CUDA Parallel Programming

Chen Chen

October 16, 2025

Contents

Installing GPGPU-Sim v4: Prerequisites (1/3)

Step 1: Download Sources & CUDA Installer

Download and Prepare Files

Get all necessary files before installation:

```
# 1. Download CUDA 11.0.2 installer
wget http://developer.download.nvidia.com/compute/cuda/11.0.2/\
local_installers/cuda_11.0.2_450.51.05_linux.run

# 2. Clone GPGPU-Sim v4 Distribution
git clone https://github.com/accel-sim/gpgpu-sim_distribution

# 3. Rename the directory for convenience
mv gpgpu-sim_distribution gpgpusim-v4
```

Installing GPGPU-Sim v4: Prerequisites (2/3)

Step 2: Adjust GCC/G++ Version to 9

Configure Compiler Compatibility

CUDA 11.0 requires GCC/G++ 9 for compatibility:

```
sudo apt install gcc-9 g++-9

# Set GCC/G++ version 9 as default
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 100
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-9 100
```

Installing GPGPU-Sim v4: CUDA Setup (2/3)

CUDA Installation and Environment Variables

Step 3: Install CUDA 11.0 and Set Paths

Execute the CUDA installer (deselect the driver installation option):

```
sudo sh cuda_11.0.2_450.51.05_linux.run
```

Add CUDA paths to your ~/.bashrc:

```
echo 'export PATH=$PATH:/usr/local/cuda/bin' >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib:/usr\
/local/cuda/lib64' >> ~/.bashrc
echo 'export CUDA_INSTALL_PATH=/usr/local/cuda' >> ~/.bashrc
source ~/.bashrc

# Verify installation
nvcc -V
# Check if output shows: V11.0.194
```

Installing GPGPU-Sim v4: Final Steps (3/3)

Dependencies and Compilation

Step 4: Install OS Dependencies

Install essential build tools and libraries:

```
sudo apt-get install -y wget build-essential xutils-dev bison\
zlib1g-dev flex libglu1-mesa-dev git g++ libssl-dev libxml2-dev \
libboost-all-dev vim python3-pip libxi-dev libxmu-dev libglut3-dev
```

Step 5: Compile GPGPU-Sim v4

Load environment variables and compile the simulator:

```
cd gpgpusim-v4
source setup_environment
make -j$(nproc)
```

Scan

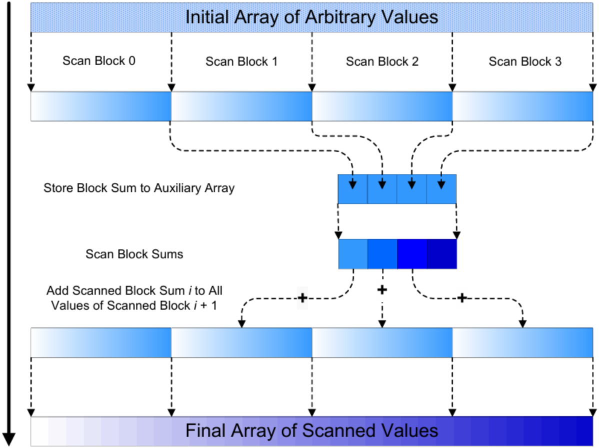


Figure: Algorithm for performing a sum scan on a large array of values.

Header and Definitions

```
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

#define MAX_THREADS_PER_BLOCK 1024
#define MAX_ELEMENTS_PER_BLOCK (MAX_THREADS_PER_BLOCK * 2)
```

Kernel: Data Load

```
__global__ void parallel_large_scan_kernel(int *data, int *prefix_sum, int N, int *sums)
{
    __shared__ int tmp[MAX_ELEMENTS_PER_BLOCK];
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int block_offset = bid * MAX_ELEMENTS_PER_BLOCK;
    int leaf_num = MAX_ELEMENTS_PER_BLOCK;

    tmp[tid * 2]     = (tid * 2 + block_offset < N) ? data[tid * 2 + block_offset] : 0;
    tmp[tid * 2 + 1] = (tid * 2 + 1 + block_offset < N) ? data[tid * 2 + 1 + block_offset] : 0;
    __syncthreads();
```

Upsweep Stage

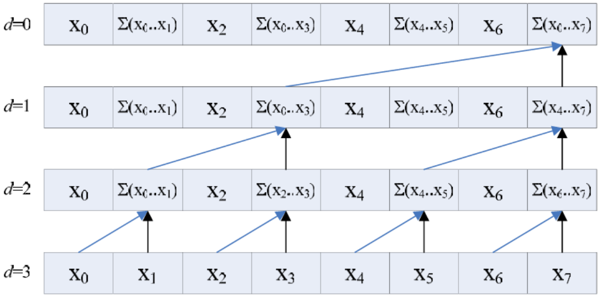


Figure: An illustration of the up-sweep phase of a work-efficient sum scan algorithm.

