 1.2B | 1.5 s | 4.47 GB \| 21.98 GB |
| 1.5B | OOM | 5.58 GB \| 21.98 GB |
| 2B | OOM | 7.45 GB \| 21.98 GB |
| 3B | OOM | 11.2 GB \| 21.98 GB |
| 5B | OOM | 18.2 GB \| 21.98 GB |
| 10 B | Failure | 37.2 GB \| 21.98 GB |

# Massive Scale Similarity Search

Task: Benchmark massive scale distance computation on CUDA

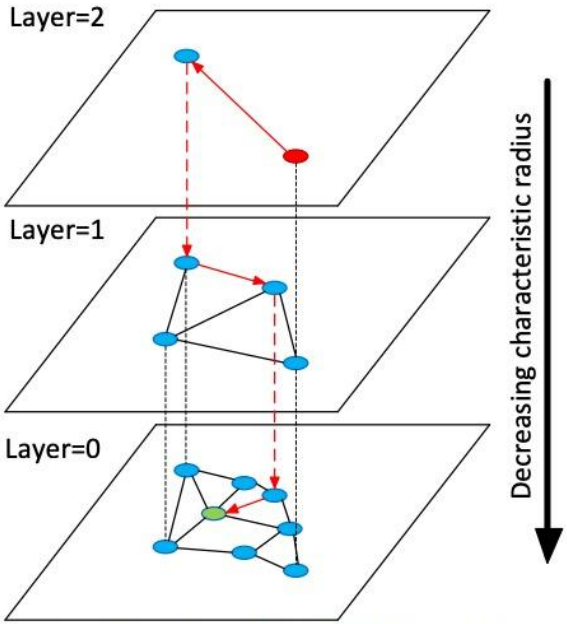



Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

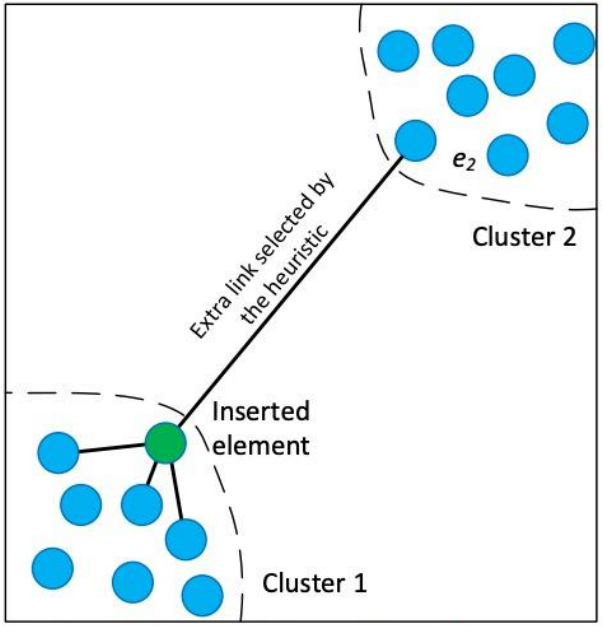



Fig. 2. Illustration of the heuristic used to select the graph neighbors for two isolated clusters. A new element is inserted on the boundary of Cluster 1. All of the closest neighbors of the element belong to the Cluster 1, thus missing the edges of Delaunay graph between the clusters. The heuristic, however, selects element $e_2$ from Cluster 2, thus, maintaining the global connectivity in case the inserted element is the closest to $e_2$ compared to any other element from Cluster 1.

$$L = k\text{-argmin}_{i=0:\ell} \|x - y_i\|_2,$$

https://arxiv.org/pdf/1603.09320

# L2 Norm Computation on GPU

```
__global__ void compute_distances(float *nodes, float *query, float *distances, int dim, size_t num_nodes) {
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_nodes) {
        float distance = 0.0f;
        for (int d = 0; d < dim; d++) {
            float diff = nodes[idx * dim + d] - query[d];
            distance += diff * diff;
        }
        distances[idx] = sqrtf(distance);
    }
}
```

# Host Orchestration

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <num_nodes> <dim>\n", argv[0]);
        return EXIT_FAILURE;
    }

    size_t num_nodes = atoll(argv[1]);
    int dim = atoi(argv[2]);

    printf("Allocating host memory...\n");
    float *nodes = (float*)malloc(num_nodes * dim * sizeof(float));
    float *query = (float*)malloc(dim * sizeof(float));
    float *distances = (float*)malloc(num_nodes * sizeof(float));

    printf("Initializing data...\n");
    for (size_t i = 0; i < num_nodes * dim; i++) nodes[i] = rand() / (float)RAND_MAX;
    for (int i = 0; i < dim; i++) query[i] = rand() / (float)RAND_MAX;

    float *d_nodes, *d_query, *d_distances;

    printf("Allocating GPU memory...\n");
    cudaMalloc(&d_nodes, num_nodes * dim * sizeof(float));
    cudaMalloc(&d_query, dim * sizeof(float));
    cudaMalloc(&d_distances, num_nodes * sizeof(float));

    printf("Transferring data to GPU...\n");
    cudaMemcpy(d_nodes, nodes, num_nodes * dim * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_query, query, dim * sizeof(float), cudaMemcpyHostToDevice);

    printGPUMemoryUsage("After allocations & transfers");

    int threadsPerBlock = 256;
    int blocksPerGrid = (num_nodes + threadsPerBlock - 1) / threadsPerBlock;

    printf("Starting CUDA kernel execution...\n");
    auto start = std::chrono::high_resolution_clock::now();

    compute_distances<<<blocksPerGrid, threadsPerBlock>>>(d_nodes, d_query, d_distances, dim, num_nodes);
    cudaDeviceSynchronize();

