Fakultät für
Mathematik und Informatik

Harnessing the full power of modern hardware accelerators using idomatic C++

Stefan Butz

Fakultät für **Mathematik und Informatik**
Seminar Rechnerarchitekturen
Lehrstuhl Prof. Dr. Lena Oden
Stefan Butz

50 JAHRE FernUniversität in Hagen

Why should we use hardware accelerators?

- **Computational Power vs. Complexity**
  - Modern applications require high-performance computing (HPC).
  - Hardware accelerators (GPUs, FPGAs) provide massive parallelism.
  - However, using them effectively requires specialized programming models.
- **Existing Solutions and Their Challenges**
  - CUDA (NVIDIA) and ROCm (AMD) require vendor-specific knowledge.
  - Open standards like OpenCL and SYCL attempt to provide portability.
  - C++ standard parallelism (stdpar) aims to integrate parallel computing directly into the language.

CUDA Example

- CUDA is the native programming model for NVIDIA GPUs
- Simple CUDA Kernel which doubles the values of a given integer list

```
#include <iostream>
#include <cuda_runtime.h>

__global__ void double_values(int *arr, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) arr[idx] *= 2;
}

int main() {
    int h_arr[10] = {1,2,3,4,5,6,7,8,9,10}, *d_arr;
    cudaMalloc(&d_arr, 10 * sizeof(int));
    cudaMemcpy(d_arr, h_arr, 10 * sizeof(int), cudaMemcpyHostToDevice);

    double_values<<<1, 10>>>(d_arr, 10);

    cudaMemcpy(h_arr, d_arr, 10 * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(d_arr);

    for (int i : h_arr) std::cout << i << " ";
}
```

Challenges with CUDA

- **Problems:**
  - Vendor lock-in (NVIDIA only)
  - Manual memory management
  - Requires explicit kernel definitions and launch configurations
  - Steep learning curve
- **Solution: Higher-level abstractions**

SYCL

- **What is SYCL?**
  - A modern C++ framework for parallel computing
  - Enables single-source heterogeneous programming
  - Portable across CPUs, GPUs, and other accelerators
- **Why SYCL?**
  - High-level abstraction while still providing performance control
  - Open standard (Khronos Group), not vendor-locked

SYCL Example

- Snippet shows a SYCL program that doubles the values of a given integer list

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

int main() {
    sycl::queue q;
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *d_arr = sycl::malloc_shared<int>(10, q);

    std::copy(arr, arr + 10, d_arr);

    q.parallel_for(10, [=](sycl::id<1> i) { d_arr[i] *= 2;
}).wait();
    for (int i = 0; i < 10; i++) std::cout << d_arr[i] << " ";

    sycl::free(d_arr, q);
}
```

SYCL Memory Management

- **Buffers & Accessors**
- **Unified Shared Memory (USM)**

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

int main() {
    sycl::queue q;
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *d_arr = sycl::malloc_shared<int>(10, q);

    std::copy(arr, arr + 10, d_arr);

    q.parallel_for(10, [=](sycl::id<1> i) { d_arr[i] *= 2;
}).wait();
    for (int i = 0; i < 10; i++) std::cout << d_arr[i] << " ";

    sycl::free(d_arr, q);
}
```

