# Design patterns and best practices

Thematic CERN School of Computing

Daniel Hugo Cámpora Pérez
dcampora@cern.ch

2023

# Table of Contents

Good practices

Other standards: OpenCL, HIP, SYCL

Middleware libraries: Alpaka, Kokkos

Parallel design patterns

Summary

Resources

# Table of Contents

Good practices

Other standards: OpenCL, HIP, SYCL

Middleware libraries: Alpaka, Kokkos

Parallel design patterns

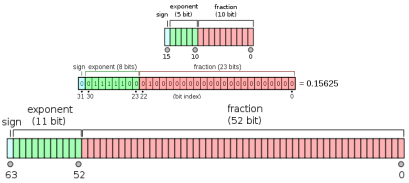
Summary

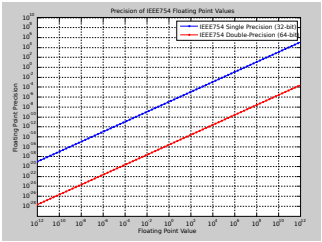
Resources

# Precision for computing

When using decimal numbers, one typically uses **floating point numbers**.

The IEEE 754 standard defines single precision, double precision and half precision as follows:




Still, how much precision is enough?

- Depends on use case.
- Use the least you need.

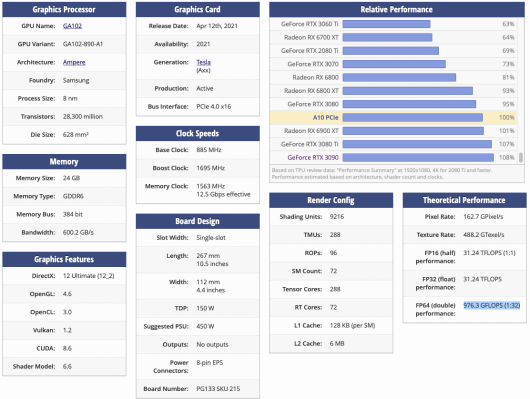
# What precision does your algorithm need?

The answer may not be as simple as one-precision-fits-all. You should consider what precision you need for **storage** and for **arithmetic**!

| Arithmetic | Storage |
|---|---|
| Double precision | Double precision |
| Double precision | Single precision |
| Single precision | Single precision |
| Single precision | Half precision |
| Half precision | Half precision |

# Precision matters (a lot) in GPUs

This is especially true for GPUs, where you would have to go for expensive scientific cards to be able to get good performance with double precision.



| Graphics Processor | |
|---|---|
| GPU Name: | GA102 |
| GPU Variant: | GA102-890-A1 |
| Architecture: | Ampere |
| Foundry: | Samsung |
| Process Size: | 8 nm |
| Transistors: | 28,300 million |
| Die Size: | 628 mm² |

| Memory | |
|---|---|
| Memory Size: | 24 GB |
| Memory Type: | GDDR6 |
| Memory Bus: | 384 bit |
| Bandwidth: | 600.2 GB/s |

| Graphics Features | |
|---|---|
| DirectX: | 12 Ultimate (12_2) |
| OpenGL: | 4.6 |
| OpenCL: | 3.0 |
| Vulkan: | 1.2 |
| CUDA: | 8.6 |
| Shader Model: | 6.6 |

| Graphics Card | |
|---|---|
| Release Date: | Apr 12th, 2021 |
| Availability: | 2021 |
| Generation: | Tesla (Ax) |
| Production: | Active |
| Bus Interface: | PCIe 4.0 x16 |

| Clock Speeds | |
|---|---|
| Base Clock: | 885 MHz |
| Boost Clock: | 1695 MHz |
| Memory Clock: | 1563 MHz 12.5 Gbps effective |

| Board Design | |
|---|---|
| Slot Width: | Single-slot |
| Length: | 267 mm 10.5 inches |
| Width: | 112 mm 4.4 inches |
| TDP: | 150 W |
| Suggested PSU: | 450 W |
| Outputs: | No outputs |
| Power Connectors: | 8-pin EPS |
| Board Number: | PG133 SKU 215 |