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> elapsed = end - start;

    printf("CUDA kernel execution completed.\n");
    printf("Kernel Execution Time: %.3f ms\n", elapsed.count());
```

| Size | CUDA | CUDA Memory | CPP OpenMP |
|---|---|---|---|
| 1M | 3 ms | 0.73 GB | 15 ms |
| 10M | 24 ms | 5 GB | 140 ms |
| 20 M | 49 ms | 9.8 GB | 279 ms |
| 30 M | 73 ms | 14.5 GB | 418 ms |
| 40M | 97 ms | 19 GB | 555  ms |
| 45M | 109 ms | 21.8 GB | 624 ms |

# Checksum Computation

Task: Benchmark massive scale checksum computation for 16MB Data on CUDA

## Mathematical Definition

**Checksum** = $\sum_{i=0}^{n-1} data[i] = data[0] + data[1] + \cdots + data[n-1]$

## Parallel Reduction Algorithm

### Block-Level Reduction

For a block with B threads and shared memory array S:

**Algorithm: Parallel Reduction within Block**

1. **Require:** Block size B = 2^k for some integer k
2. **Require:** Shared memory array S[0..B-1]
3. **Require:** Thread ID tid $\in$ [0, B-1]
4. Initialize: S[tid] $\leftarrow$ data[globalIdx]
5. **syncthreads()**
6. **for** stride = B/2; stride > 0; stride = stride/2 **do**
7. **if** tid < stride **then**
8. S[tid] $\leftarrow$ S[tid] + S[tid + stride]
9. **end if**
10. **syncthreads()**
11. **end for**
12. **if** tid = 0 **then**
13. **atomicAdd**(globalChecksum, S[0])
14. **end if**

## Tree-Based Reduction Pattern

The reduction follows a binary tree pattern:

**Level 0:** $S[i] = data[i] \quad \forall i \in [0, B-1]$

**Level 1:** $S[i] = S[i] + S[i + B/2] \quad \forall i \in [0, B/2-1]$

**Level 2:** $S[i] = S[i] + S[i + B/4] \quad \forall i \in [0, B/4-1]$

$\vdots$

**Level k:** $S[0] = S[0] + S[1]$

## Multi-Block Global Reduction

For G blocks, each contributing partial sum $P_j$:

$P_j = \sum_{i=j \cdot B}^{(j+1) \cdot B-1} data[i]$ for block j

**Checksum** = $\sum_{j=0}^{G-1} P_j = \sum_{j=0}^{G-1} \sum_{i=j \cdot B}^{(j+1) \cdot B-1} data[i]$

# Complexity Analysis

## Time Complexity

$T_{block} = O(\log_2 B)$ per block

$T_{total} = O(\log_2 B)$ (parallel across blocks)

```cuda
// CUDA Kernel: Compute simple checksum
__global__ void computeChecksum(unsigned char* data, size_t size, unsigned int* checksum) {
    __shared__ unsigned int shared_sum[BLOCK_SIZE];

    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int tid = threadIdx.x;

    // Initialize shared memory
    shared_sum[tid] = (idx < size) ? data[idx] : 0;
    __syncthreads();

    // Parallel reduction within block
    for (int stride = BLOCK_SIZE / 2; stride > 0; stride /= 2) {
        if (tid < stride)
            shared_sum[tid] += shared_sum[tid + stride];
        __syncthreads();
    }

    // Atomic add block result to global checksum
    if (tid == 0)
        atomicAdd(checksum, shared_sum[0]);
}
```

```cpp
int main() {
    const size_t dataSize = 1 << 24; // 16 MB
    unsigned char* h_data = (unsigned char*)malloc(dataSize);

    // Initialize data (simulate read from storage)
    for (size_t i = 0; i < dataSize; ++i)
        h_data[i] = rand() % 256;

    unsigned char* d_data;
    unsigned int* d_checksum;
    unsigned int h_checksum = 0;

    cudaMalloc(&d_data, dataSize);
    cudaMalloc(&d_checksum, sizeof(unsigned int));
    cudaMemcpy(d_data, h_data, dataSize, cudaMemcpyHostToDevice);
    cudaMemset(d_checksum, 0, sizeof(unsigned int));

    dim3 block(BLOCK_SIZE);
    dim3 grid((dataSize + BLOCK_SIZE - 1) / BLOCK_SIZE);

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

    cudaEventRecord(start);
    computeChecksum<<<grid, block>>>(d_data, dataSize, d_checksum);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds;
    cudaEventElapsedTime(&milliseconds, start, stop);

    cudaMemcpy(&h_checksum, d_checksum, sizeof(unsigned int), cudaMemcpyDeviceToHost);

    printf("Checksum: %u\n", h_checksum);
    printf("Processing Time: %.4f ms\n", milliseconds);

