Realtime Computer Graphics on GPUs

GPGPU II

Jan Kolomazník

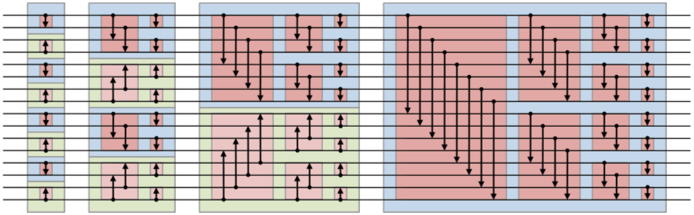
Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague

Computer Graphics Charles University

Parallel Algorithms

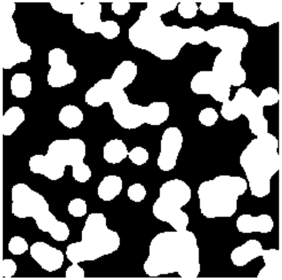
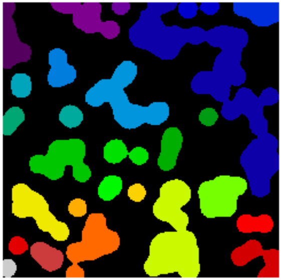
Bitonic Sort

- $O(n \log^2 n)$ comparators
- Alternating bitonic sequences
- Comparisons in a layer are processed in parallel



Connected Component Labeling

- CCL
- Sequential algorithms:
  - Search and ”bucket-fill”
  - Two pass equivalence search
- Sequential algorithms:
  - Union-find

Graph Algorithms

- BFS
  - Queue processed in parallel
  - Managed using atomics
- Graph-cut/Max flow:
  - Push-Relabel algorithm

```
Data: graph
Result: valid flow net
set excess to 0 for all vertices;
set label to 0 for all vertices except source which is $\infty$;
while push or relabel applicable do
    execute the operation;
end
```

CUDA Continuation

Compilation Tools: nvcc

- **nvcc:** NVIDIA's CUDA Compiler (nvcc) is the primary tool for compiling CUDA C/C++ code.
- It supports both host and device code compilation, seamlessly integrating CUDA kernels with host code.
- **Compilation Process:**
  - Preprocessing: Handles CUDA-specific preprocessor directives and includes.
  - Compilation: Translates CUDA C/C++ code into GPU machine code (PTX).
  - Linking: Combines compiled GPU code with host code to generate the final executable.

Compilation Tools: Host Compilers

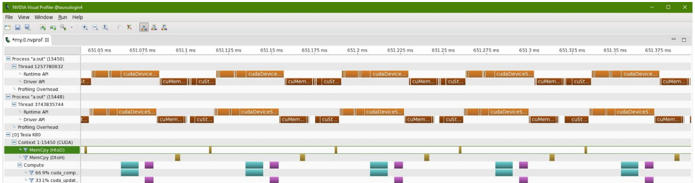
- **Host Compilers:** Various host compilers such as GCC and Clang can be used with nvcc for compiling host code.
- It's essential to check for version compatibility (ABI) to ensure seamless integration with nvcc.
- **Clang:**
  - Recent versions of Clang have added support for compiling CUDA device code.
  - While not fully compatible with nvcc, Clang provides an alternative compiler option.
  - Limitations include lack of full compatibility and missing features like texture memory support.

Compilation Tools: PGI Compiler

- **PGI Compiler:** The PGI compiler suite offers support for compiling CUDA device code for both GPUs and CPUs.
- It provides additional flexibility by allowing device code to be compiled for execution on the CPU.
- This feature is particularly useful for debugging and testing CUDA code on CPU platforms without requiring a GPU.

Profiling Tools

- **nvprof:** NVIDIA Profiler (nvprof) is a command-line profiling tool for analyzing the performance of CUDA applications.
- It provides detailed information on kernel execution times, memory usage, and API calls.
- **nvvp:** NVIDIA Visual Profiler (nvvp) is a graphical profiling tool that offers a more intuitive interface for visualizing and analyzing CUDA application performance.
- It allows developers to identify performance bottlenecks and optimize their code using interactive visualizations.
- Both tools are essential for optimizing CUDA applications for better performance and efficiency.