Kernel: Upsweep Stage

```
int offset = 1;
for (int d = leaf_num >> 1; d > 0; d >>= 1)
{
    if (tid < d) {
        int ai = offset * (2 * tid + 1) - 1;
        int bi = offset * (2 * tid + 2) - 1;
        tmp[bi] += tmp[ai];
    }
    offset *= 2;
    __syncthreads();
}
```

Downsweep Stage

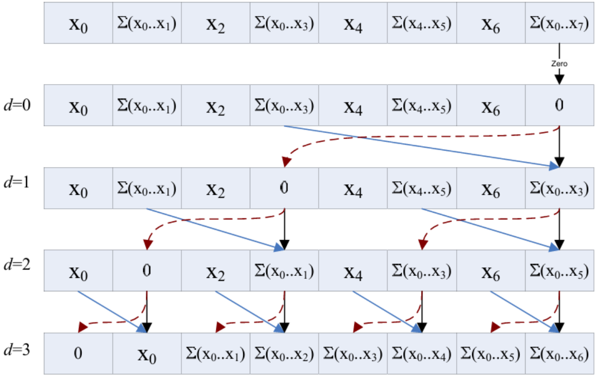


Figure: An illustration of the down-sweep phase of the work efficient parallel sum scan algorithm. Notice that the first step zeros the last element of the array.

Kernel: Exclusive Downsweep

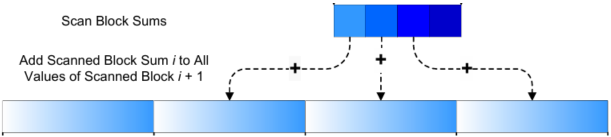
```
    if (tid == 0) {
        sums[bid] = tmp[leaf_num - 1];
        tmp[leaf_num - 1] = 0;
    }
    __syncthreads();

    for (int d = 1; d < leaf_num; d *= 2)
    {
        offset >>= 1;
        if (tid < d) {
            int ai = offset * (2 * tid + 1) - 1;
            int bi = offset * (2 * tid + 2) - 1;
            int t = tmp[ai];
            tmp[ai] = tmp[bi];
            tmp[bi] += t;
        }
        __syncthreads();
    }
}
```

Write Back

```
    if (tid * 2 + block_offset < N)
        prefix_sum[tid * 2 + block_offset] = tmp[tid * 2];
    if (tid * 2 + 1 + block_offset < N)
        prefix_sum[tid * 2 + 1 + block_offset] = tmp[tid * 2 +
        ↪  1];
```

Add Kernel




```
__global__ void add_kernel(int *prefix_sum, const int *values, int N)
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int block_offset = bid * MAX_ELEMENTS_PER_BLOCK;
    int ai = tid * 2 + block_offset;
    int bi = tid * 2 + 1 + block_offset;

    if (ai < N)
        prefix_sum[ai] += values[bid];
    if (bi < N)
        prefix_sum[bi] += values[bid];
}
```

Recursive Scan

```
void recursive_scan(int *d_data, int *d_prefix_sum, int N)
{
    int block_num = (N + MAX_ELEMENTS_PER_BLOCK - 1) / MAX_ELEMENTS_PER_BLOCK;
    int *d_sums = nullptr, *d_sums_prefix = nullptr;
    cudaMalloc(&d_sums, block_num * sizeof(int));
    cudaMalloc(&d_sums_prefix, block_num * sizeof(int));

    parallel_large_scan_kernel<<<block_num, MAX_THREADS_PER_BLOCK>>>(d_data, d_prefix_sum, N, d_sums);
    if (block_num > 1) {
        recursive_scan(d_sums, d_sums_prefix, block_num);
        add_kernel<<<block_num, MAX_THREADS_PER_BLOCK>>>(d_prefix_sum, d_sums_prefix, N);
    }
    cudaFree(d_sums);
    cudaFree(d_sums_prefix);
}
```

Performance Comparison

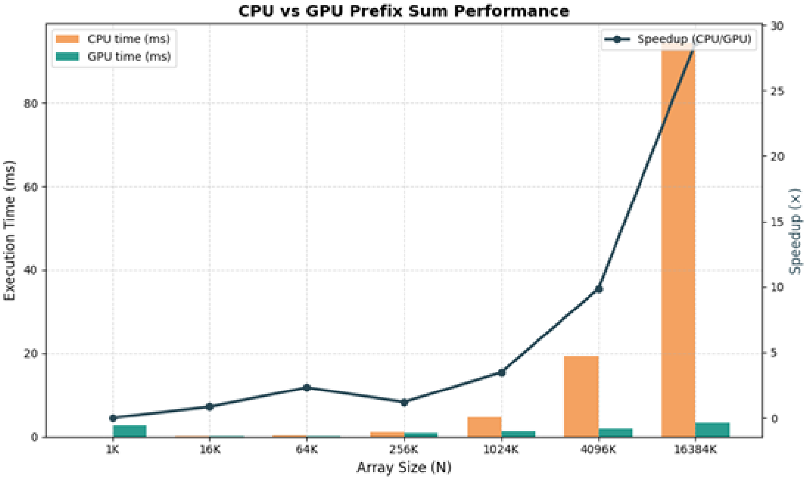


Figure: Performance Comparison of Parallel Scan and Serial Scan at Different Scale.