    // Cleanup
    cudaFree(d_data);
    cudaFree(d_checksum);
    free(h_data);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

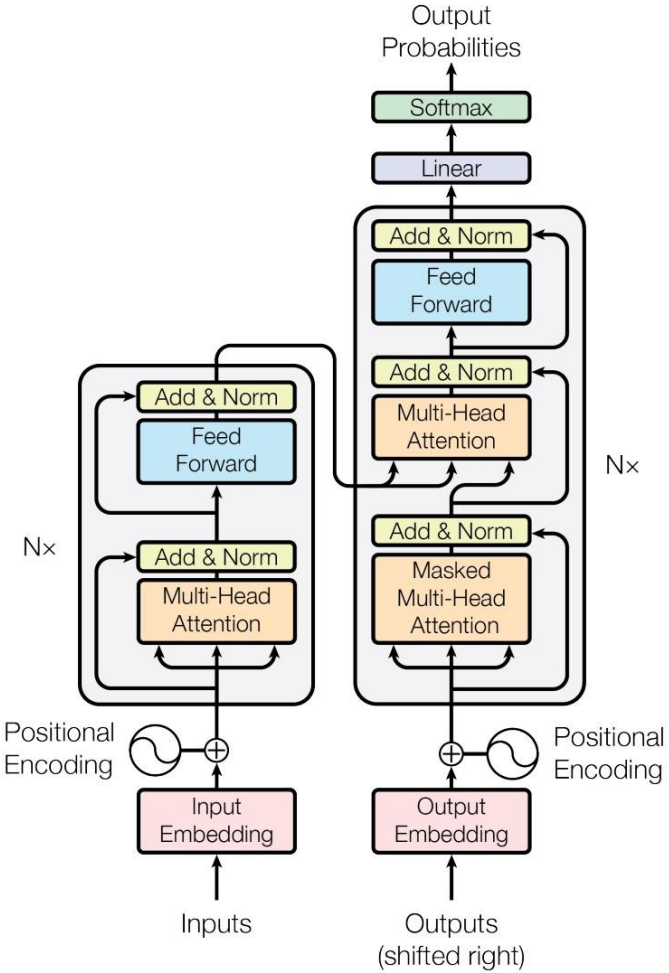
*// Assuming BLOCK_SIZE = 256:*
dataSize = 16,777,216 bytes
BLOCK_SIZE = 256 threads per block
grid_size = (16,777,216 + 256 - 1) / 256 = 16,777,471 / 256 = 65,536 blocks

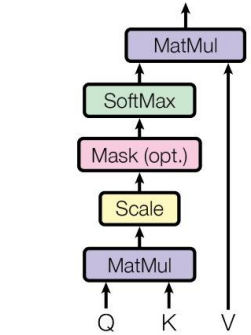
| Block Size | CUDA | CPP | CUDA Speed up |
|---|---|---|---|
| 256 | 0.3512 ms | 13.4741 ms | 38x |
| 512 | 0.3891 ms | 14.2124 ms | 36x |
| 1024 | 0.5335 ms | 16.5069 ms | 30x |

