

Bringing Next Generation C++ to GPUs

Michael Haidl¹, Michel Steuer², Lars Klein¹ and Sergei Gorlatch¹

¹University of Muenster, Germany

²University of Edinburgh, UK



THE PROBLEM: DOT PRODUCT

```
std::vector<int> a(N), b(N), tmp(N);
std::transform(a.begin(), a.end(), b.begin(), tmp.begin(),
               std::multiplies<int>());
auto result = std::accumulate(tmp.begin(), tmp.end(), 0,
                               std::plus<int>());
```

- The STL is the C++ programmers swiss knife
- STL containers, iterators and algorithms introduce a high-level of abstraction
- Since C++17 it is also parallel

THE PROBLEM: DOT PRODUCT

```
std::vector<int> a(N), b(N), tmp(N);  
std::transform(a.begin(), a.end(), b.begin(), tmp.begin(),  
              std::multiplies<int>());  
auto result = std::accumulate(tmp.begin(), tmp.end(), 0,  
                              std::plus<int>());
```

f1.hpp

```
template<typename T>  
auto mult(const std::vector<T>& a  
         const std::vector<T>& b){  
    std::vector<T> tmp(a.size());  
    std::transform(a.begin(), a.end(),  
                  b.begin(), tmp.begin(),  
                  std::multiplies<T>());  
    return tmp;  
}
```



f2.hpp

```
template<typename T>  
auto sum(const std::vector<T>& a){  
    return std::accumulate(a.begin(),  
                           a.end(), T(), std::plus<T>());  
}
```



THE PROBLEM: DOT PRODUCT

```
std::vector<int> a(N), b(N);  
auto result = sum(mult(a, b));
```

f1.hpp

```
template<typename T>  
auto mult(const std::vector<T>& a  
          const std::vector<T>& b){  
    std::vector<T> tmp(a.size());  
    std::transform(a.begin(), a.end(),  
                  b.begin(), tmp.begin(),  
                  std::multiplies<T>());  
    return tmp;  
}
```

f2.hpp

```
template<typename T>  
auto sum(const std::vector<T>& a){  
    return std::accumulate(a.begin(),  
                           a.end(), T(), std::plus<T>());  
}
```

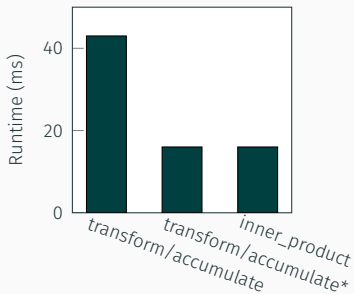


THE PROBLEM: DOT PRODUCT

```
std::vector<int> a(N), b(N);  
auto result = sum(mult(a, b));
```

Performance:

- vectors of size 25600000
- Clang/LLVM 5.0.0svn -O3 optimized



THE PROBLEM: DOT PRODUCT ON GPUS

```
thrust::device_vector<int> a(N), b(N), tmp(N);
thrust::transform(a.begin(), a.end(), b.begin(), tmp.begin(),
                 thrust::multiplies<int>());
auto result = thrust::reduce(tmp.begin(), tmp.end(), 0,
                             thrust::plus<int>());
```

- Highly tuned STL-like library for GPU programming
- Thrust offers containers, iterators and algorithms
- Based on CUDA

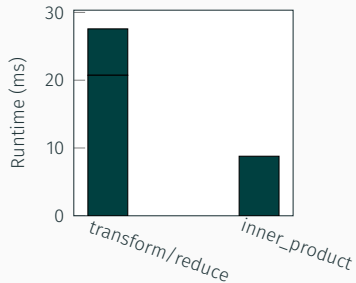
THE PROBLEM: DOT PRODUCT ON GPUS

```
thrust::device_vector<int> a(N), b(N), tmp(N);
thrust::transform(a.begin(), a.end(), b.begin(), tmp.begin(),
                 thrust::multiplies<int>());
auto result = thrust::reduce(tmp.begin(), tmp.end(), 0,
                              thrust::plus<int>());
```

- Highly tuned STL-like library for GPU programming
- Thrust offers containers, iterators and algorithms
- Based on CUDA

Same Experiment:

- `nvcc -O3` (from CUDA 8.0)