CUDA Support for Multi-GPU Programming

CUDA Technologies for Multi-GPU Programming:

- **NVIDIA GPUDirect RDMA:** GPUs within the same system to directly access each other's memory without CPU
- **CUDA Streams and Asynchronous Execution:** Concurrent kernel execution and memory operations on different GPUs
- **Unified Memory:** accessible by GPUs and the CPU
- **Multi-GPU Collective Operations:** NVIDIA Collective Communications Library (NCCL) – collective operations (e.g., all-gather, reduce) across multiple GPUs

Hardware Requirements for Multi-GPU Programming:

- **NVIDIA GPU Architecture:** supported on NVIDIA GPUs with compatible architectures, such as Pascal, Volta, Turing, and Ampere.
- **PCIe Bus and NVLink Connectivity:** GPUs connected either through PCIe bus or NVLink interconnect

TENSOR CORES

- New in Volta and Turing architectures
- Accelerate matrix problems of the form $D = A * B + C$
  - Single core works on 4x4 matrices
  - fp16 precision for multiplication
  - fp16 or fp32 for accumulation
- Warp matrix functions
  - Special calls for specific matrix sizes
- Significant speedup of NN training and inference
- Denoising in raytracing APIs

CUDA Graphs and Dependency Tracking

- Dependencies between GPU kernels.
- Efficient scheduling and execution of GPU workloads

Key Concepts:

- **Computation Graphs:** Represent the sequence of GPU operations and their dependencies, allowing for explicit specification of parallel tasks and their interdependencies.
- **Dependency Tracking:** ensuring that dependent kernels wait for their inputs to become available before execution.

Warp voting

- **`__all(predicate)`**: Returns true if the predicate is true for *all* active threads in the warp. Useful for making collective decisions like memory deallocation when all threads agree.
- **`__any(predicate)`**: Returns true if the predicate is true for *any* active thread in the warp. Useful for triggering an action if at least one thread meets a condition.
- **`__ballot(predicate)`**: Returns a 32-bit integer where each bit corresponds to the result of the predicate for each thread in the warp. This function is powerful for complex conditional processing and can be used to count the number of true predicates.

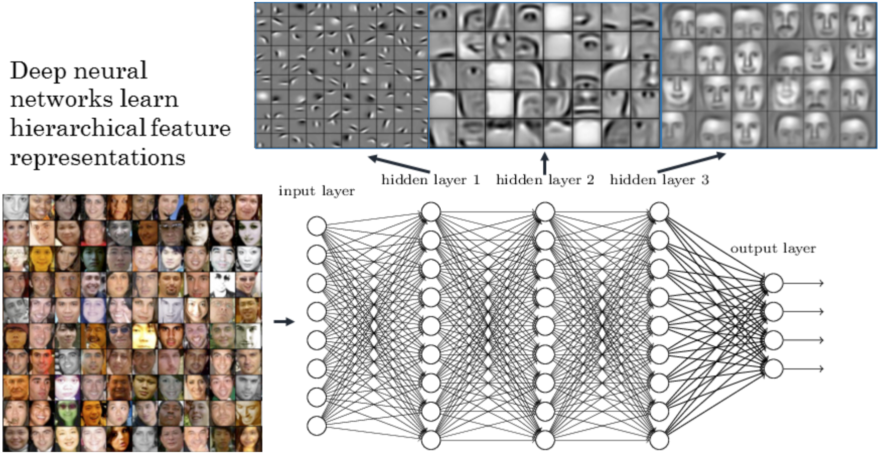
Deep Neural Networks

Deep Neural Networks

- Another neural networks renaissance
- Large neural networks with lots of layers
  - Convolutional networks
- Large numbers of identical neurons – highly parallel by nature
- Backpropagation
  - Millions of parameters
  - Large training set
- Training vs. inference

EXAMPLE

Deep neural networks learn hierarchical feature representations



cuDNN

- GPU-accelerated library of primitives for deep neural networks
- Used for speedup of DNN frameworks like:
  - TensorFlow
  - Caffe
  - Pytorch
  - Keras
  - Matlab
  - …
- Special implementations for selected common cases – highly optimized

TensorRT

- Also by Nvidia
- Inference optimization
  - Significantly less computationally demanding than training
  - Deployed on embedded systems – memory constrains
- Change network topology without sacrificing inference precision
- Use lower numerical precision
- Less space occupied by weights
- Usage of tensor cores

OpenCL

OpenCL

- Alternative to *C for CUDA*
- Basic idea from *C for CUDA*, 1:1 equivalence in some parts
- Programming model for execution of massivily parallel tasks on **CPU, GPU, Cell, . . .**
- Language:
  - Originally subset of C99
  - Subset of C++17 in OpenCL 2.x, 3.0 and backported to 1.0
  - Just-in-time compilation

Host Code

- Code structure similar to shader programming
- API:
  - clBuildProgram()
  - clCreateCommandQueue()
  - clCreateBuffer()
  - clEnqueueWriteBuffer()
  - ...