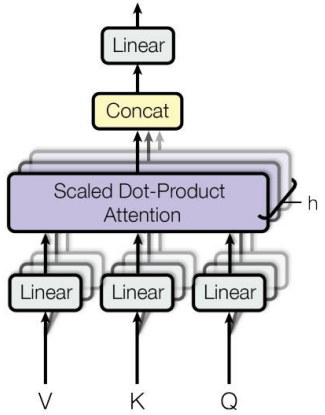
# LLM Inference with Paged Attention

Task: Implement Paged Attention from Scratch on CUDA

https://arxiv.org/pdf/1706.03762



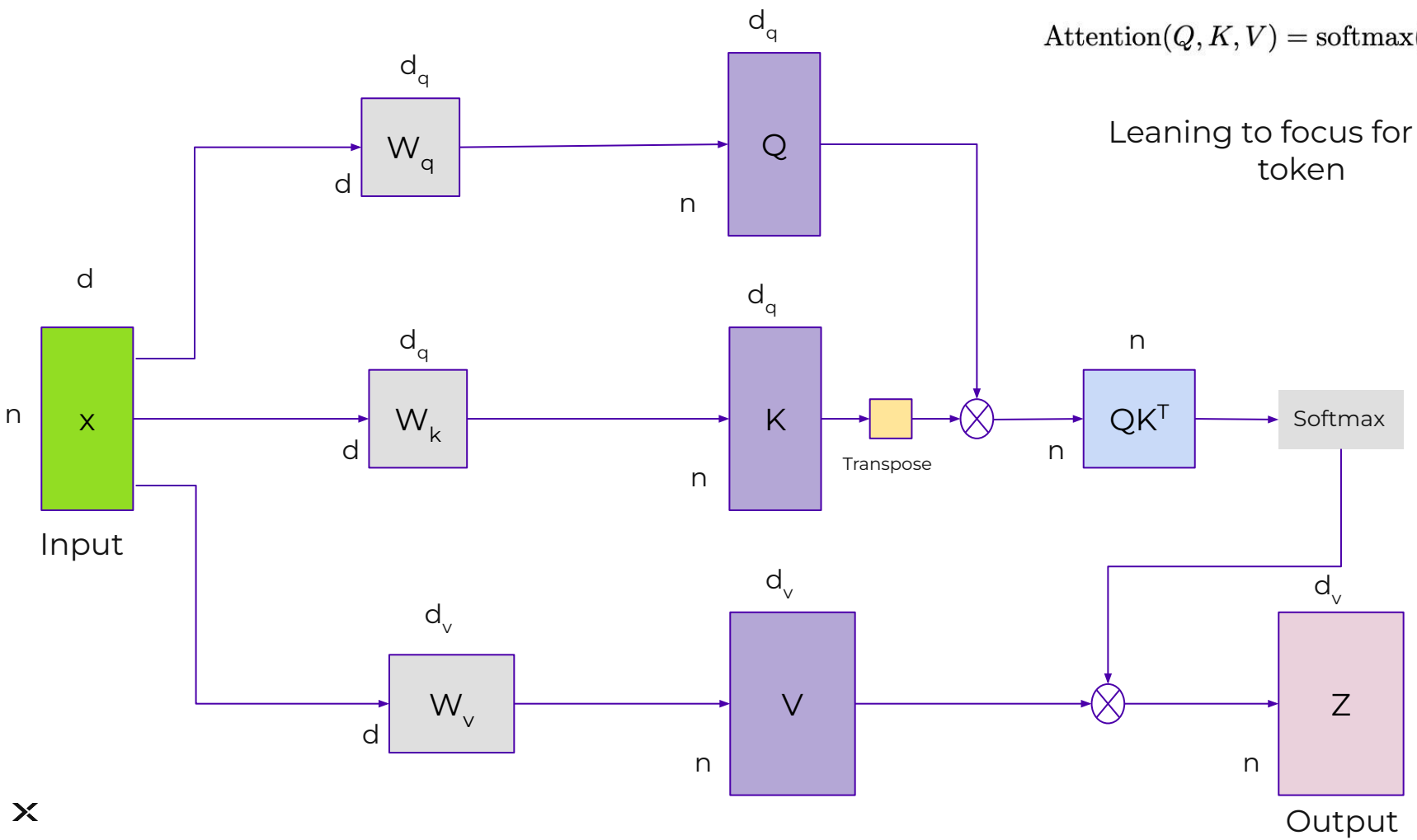



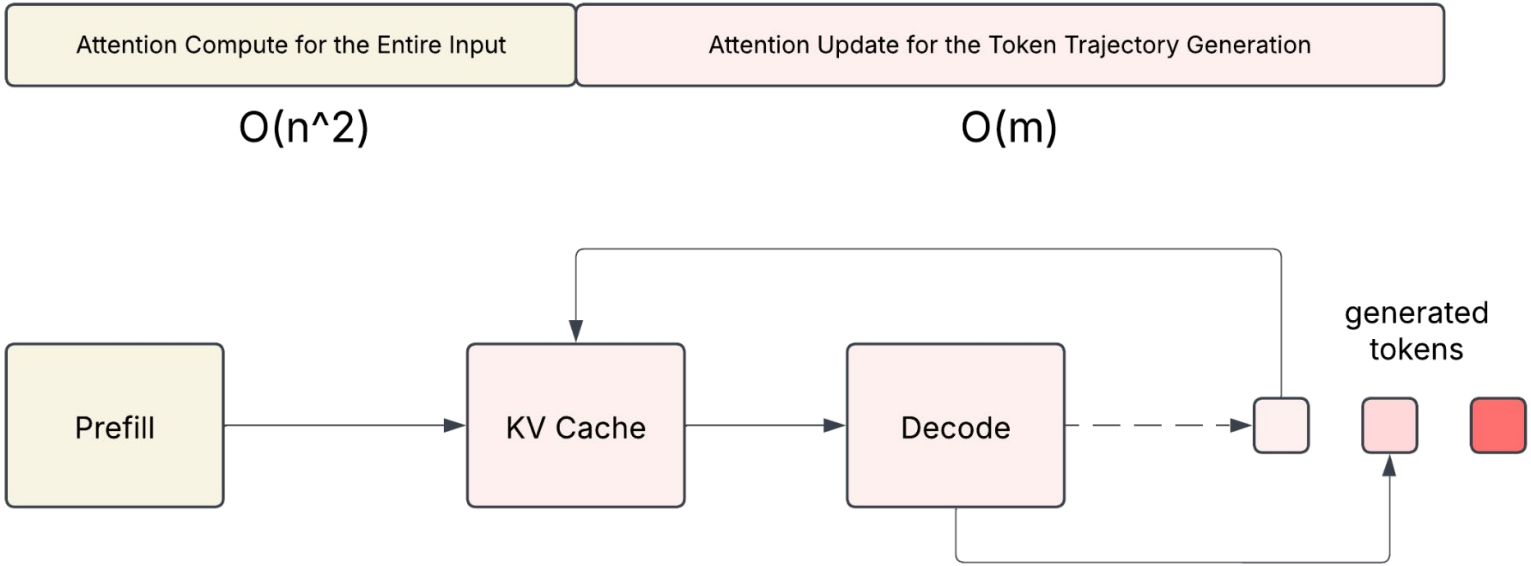

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Leaning to focus for each token