Relative Performance

| | |
|---|---|
| GeForce RTX 3060 Ti | 63% |
| Radeon RX 6700 XT | 64% |
| GeForce RTX 2080 Ti | 69% |
| GeForce RTX 3070 | 73% |
| Radeon RX 6800 | 81% |
| Radeon RX 6800 XT | 93% |
| GeForce RTX 3080 | 95% |
| A10 PCIe | 100% |
| Radeon RX 6900 XT | 101% |
| GeForce RTX 3080 Ti | 107% |
| GeForce RTX 3090 | 108% |

Based on TPU review data: "Performance Summary" at 1920x1080, 4K for 2080 Ti and faster. Performance estimated based on architecture, shader count and clocks.

| Render Config | |
|---|---|
| Shading Units: | 9216 |
| TMUs: | 288 |
| ROPs: | 96 |
| SM Count: | 72 |
| Tensor Cores: | 288 |
| RT Cores: | 72 |
| L1 Cache: | 128 KB (per SM) |
| L2 Cache: | 6 MB |

| Theoretical Performance | |
|---|---|
| Pixel Rate: | 162.7 GPixel/s |
| Texture Rate: | 488.2 GTexel/s |
| FP16 (half) performance: | 31.24 TFLOPS (1:1) |
| FP32 (float) performance: | 31.24 TFLOPS |
| FP64 (double) performance: | 976.3 GFLOPS (1:32) |

Source: TechPowerUp

# Floating point rounding

The standard describes four rounding modes:

- round to nearest (typically the default)
- round down
- round up
- round towards zero

In addition, *Fused Multiply-Add* (FMA) units add **precision and performance** when doing floating point operations... which changes slightly the result!

With no optimization flags, GPU compilers have FMAs turned on as opposed to CPU compilers. However, as a general rule *one should not expect FP bit-level precision across different architectures or compilers*, even if they run under the same standard!

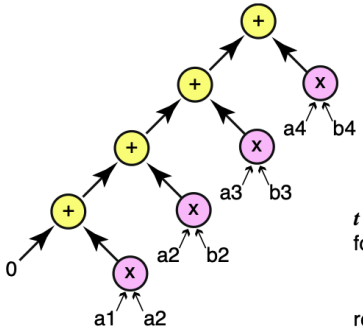
# Floating point rounding - An example

Consider the dot product example:

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4$$

# Dot product - Serial approach

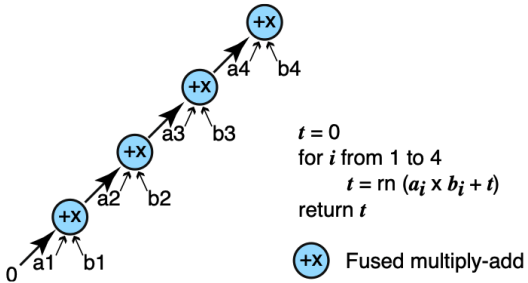
In the serial approach, every element is calculated sequentially.




$t = 0$
for $i$ from 1 to 4
$p = \text{rn}\ (a_i \times b_i)$
$t = \text{rn}\ (t + p)$
return $t$

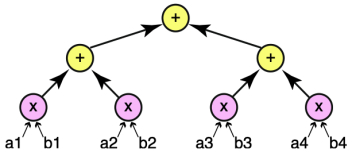
# Dot product - FMA method

Using FMAs, each subsequent multiplication and addition is done in one instruction.



# Dot product - Parallel method

The parallel method divides the problem such that each multiplication is done, followed by the additions, in a Divide and Conquer approach.