OpenCL Concepts

OpenCL	CUDA equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

OpenCL threads

OpenCL	**CUDA equivalent**
`get_global_id(0)`	`blockIdx.x` $\cdot$ `blockDim.x` + `threadIdx.x`
`get_local_id(0)`	`threadIdx.x`
`get_global_size(0)`	`gridDim.x` $\cdot$ `blockDim.x`
`get_local_size(0)`	`blockDim.x`

OpenCL Memory

OpenCL	**CUDA equivalent**
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory

Sample OpenCL Kernel

```
// Kernel definition
__kernel void VecAdd(
   __global const float *A,
   __global const float *B,
   __global float *C)
{
   int id = get_global_id(0);
   C[id] = A[id] + B[id];
}
```

SPIR

- Standard Portable Intermediate Representation
- Distribute device-specific pre-compiled binaries
- SPIR-V incorporated in the core specification of:
  - OpenCL 2.1
  - Vulkan API
  - OpenGL 4.6

Comparison between C and C++ for OpenCL

- **History:**
  - **C for OpenCL:**
    - Initially adopted due to its widespread use and familiarity.
    - Suitable for low-level system programming.
  - **C++ for OpenCL:**
    - Introduced to provide a more modern and expressive alternative.
    - Addresses the demand for a higher-level programming model.

Comparison Summary

- Expressiveness: C++ offers a more modern and expressive programming model compared to C.
- Productivity: C++ enhances developer productivity with features like lambda expressions and STL integration.
- Compatibility: Both C and C++ are used for OpenCL programming, with C++ adoption growing through SYCL.
- Performance: Both languages can achieve similar performance levels in OpenCL applications.

Other APIs

C++ AMP, OpenACC

- C++ Accelerated Massive Parallelism:
  - Open specification from Microsoft
  - Builds on DX11
  - Language extensions, runtime library
  - Heterogenous computation
  - Deprecated since VS 2022
- OpenACC:
  - Similar to OpenMP
  - Compiler directives (pragmas) + runtime library

SYCL

- Standard by Khronos
- Cross-platform abstraction layer
- Builds on the underlying concepts, portability and efficiency of OpenCL
- **Single-source** style using completely standard C++

Features of SYCL

SYCL offers several key features for parallel programming:

- **Single-Source Programming:** SYCL enables developers to write parallel code in standard C++ without the need for separate kernel languages or extensions.
- **Host-Device Collaboration:** SYCL allows seamless interaction between host and device code, simplifying data transfers and synchronization.
- **Template Metaprogramming:** SYCL leverages C++ templates to enable compile-time optimizations and generate efficient device code.
- **Automatic Parallelism:** SYCL runtime automatically parallelizes and executes code on available hardware accelerators, maximizing performance.

SAMPLE SYCL CODE

```
#include<CL/sycl.hpp>
#include<iostream>

int
main() {
   using namespace cl::sycl;
   int data[1024];
   // initialize data to be worked on
   // By including all the SYCL work in a {} block, we ensure
   // all SYCL tasks must complete before exiting the block
   {
      queue myQueue;
      bufferbuffer<int, 1> resultBuf(data, range<1>(1024));
      // create a command_group to issue commands to the queue
      myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);
            // enqueuea parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
               writeResult[idx] = idx[0];});
         });
   } // end of scope, so we wait for the queued work to complete
   for(int i = 0; i < 1024; i++)
      std::cout<< "data[" << i << "] = " << data[i] << std::endl;
   return 0;
}
```

SYCL Implementations

SYCL is supported by several implementations, each providing a SYCL-compatible programming environment:

- **ComputeCpp:** Codeplay Software, SYCL implementation for wide range of hardware platforms, including CPUs, GPUs, and FPGAs.
- **Intel oneAPI DPC++ Compiler:** SYCL support for Intel CPUs, GPUs, and FPGAs
- **hipSYCL:** hipSYCL is an open-source SYCL implementation targeting AMD GPUs and CPUs
- **triSYCL:** triSYCL is an experimental and reference SYCL implementation

Thank you for your attention!