Paged Attention: KV Cache represented as a logical KV blocks

## LLM Inference Without KV Cache

context = initial_prompt_tokens

for t in range(prompt_length + 1, max_length):
    Q = W_q x(context)  [ $ndd_q$ ]
    K = W_k x (context)  [ $ndd_q$ ]
    V = W_v x (context)  [ $ndd_v$ ]

    q_t = Q[-1]  # Query for last token [ $d_q$ ]

    attention_scores = softmax(q_t @ K.T / sqrt(d_k)) [ $nd_q$ ]
    o_t = attention_scores @ V [ $nd_v$ ]

    next_token = sample_from(o_t)
    context.append(next_token)

**Overall Complexity:** $O(dd_q n^2)$

## LLM Inference With KV Cache

KV_cache = prefill(prompt_tokens)

for t in range(prompt_length + 1, max_length):
    q_t = W_q(x[t-1]) [ $dd_q$ ]
    k_t = W_k(x[t-1]) [ $dd_q$ ]
    v_t = W_v(x[t-1]) [ $dd_q$ ]

    KV_cache.append(k_t, v_t)

    attention_scores = softmax(q_t @ KV_cache.K / sqrt(d_k)) [ $nd_q$ ]
    o_t = attention_scores @ KV_cache.V [ $nd_q$ ]

    x[t] = sample_from(o_t)

**Overall Complexity:** $O(d_q n^2)$

## Experimental Parameters

```cpp
// paged_attention_config.h
#pragma once

namespace paged_attention {

struct PagedAttentionConfig {
    int batch_size;
    int num_heads;
    int head_dim;
    int max_seq_len;
    int page_size;
    int num_pages;
    bool recycle_pages;
};

// Minimal configuration that will definitely work
constexpr PagedAttentionConfig DEFAULT_CONFIG = {
    .batch_size = 64, // [ 2, 4, 8, 16, 32, 64]
    .num_heads = 32,
    .head_dim = 64,
    .max_seq_len = 512, // [512, 1024, 2048, 4096, 8192]
    .page_size = 16,
    .num_pages = 2048,
    .recycle_pages = true
};

} // namespace paged_attention
```

```cpp
// Simple CUDA kernel for updating KV cache
__global__ void update_kv_cache_kernel(
    float* key_cache,
    float* value_cache,
    const float* keys,
    const float* values,
    const int* page_table,
    int seq_idx,
    int seq_len,
    int max_seq_len,
    int num_heads,
    int head_dim,
    int page_size
) {
    int token_idx = blockIdx.x;
    int head_idx = blockIdx.y;
    int dim_idx = threadIdx.x;

    if (token_idx >= seq_len || head_idx >= num_heads || dim_idx >= head_dim) {
        return;
    }

    int page_idx = page_table[seq_idx * max_seq_len + token_idx];
    if (page_idx < 0) {
        return;
    }

    int offset = token_idx % page_size;

    size_t cache_idx = (page_idx * page_size + offset) * num_heads * head_dim +
                       head_idx * head_dim + dim_idx;

    size_t input_idx = token_idx * num_heads * head_dim +
                       head_idx * head_dim + dim_idx;

    key_cache[cache_idx] = keys[input_idx];
    value_cache[cache_idx] = values[input_idx];
}
```

Divide KV Cache into Contiguous KV Blocks and then Store Key and Value Information there.

## Attention Kernel

```
// Extremely simple attention kernel - focus on correctness
__global__ void simple_attention_kernel(
    const float* key_cache,
    const float* value_cache,
    const float* queries,
    float* outputs,
    const int* page_table,
    const int* seq_lengths,
    int batch_size,
    int num_heads,
    int head_dim,
    int max_seq_len,
    int page_size,
    float scale
) {
    int batch_idx = blockIdx.x;
    int head_idx = blockIdx.y;
    int dim_idx = threadIdx.x;
    if (batch_idx >= batch_size || head_idx >= num_heads || dim_idx >= head_dim) {
        return;
    }
    int seq_len = seq_lengths[batch_idx];
    if (seq_len <= 0) {
        return;
    }
    // Use global memory for scores
    float* scores = (float*)malloc(seq_len * sizeof(float));
    if (!scores) {
        if (dim_idx == 0 && head_idx == 0 && batch_idx == 0) {
            printf("[ERROR] Failed to allocate global memory for scores (seq_len=%d)\n", seq_len);
        }
        return;
    }
```

