Other GPU frameworks

Stewart Martin-Haugh (RAL)

GPU Training, RAL 3 September 2024

GPU language recap

Everything has

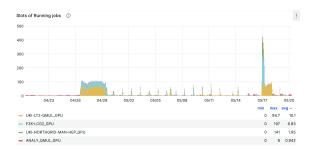
- ► Host compiler (e.g. gcc)
- Device compiler (e.g. nvcc, hipcc)
- ▶ API to allocate memory, synchronise etc (cudaMalloc, cudaDeviceSynchronize)

Language extensions have

Extra syntax for indicating device functions

CUDA advantages

- CUDA is the frontrunner
- Nvidia hardware is the most commonly installed
 - ► E.g. no ATLAS grid sites with non-Nvidia GPUs (for now!)



CUDA

- Very powerful, but only runs on Nvidia hardware
- ► Cannot compile for CPU only
 - Ocelot project aims to make this possible, but does not have official support

AMD GPUs

History

- ► ATI video game graphics cards
- ▶ Bought by AMD in 2006
- General purpose GPU programming with ROCm framework, HIP language/compiler



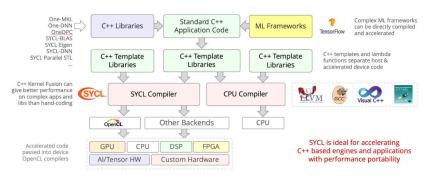
ROCm/HIP

- Syntax similar to CUDA
- Find and replace possible for basic programs
- ► Similar ecosystem available, but less user-friendly

hipify-perl vector_add.cu > vector_add.hip hipcc -o hipify_vector_add vector_add.hip



- ► Open standard, pushed by Intel
- Implementations for all GPU vendors and CPU (also Intel/Altera FPGA)



► More details and examples here



- Clever C++ (templates, mainly) to generate a variety of different backends
- ► Can write something similar to CUDA or SYCL

Accelerator Back-ends

Accelerator Back-end	Lib/API	Devices	Execution strategy grid-blocks	Execution strategy block-threads	
Serial	n/a	Host CPU (single core)	sequential	sequential (only 1 thread per block)	
OpenMP 2.0+ blocks	OpenMP 2.0+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)	
OpenMP 2.0+ threads	OpenMP 2.0+	Host CPU (multi core)	sequential	parallel (preemptive multitasking)	
std::thread	std::thread	Host CPU (multi core)	sequential	parallel (preemptive multitasking)	
ТВВ	TBB 2.2+	Host CPU (multi core)	parallel (preemptive multitasking)	sequential (only 1 thread per block)	
CUDA	CUDA 9.0+	NVIDIA GPUs	parallel (undefined)	parallel (lock-step within warps)	
HIP(clang)	HIP 5.1+	AMD GPUs	parallel (undefined)	parallel (lock-step within warps)	

Kokkos

- Contains higher-level abstractions
- Good support

```
#include <Kokkos_Core.hpp>
#include <cstdio>
#include <typeinfo>
struct hello_world {
  KOKKOS INLINE FUNCTION
  void operator()(const int i) const {
    Kokkos::printf("Hello from i = %i\n", i);
};
int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  printf("Hello World on Kokkos execution space %s\n",
         typeid(Kokkos::DefaultExecutionSpace).name());
  Kokkos::parallel_for("HelloWorld", 15, hello_world());
  Kokkos::finalize();
```

Summary of major options

	NVidia	AMD	Intel	CPU
CUDA	✓	X	X	lsh
HIP	✓	✓	lsh	lsh
SYCL	✓	✓	✓	✓
Alpaka	✓	✓	✓	✓
Kokkos	✓	1	✓	/

The future? standard C++

- ► From C++17 onwards, four execution policies:
 - std::execution::seq: Sequential execution. No parallelism is allowed.
 - ▶ std::execution::unseq: Vectorized execution on the calling thread (this execution policy was added in C++20).
 - ▶ std::execution::par: Parallel execution on one or more threads.
 - std::execution::par_unseq: Parallel execution on one or more threads, with each thread possibly vectorized.
- Support from compilers
- Interest from NVidia
- ► Could eventually write standard C++ and specify target GPU at compile-time...
- But we're not there yet

Comparison with writing performant CPU code

- Using all the capabilities of a modern x86 CPU doesn't happen out-of-the-box
- ► Still takes significant expertise to get good performance despite 20 years of x86 standard: compilers are clever but don't do everything
- But we can at least
 - Compile for ARM, PowerPC without too much effort and get something that largely works
 - Get something that works on Intel and AMD x86 (limited vendor changes)

Live demo

 $StewMH/gpu_translate\ on\ GitHub$