$p1$ = rn ($a_1$ x $b_1$)
$p2$ = rn ($a_2$ x $b_2$)
$p3$ = rn ($a_3$ x $b_3$)
$p4$ = rn ($a_4$ x $b_4$)
$s_{left}$ = rn ($p1$ + $p2$)
$s_{right}$ = rn ($p3$ + $p4$)
$t$ = rn ($s_{left}$ + $s_{right}$)
return $t$

# Results

The results for vectors:

$$a = [1.907607, -0.7862027, 1.147311, 0.9604002] \tag{1}$$

$$b = [-0.9355000, -0.6915108, 1.724470, -0.7097529] \tag{2}$$

is:

| method | result | float value |
|---|---|---|
| exact | .0559587528435... | 0x3D65350158... |
| serial | .0559588074 | 0x3D653510 |
| FMA | .0559587515 | 0x3D653501 |
| parallel | .0559587478 | 0x3D653500 |

# Compiler flags

Last, you have some control over how your computation is done. You may want to consider *fast math*, which can impact performance and results quite substantially.

| mode | flags |
|---|---|
| IEEE 754 mode (default) | `-ftz=false`<br>`-prec-div=true`<br>`-prec-sqrt=true` |
| fast mode | `-ftz=true`<br>`-prec-div=false`<br>`-prec-sqrt=false` |

Bear in mind with fast mode:

- Denormals are flushed to zero.
- Division and square root are not computed to the nearest FP value.

# A practical example

What is wrong with the following code?

```
__global__ void shared_memory_example(float* dev_array) {
  for (int i = threadIdx.x; i < 256; i += blockDim.x) {
    dev_array[i] = 1 / std::sqrt(2. + dev_array[i]);
  }
}
```

**Listing:** FP example.

# A practical example

What is wrong with the following code?

```
1 __global__ void shared_memory_example(float* dev_array) {
2   for (int i = threadIdx.x; i < 256; i += blockDim.x) {
3     dev_array[i] = 1 / std::sqrt(2. + dev_array[i]);
4   }
5 }
```

**Listing:** FP example.

*Use compiler flag `-Wdouble-promotion` to avoid surprises!*

# Register spilling

Every thread has a maximum number of registers it can use:

- In GPUs, this limit is configurable (typically between 63 and 255).
- If this limit is surpassed, the kernel will use **local memory** as swap space.

It is "local" because each thread has its own private area. It is actually stored in **global memory** (yes, the slow one).

# Register spilling (2)

Developers have no control over the spilling process:

- Address of global memory where memory is swapped is resolved by compiler.
- Stores are cached in L1 memory.

# Register spilling (2)

Developers have no control over the spilling process:

- Address of global memory where memory is swapped is resolved by compiler.
- Stores are cached in L1 memory.

Spilling could hurt performance:

- Increases memory traffic
- Increases instruction count

# Register spilling (2)

Developers have no control over the spilling process:

- Address of global memory where memory is swapped is resolved by compiler.
- Stores are cached in L1 memory.

Spilling could hurt performance:

- Increases memory traffic
- Increases instruction count

But it is not always bad:

- If accesses are cached
- If your code is not instruction-throughput limited

# How to deal with register spilling

One can evaluate the impact of register spilling through profiling.

The developer has several tools to impact register spilling:

- Increase globally the amount of registers per kernel.
- Increate the amount of configurable L1 cache.
- Some compilers allow to specify *non-caching loads* for global memory.
- `__launch_bounds__` (HIP, CUDA) – Controls *maximum threads per block* and *minimum blocks per SM*. These two impact the number of registers in a kernel.

# Write single-source kernels

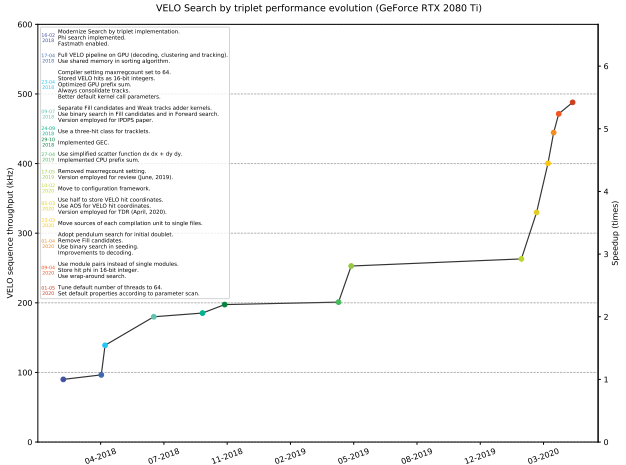
It is possible to organize the code with several header files, containing **definitions**, and source files, containing **implementations**.