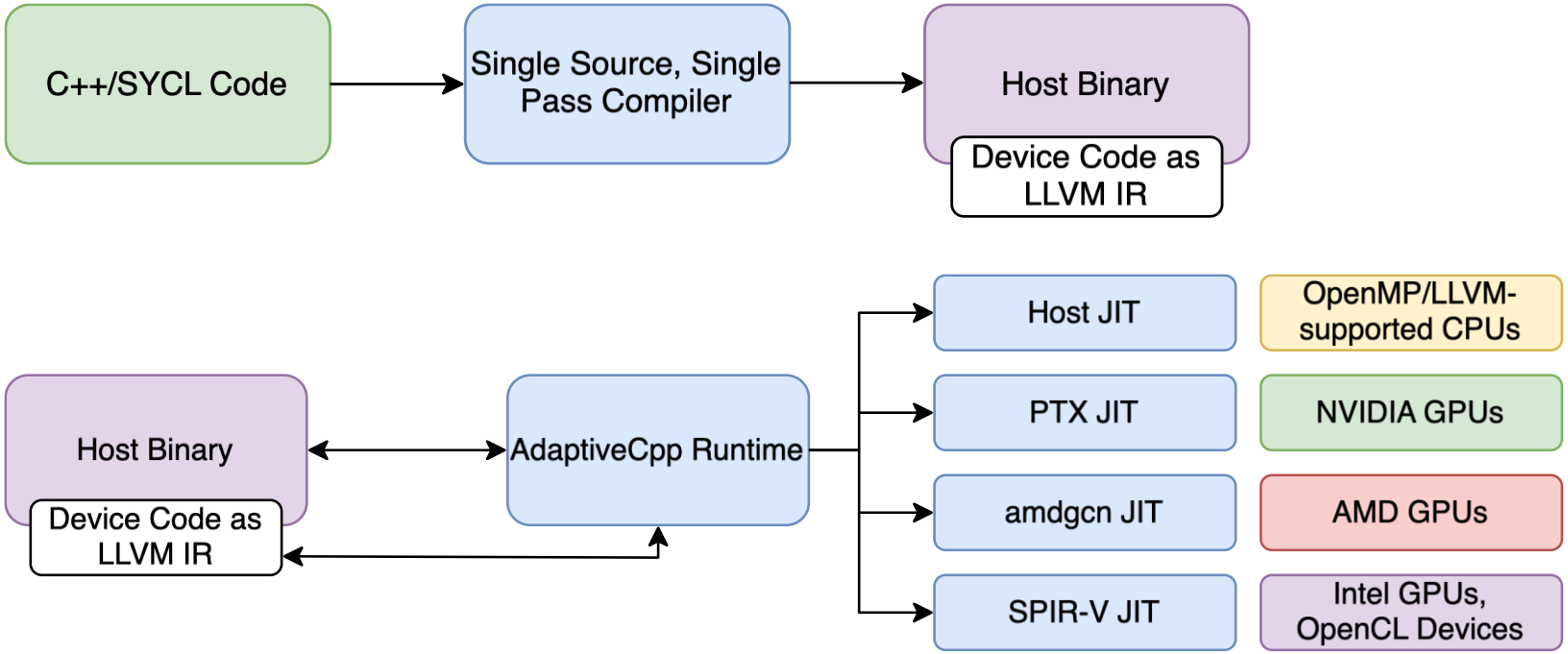
SYCL Execution Model

- **Queues:** Operations scheduled for execution on hardware accelerators
- **In-order queue / Out-of-order queue**
- **Implicit or explicit synchronization**

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

int main() {
    sycl::queue q;
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *d_arr = sycl::malloc_shared<int>(10, q);

    std::copy(arr, arr + 10, d_arr);

    q.parallel_for(10, [=](sycl::id<1> i) { d_arr[i] *= 2;
}).wait();
    for (int i = 0; i < 10; i++) std::cout << d_arr[i] << " ";

    sycl::free(d_arr, q);
}
```

AdaptiveCpp: A SYCL implementation

- **Open-source** SYCL implementation with support for **different platforms**
- **Single-Source, Single-Pass Compilation**



stdpar: C++ Standard Parallelism

- Introduced in C++17
- Proivdes execution policies to parallelize STL algorithms
  - std::execution::seq
  - std::execution::par
  - std::execution::par_unseq

```
#include <iostream>
#include <vector>
#include <execution>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
    std::for_each(std::execution::par, arr.begin(),
                  arr.end(),
                  [](int &x) { x *= 2; });

    for (int i : arr) std::cout << i << " ";
    return 0;
}
```

Challenges in integrating stdpar with SYCL

- Execution Model
- Memory Management
- Synchronization
- Device Execution Constraints
- Performance Overhead

```
#include <iostream>
#include <vector>
#include <execution>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
    std::for_each(std::execution::par, arr.begin(),
                  arr.end(),
                  [](int &x) { x *= 2; });

    for (int i : arr) std::cout << i << " ";
    return 0;
}
```

Challenge 1: Execution Model

- **Detection of parallel code by compiler**
- **Mapping of executions to operations in sycl queues**
- **Kernel Mapping:** std_for_each to sycl::parallel_for

```
#include <iostream>
#include <vector>
#include <execution>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
    std::for_each(std::execution::par, arr.begin(),
                  arr.end(),
                  [](int &x) { x *= 2; });

    for (int i : arr) std::cout << i << " ";
    return 0;
}
```

Challenge 2&3: Memory Management and Synchronization

- Compiler must detect all inputs of parallel code blocks
- Intercepts all memory allocations and allocates USM if necessary
- USM handles data transfer transparently
- Automatically insert sycl::events to to enforce execution order where needed

```
#include <iostream>
#include <vector>
#include <execution>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
    std::for_each(std::execution::par, arr.begin(),
                  arr.end(),
                  [](int &x) { x *= 2; });

    for (int i : arr) std::cout << i << " ";
    return 0;
}
```

Challenge 4: Device Execution Constraints

- Compiler has to limit usage of certain language features (e.g. exceptions)

```
#include <iostream>
#include <vector>
#include <execution>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
    std::for_each(std::execution::par, arr.begin(),
                  arr.end(),
                  [](int &x) { x *= 2; });

    for (int i : arr) std::cout << i << " ";
    return 0;
}
```

Challenge 5: Performance Overhead

- Reducing kernel launch overhead
  - kernel fusion, batching
- Memory management optimizations
  - buffer pooling
- Minimizing synchronization overhead
  - dependency tracking
- Conditional Offloading

Conclusion: It can be this simple

```
#include <iostream>
#include <vector>
#include <execution>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
    std::for_each(std::execution::par, arr.begin(),
                  arr.end(),
                  [](int &x) { x *= 2; });

    for (int i : arr) std::cout << i << " ";
    return 0;
}
```

acpp --acpp-targets=generic -o test test.cpp

References

Same is in my paper.

Thank you for your attention!

Now it’s time for your questions.