- range-v3 prototype implementation by E. Niebler
- Proposed as N4560 for the C++ Standard

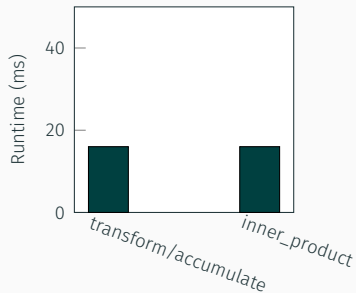
```
std::vector<int> a(N), b(N);  
auto mult = [](auto tpl) { return get<0>(tpl) * get<1>(tpl); };  
auto result = accumulate(view::transform(view::zip(a, b), mult), 0);
```


THE NEXT GENERATION: RANGES FOR THE STL

```
std::vector<int> a(N), b(N);  
auto mult = [](auto tpl) { return get<0>(tpl) * get<1>(tpl); };  
auto result = accumulate(view::transform(view::zip(a, b), mult), 0);
```

Performance?

- Clang/LLVM 5.0.0svn -O3 optimized



THE NEXT GENERATION: RANGES FOR THE STL

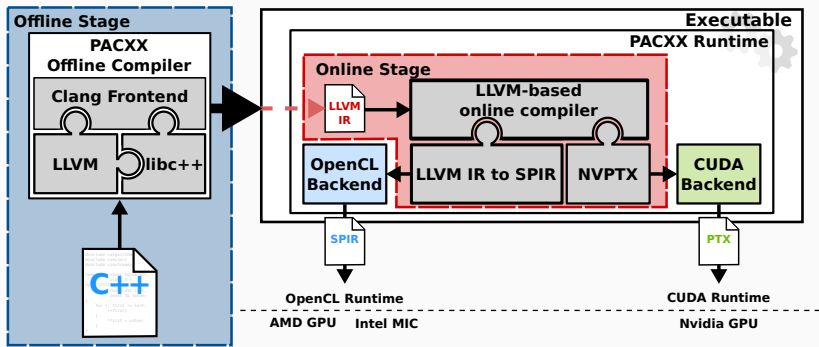
```
std::vector<int> a(N), b(N);  
auto mult = [](auto tpl) { return get<0>(tpl) * get<1>(tpl); };  
auto result = accumulate(view::transform(view::zip(a, b), mult), 0);
```

- Views describe lazy, non-mutating operations on ranges
- Evaluation happens inside an algorithm (e.g., `accumulate`)
- Fusion is **guaranteed** by the implementation

- Extended **range-v3** with GPU-enabled container and algorithms
- Original code of **range-v3** remains unmodified

```
std::vector<int> a(N), b(N);  
auto mult = [](auto tpl) { return get<0>(tpl) * get<1>(tpl); };  
auto ga = gpu::copy(a);  
auto gb = gpu::copy(b);  
auto result = gpu::reduce(view::transform(view::zip(ga, gb), mult), 0);
```

PROGRAMMING ACCELERATORS WITH C++ (PACXX)

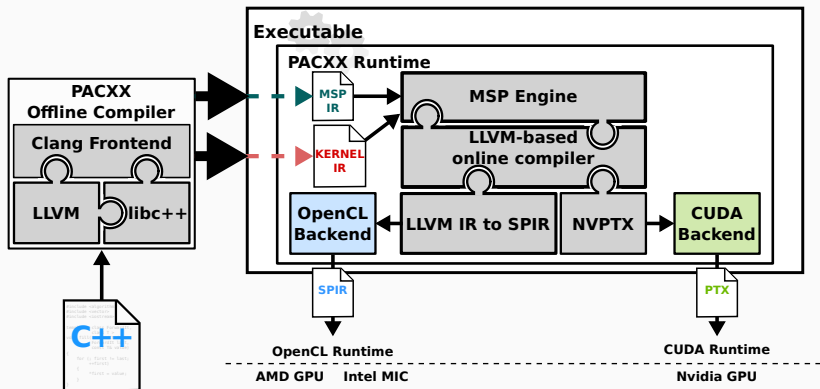


- Based entirely on LLVM / Clang
- Supports C++14 for GPU Programming
- Just-In-Time Compilation of LLVM IR for target accelerators