Parallel Reduction

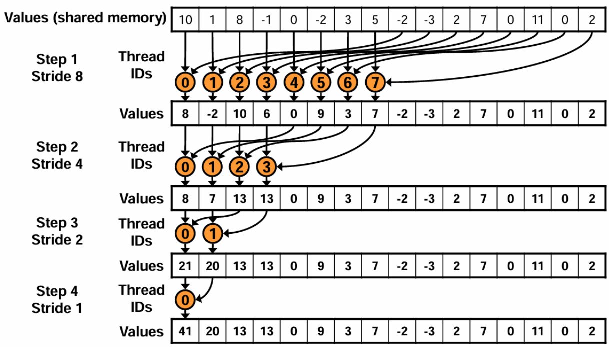


Figure: Sequential addressing is conflict free.

Reduction: Basic Version

Simple In-Block Reduction Kernel

```
__global__ void reduce1(int* iData, int* oData, int n) {
    extern __shared__ int temp[];
    int thid = threadIdx.x;
    int gid  = blockIdx.x * blockDim.x + threadIdx.x;

    // Load input data into shared memory
    temp[thid] = (gid < n) ? iData[gid] : 0;
    __syncthreads();

    // Perform reduction in shared memory
    for (int s = blockDim.x >> 1; s > 0; s >>= 1) {
        if (thid < s) {
            temp[thid] += temp[thid + s];
        }
        __syncthreads();
    }

    // Write result for this block to global memory
    if (thid == 0)
        oData[blockIdx.x] = temp[0];
}
```

Warp Reduce

When $s <= 32$, we have only one warp left. Instructions are SIMD synchronous within a warp. That means when $s <= 32$: We don't need to `__syncthreads()`

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* temp, unsigned
↪  int thid) {
    if (blockSize >= 64) temp[thid] += temp[thid + 32];
    if (blockSize >= 32) temp[thid] += temp[thid + 16];
    if (blockSize >= 16) temp[thid] += temp[thid +  8];
    if (blockSize >=  8) temp[thid] += temp[thid +  4];
    if (blockSize >=  4) temp[thid] += temp[thid +  2];
    if (blockSize >=  2) temp[thid] += temp[thid +  1];
}
```

Reduction

```
template <unsigned int blockSize>
__global__ void reduce2(int* iData, int* oData, int n) {
    extern __shared__ int temp[];
    unsigned int thid = threadIdx.x;
    unsigned int i = blockIdx.x * (blockSize * 2) + threadIdx.x;
    unsigned int gridSize = blockSize * 2 * gridDim.x;
    temp[thid] = 0;

    while (i < n) {
        int a = iData[i];
        int b = (i + blockSize < n) ? iData[i + blockSize] : 0;
        temp[thid] += a + b;
        i += gridSize;
    }
```

Reduction

```
    __syncthreads();

    if (blockSize >= 512) { if (thid < 256) temp[thid] += temp[thid + 256]; __syncthreads(); }
    if (blockSize >= 256) { if (thid < 128) temp[thid] += temp[thid + 128]; __syncthreads(); }
    if (blockSize >= 128) { if (thid <  64) temp[thid] += temp[thid +  64]; __syncthreads(); }

    if (thid < 32) warpReduce<blockSize>(temp, thid);
    if (thid == 0) oData[blockIdx.x] = temp[0];
}
```

Performance Comparison

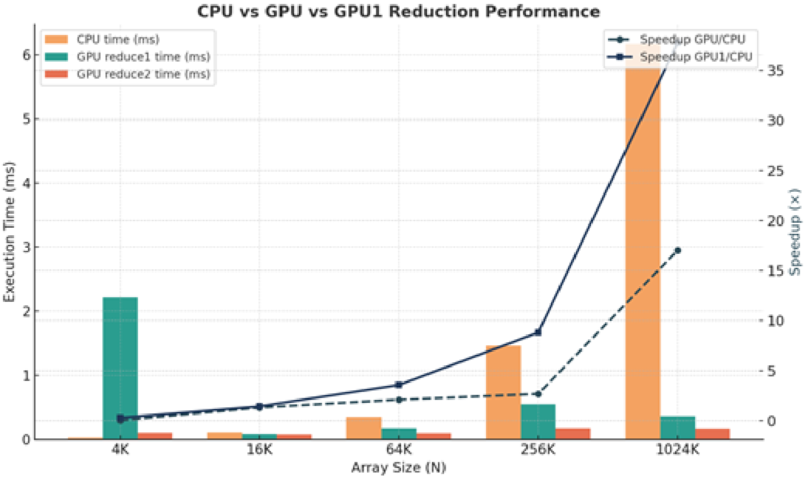


Figure: Performance Comparison of Parallel Reduction and Serial Reduction at Different Scale.