However, doing so in GPU code heavily affects performance. The reason is that the compiler optimizes functions to use a number of registers, shared memory and threads, and it cannot perform that optimization if the compilation unit cannot see all code involved.

In other words, if your `__global__` function calls `__device__` functions either free standing or within structs, those should be defined in either:

- The same source file.
- A header file, either **templated** or **inlined**.

# A practical use-case: The Velo Pixel subdetector of LHCb



# Table of Contents

Good practices

**Other standards: OpenCL, HIP, SYCL**

Middleware libraries: Alpaka, Kokkos

Parallel design patterns

Summary

Resources

# OpenCL


*OpenCL™ (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs, and other devices.*

OpenCL 3.0 was released in September 2020:

- It has increased the flexibility by making every functionality from OpenCL 1.2 onwards **optional and queryable**.
- C++ for OpenCL adopts C++17.
- Unified specification.

# OpenCL compilation

OpenCL can either be compiled *offline* or *online*:

- **Offline compilation**: Kernel is pre-built with an OpenCL compiler. Pro: It runs with low invocation latency. Con: It is compiled for a specific architecture.
- **Online compilation**: Kernel source code is distributed instead. Pro: It can run on various architectures. Con: It needs to be JIT-compiled.

Offline compilation is supported through the clang compiler:

OpenCL Offline Compiler Flow

# C++ for OpenCL

C++17 support has arrived with the new *C++ for OpenCL*. The current C++ support has some caveats:

### 2.1.1. Restrictions to C++ features

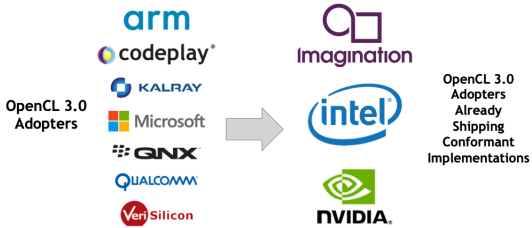
The following C++ language features are not supported:

- Virtual functions (C++17 `[class.virtual]`);
- References to functions including member functions (C++17 `[class.mfct]`);
- Pointers to class member functions (in addition to the regular non-member functions that are already restricted in OpenCL C);
- Exceptions (C++17 `[except]`);
- `dynamic_cast` operator (C++17 `[expr.dynamic.cast]`);
- Non-placement `new`/`delete` operators (C++17 `[expr.new]`/`[expr.delete]`);
- Standard C++ libraries (C++17 `[library]`).

You can try yourself with godbolt.

# Status of OpenCL as of 2021

- OpenCL is perhaps staying relevant after all these years.
- It has gained back an implementation by NVIDIA.
- At the same time, Apple stopped supporting OpenCL (in favor of Metal) and so did AMD (in favor of ROCm).




For more information on OpenCL check out:

- Khronos website on OpenCL
- IWOCL 2021 presentation

# ROCm


*AMD ROCm is the first open-source software development platform for HPC/Hyperscale-class GPU computing.*

ROCm is a platform that has appeared in recent years and is quickly evolving and adapting. It includes:

- Frameworks (Tensorflow / Pytorch).
- Libraries (MIOpen / Blas / RCCL).
- Programming model (HIP).
- Inter-connect (OCD).

# HIP

*HIP is a high performance, CUDA-like programming model that is built on an open and portable framework.*

HIP has several advantages:

- It supports C++17.
- It is almost a 1:1 copy of CUDA – most of the time changes required are very minimal and non-intrusive.
- It supports AMD and NVIDIA targets.

# HIP == CUDA?

- Library call prefix is `hip` instead of `cuda`.
- Warp size depends on GPU: 64 on AMD and 32 on NVIDIA.
- Profiling / debugging is in its infancy.
- No support for latest consumer cards (yet?).
- Low-level calls are different, later CUDA syntax developments are (will always be?) ahead of HIP.
- Specialized hardware (tensor cores, ray tracing) is naturally not there.