MULTI-STAGE PROGRAMMING

```
template <typename InRng, typename T, typename Fun>
auto reduce(InRng&& in, T init, Fun&& fun) {
    // 1. preparation of kernel call
    ...
    // 2. create GPU kernel
    auto kernel = pacxx::kernel(
        [fun](auto&& in, auto&& out, int size, auto init) {
            // 2a. stage elements per thread
            auto ept = stage([&]{ return size / get_block_size(0); });
            // 2b. start reduction computation
            auto sum = init;
            for (int x = 0; x < ept; ++x) {
                sum = fun(sum, *(in + gid));
                gid += glbSize; }
            // 2c. perform reduction in shared memory
            ...
            // 2d. write result back
            if (lid == 0) *(out + bid) = shared[0];
        }, blocks, threads);
    // 3. execute kernel
    kernel(in, out, distance(in), init);
    // 4. finish reduction on the CPU
    return std::accumulate(out, init, fun); }
```

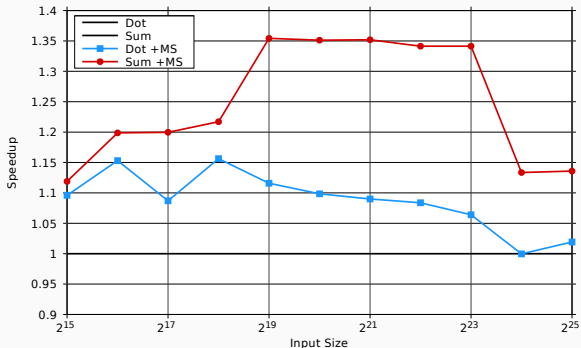
MSP INTEGRATION INTO PACXX



- MSP Engine JIT compiles the MSP IR,
- evaluates **stage** prior to a kernel launch, and
- replaces the calls to **stage** in the kernel's IR with the results.
- Enables more optimizations (e.g., loop-unrolling) in the online stage.

PERFORMANCE IMPACT OF MSP

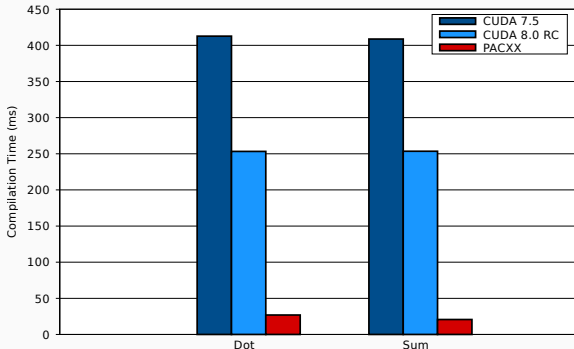
`gpu::reduce` on Nvidia K20c



Up to 35% better performance compared to non-MSP version

JUST-IN-TIME COMPILATION OVERHEAD

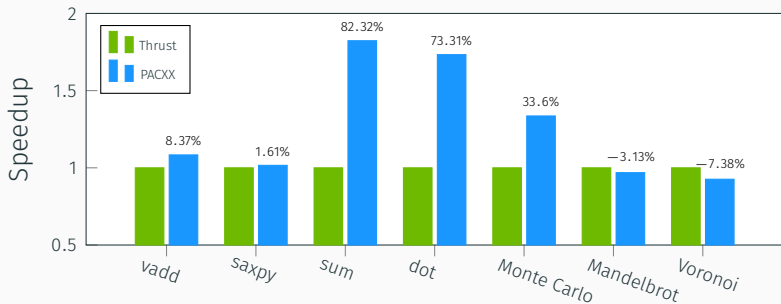
Comparing MSP in PACXX with Nvidia's nvrtc library



10 to 20 times faster because front-end actions are performed.

BENCHMARKS

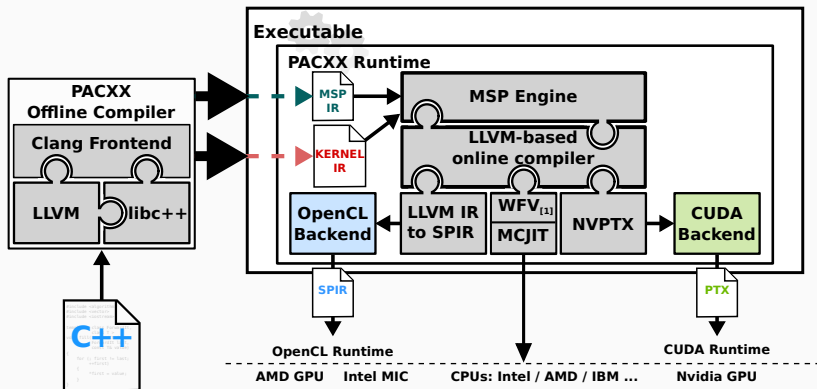
range-v3 + PACXX vs. Nvidia's Thrust



- Evaluated on a Nvidia K20c GPU
 - 11 different input sizes
 - 1000 runs for each benchmark