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$$\text{softmax}(x) = \text{softmax}(x - \alpha)$$

$$\alpha = \max_i x_i$$

```
int q_idx = (batch_idx * num_heads + head_idx) * head_dim + dim_idx;
int out_idx = q_idx;
float output_val = 0.0f;
float max_score = -INFINITY;
for (int token_idx = 0; token_idx < seq_len; token_idx++) {
    int page_idx = page_table[batch_idx * max_seq_len + token_idx];
    if (page_idx < 0) {
        scores[token_idx] = -INFINITY;
        continue;
    }
    int offset = token_idx % page_size;
    float dot_product = 0.0f;
    for (int d = 0; d < head_dim; d++) {
        int k_idx = (page_idx * page_size + offset) * num_heads * head_dim +
                    head_idx * head_dim + d;
        int q_d_idx = (batch_idx * num_heads + head_idx) * head_dim + d;
        dot_product += queries[q_d_idx] * key_cache[k_idx];
    }
    float score = dot_product * scale;
    scores[token_idx] = score;
    if (score > max_score) {
        max_score = score;
    }
}
float sum_exp = 0.0f;
for (int token_idx = 0; token_idx < seq_len; token_idx++) {
    if (scores[token_idx] == -INFINITY) {
        continue;
    }
    int page_idx = page_table[batch_idx * max_seq_len + token_idx];
    int offset = token_idx % page_size;
    float exp_score = expf(scores[token_idx] - max_score);
    sum_exp += exp_score;
    int v_idx = (page_idx * page_size + offset) * num_heads * head_dim +
                head_idx * head_dim + dim_idx;
    output_val += exp_score * value_cache[v_idx];
}
if (sum_exp > 0.0f) {
    outputs[out_idx] = output_val / sum_exp;
} else {
    outputs[out_idx] = 0.0f;
}
```

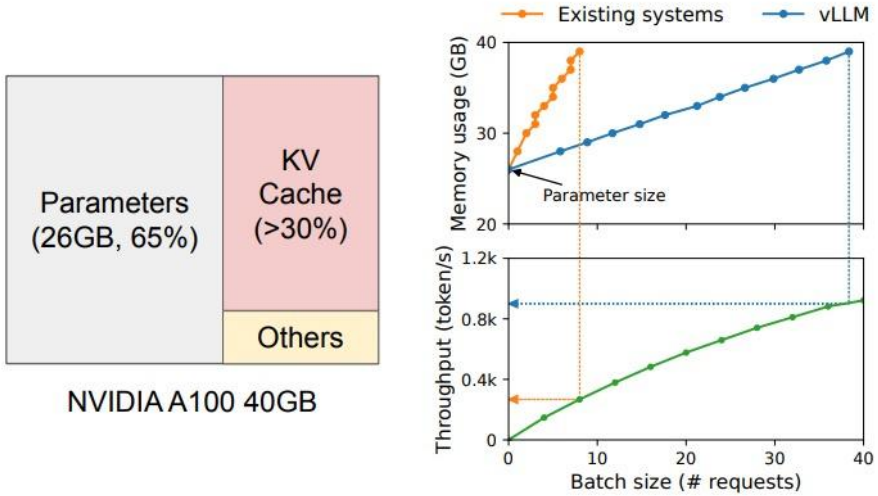



**Figure 1.** *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. *Right:* vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [31, 60], leading to a notable boost in serving throughput.

**Efficient Memory Management for Large Language Model Serving with *PagedAttention***

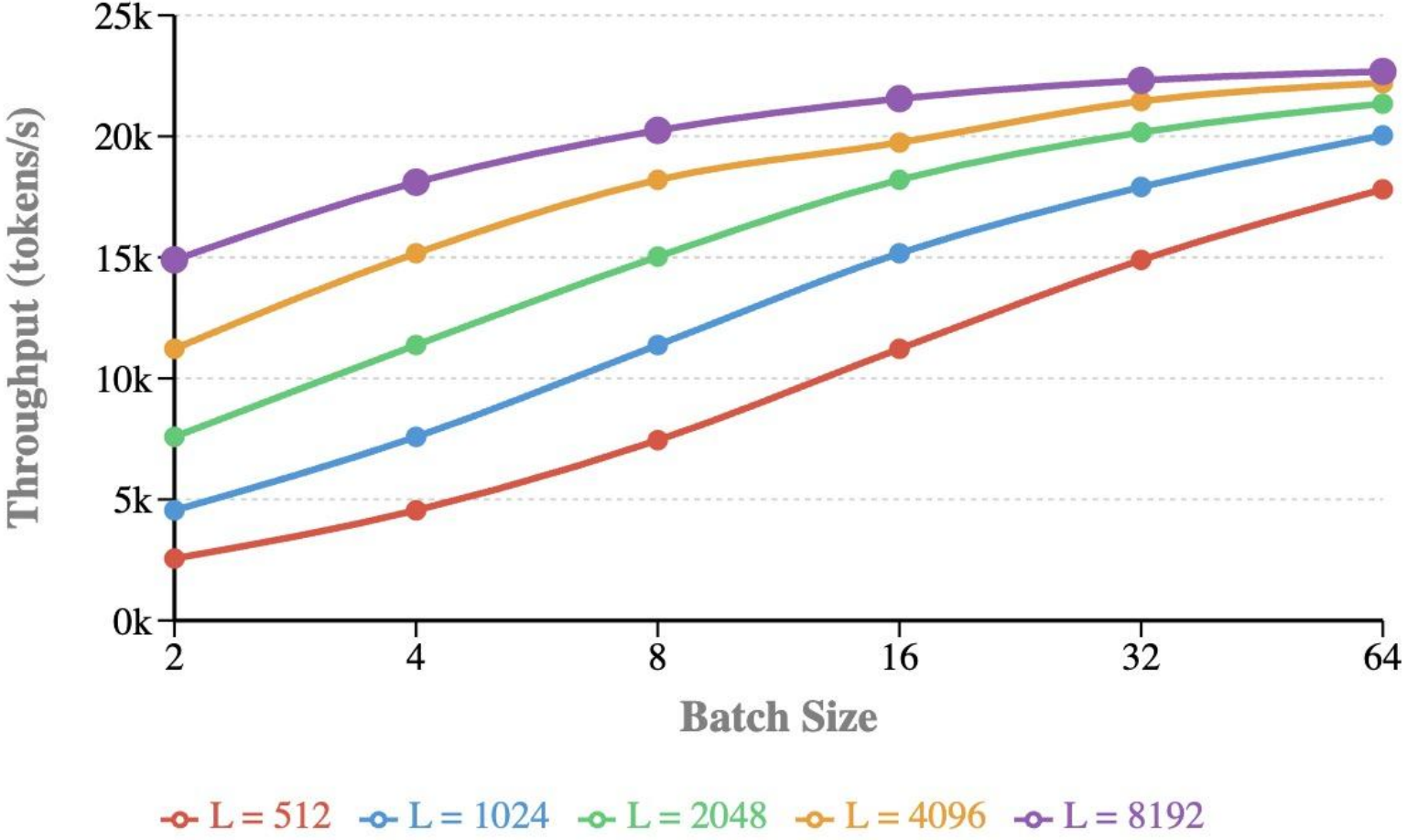
Woosuk Kwon[1,*] Zhuohan Li[1,*] Siyuan Zhuang[1] Ying Sheng[1,2] Lianmin Zheng[1] Cody Hao Yu[3]
Joseph E. Gonzalez[1] Hao Zhang[4] Ion Stoica[1]
[1]UC Berkeley [2]Stanford University [3]Independent Researcher [4]UC San Diego

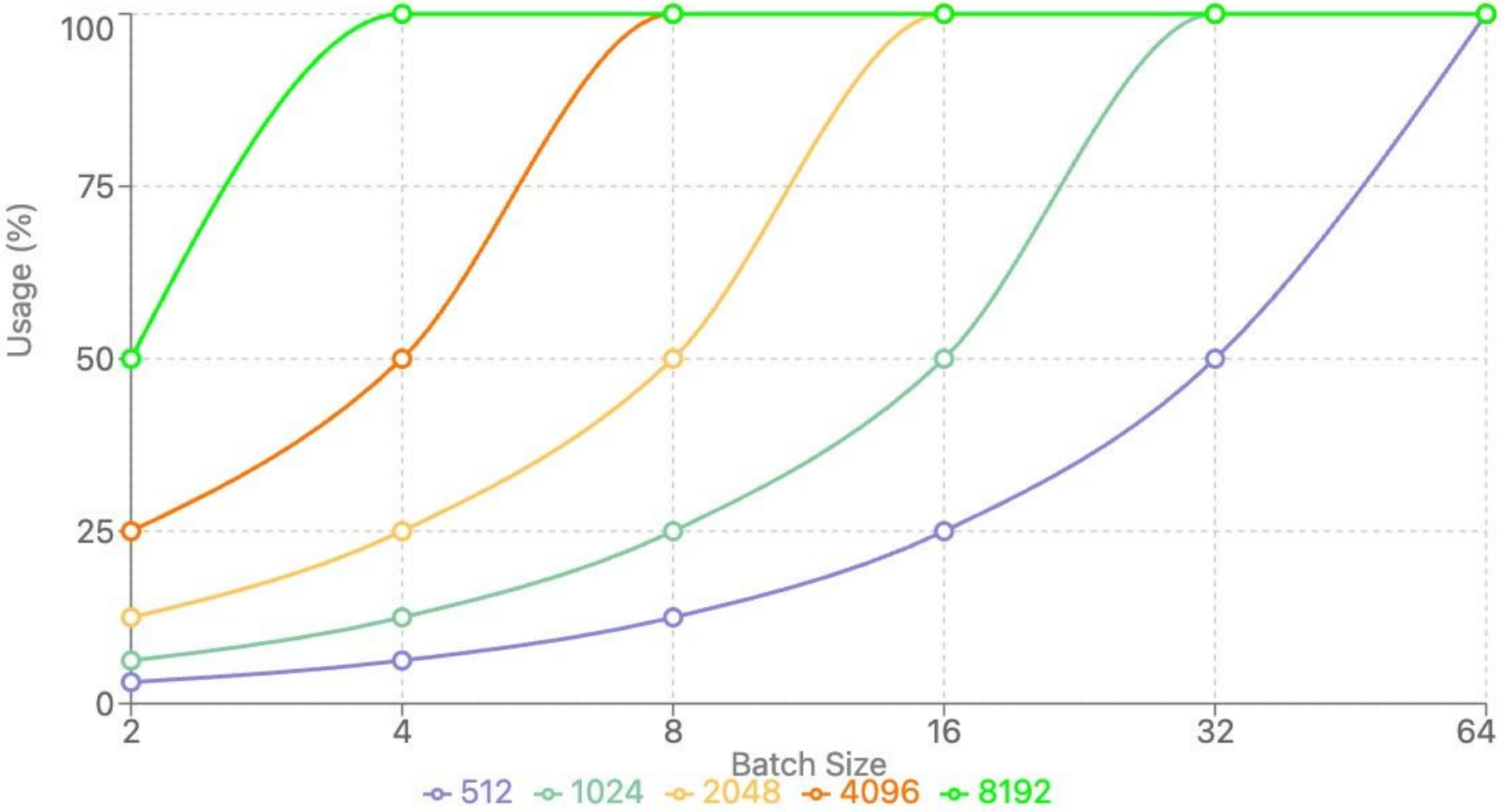
Can we reproduced the patterns shown in the Paged Attention Paper?