Competitive performance with a **composable** GPU programming API

GOING NATIVE: WORK IN PROGRESS

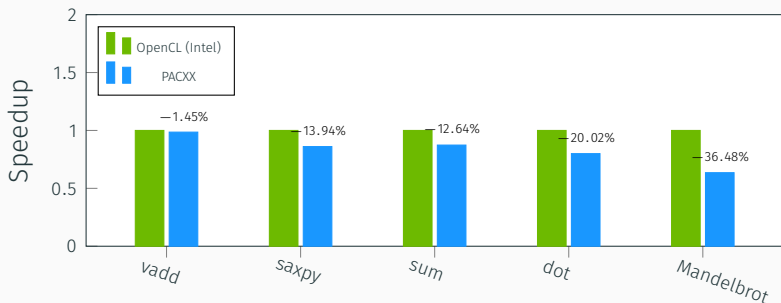


- PACXX is extended by a **native** CPU backend
- The Kernel IR is modified to be runnable on a CPU
- Kernels are vectorized by the Whole Function Vectorizer (WFV) [1]
- MCJIT compiles the kernels and TBB executes them in parallel.

[1] Karrenberg, Ralf, and Sebastian Hack. "Whole-Function Vectorization." @ CGO'11, pp. 141–150

BENCHMARKS

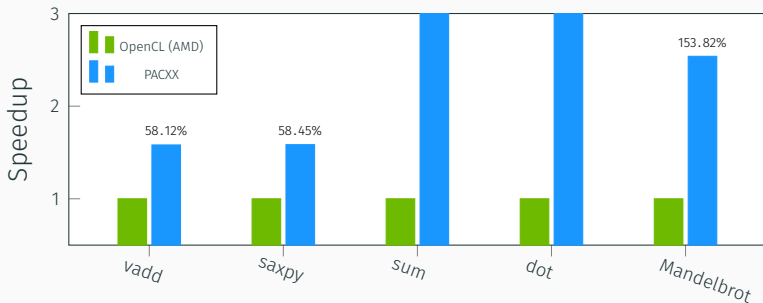
range-v3 + PACXX vs. OpenCL on x86_64



- Running on 2x Intel Xeon E5-2620 CPUs
- Intel's auto-vectorizer optimizes the OpenCL C code

BENCHMARKS

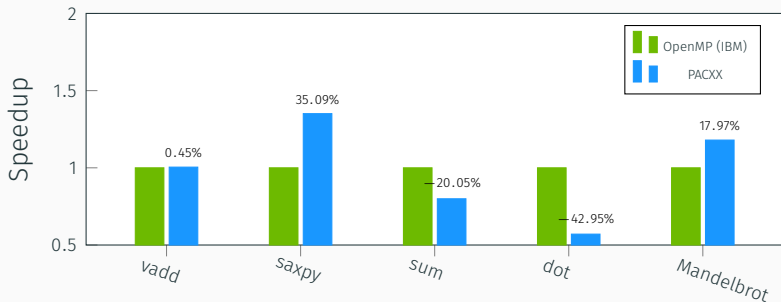
range-v3 + PACXX vs. OpenCL on x86_64



- Running on 2x Intel Xeon E5-2620 CPUs
- AMD OpenCL SDK has no auto-vectorizer
- Barriers are very expensive in AMD's OpenCL implementation (speedup up to 126x for sum)

BENCHMARKS

range-v3 + PACXX vs. OpenMP on IBM Power8



- Running on a PowerNV 8247-42L with 4x IBM Power8e CPUs
- No OpenCL implementation from IBM available
- `#pragma omp parallel for simd` parallelized loops
- Compiled with XL C++ 13.1.5

Questions?