| Batch Size | Num of Heads | Head Dimension | Page Size | Num of Pages | Max Seq Length | Response Time (s) | Throughput (tokens/s) |
|---|---|---|---|---|---|---|---|
| 4 | 32 | 64 | 16 | 2048 | 512 | 0.45 | 4551 |
| 4 | 32 | 64 | 16 | 2048 | 1024 | 0.54 | 7585 |
| 4 | 32 | 64 | 16 | 2048 | 2048 | 0.72 | 11378 |
| 4 | 32 | 64 | 16 | 2048 | 4096 | 1.08 | 15170 |
| 4 | 32 | 64 | 16 | 2048 | 8192 | 1.81 | 18104 |
| 2 | 32 | 64 | 16 | 2048 | 512 | 0.40 | 2560 |
| 2 | 32 | 64 | 16 | 2048 | 1024 | 0.45 | 4551 |
| 2 | 32 | 64 | 16 | 2048 | 2048 | 0.54 | 7585 |
| 2 | 32 | 64 | 16 | 2048 | 4096 | 0.73 | 11222 |
| 2 | 32 | 64 | 16 | 2048 | 8192 | 1.10 | 14895 |
| 2 | 32 | 64 | 16 | 2048 | 16384 | Failed X | Failed X |
| 1 | 32 | 32 | 16 | 2048 | 16384 | Failed X | Failed X |

| Batch Size | Num of Heads | Head Dimension | Page Size | Num of Pages | Max Seq Length | Response Time (s) | Throughput (tokens/s) |
|---|---|---|---|---|---|---|---|
| 16 | 32 | 64 | 16 | 2048 | 512 | 0.73 | 11222 |
| 16 | 32 | 64 | 16 | 2048 | 1024 | 1.08 | 15170 |
| 16 | 32 | 64 | 16 | 2048 | 2048 | 1.80 | 18204 |
| 16 | 32 | 64 | 16 | 2048 | 4096 | 3.32 | 19740 |
| 16 | 32 | 64 | 16 | 2048 | 8192 | 6.08 | 21558 |
| 8 | 32 | 64 | 16 | 2048 | 512 | 0.55 | 7447 |
| 8 | 32 | 64 | 16 | 2048 | 1024 | 0.72 | 11378 |
| 8 | 32 | 64 | 16 | 2048 | 2048 | 1.09 | 15031 |
| 8 | 32 | 64 | 16 | 2048 | 4096 | 1.8 | 18204 |
| 8 | 32 | 64 | 16 | 2048 | 8192 | 3.237 | 20246 |

| Batch Size | Num of Heads | Head Dimension | Page Size | Num of Pages | Max Seq Length | Response Time (s) | Throughput (tokens/s) |
|---|---|---|---|---|---|---|---|
| 64 | 32 | 64 | 16 | 2048 | 512 | 1.84 | 17809 |
| 64 | 32 | 64 | 16 | 2048 | 1024 | 3.27 | 20042 |
| 64 | 32 | 64 | 16 | 2048 | 2048 | 6.14 | 21347 |
| 64 | 32 | 64 | 16 | 2048 | 4096 | 11.81 | 22197 |
| 64 | 32 | 64 | 16 | 2048 | 8192 | 23.12 | 22677 |
| 32 | 32 | 64 | 16 | 2048 | 512 | 1.10 | 14895 |
| 32 | 32 | 64 | 16 | 2048 | 1024 | 1.83 | 17906 |
| 32 | 32 | 64 | 16 | 2048 | 2048 | 3.25 | 20165 |
| 32 | 32 | 64 | 16 | 2048 | 4096 | 6.11 | 21452 |
| 32 | 32 | 64 | 16 | 2048 | 8192 | 11.75 | 22310 |


Throughput (tokens/s)
25k
20k
15k
10k
5k
0k
2
4
8
16
32
64
Batch Size
L = 512
L = 1024
L = 2048
L = 4096
L = 8192


100
75
50
25
0
Usage (%)
2
4
8
16
32
64
Batch Size
512
1024
2048
4096
8192

# Thank You!

“People who are really serious about software should make their own hardware.” Alan Kay