It has great potential, especially as more libraries get translated to HIP. For more information:

- ROCm docs

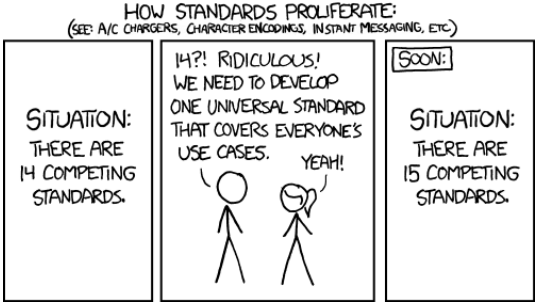
# SYCL


*SYCL 2020's primary goal is to achieve closer convergence with ISO C++, furthering our work to bring parallel heterogeneous programming to modern C++ through open standards.*

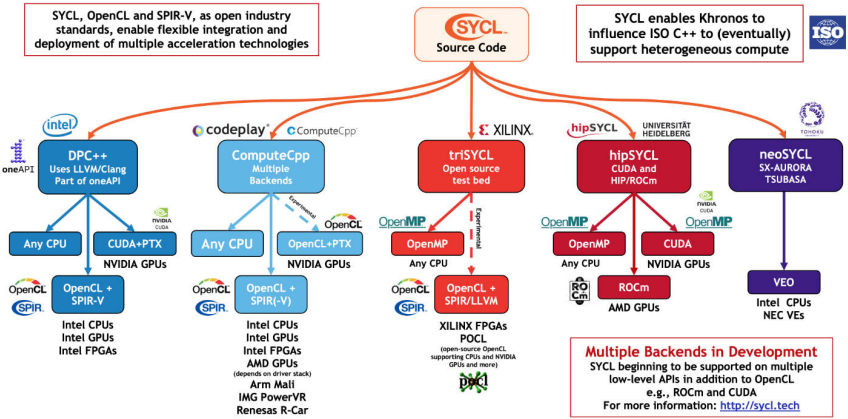
- It supports C++17, its intent is to become part of the standard.
- It attempts to support *everything* (CPUs, AMD GPUs, NVIDIA GPUs, Intel GPUs, Intel FPGAs).
- SYCL is built on top of OpenCL and SPIR-V (the low-level representation shared by eg. Vulkan or OpenGL).

# From the creators of...

It is another standard developed by the Khronos group (same as OpenCL).



# SYCL to everything



# Status of SYCL

SYCL recently released the SYCL 2020 specification.

- Intel supports SYCL as a first-class citizen through its release of OneAPI.
- Intel GPUs are coming.

- Syntax is not easily translatable from CUDA / HIP. Adapting requires work.
- There is no one-size-fits-all and there will never be.
- Given that CUDA is a low-level language, adapting to a higher level one may not be in your best interest if performance is your goal.

In spite of all that SYCL looks very interesting, the next few years will determine whether the language has wide adoption.

# Table of Contents

Good practices

Other standards: OpenCL, HIP, SYCL

**Middleware libraries: Alpaka, Kokkos**

Parallel design patterns

Summary

Resources

# Alpaka


*The alpaka library is a header-only C++14 abstraction library for accelerator development.*

- It acts as a middle layer that can target CPU, NVIDIA GPUs or AMD GPUs through a variety of backends.
- C++-style API which is optimized away by the compiler.

# Alpaka backends

| 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) |
| OpenMP 5.0+ | OpenMP 5.0+ | Host CPU (multi core) | parallel (undefined) | parallel (preemptive multitasking) |
| | | GPU | parallel (undefined) | parallel (lock-step within warps) |
| OpenACC (experimental) | OpenACC 2.0+ | Host CPU (multi core) | parallel (undefined) | parallel (preemptive multitasking) |
| | | GPU | parallel (undefined) | parallel (lock-step within warps) |
| std::thread | std::thread | Host CPU (multi core) | sequential | parallel (preemptive multitasking) |
| Boost.Fiber | boost::fibers::fiber | Host CPU (single core) | sequential | parallel (cooperative multitasking) |
| TBB | 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 4.0+ | AMD GPUs | parallel (undefined) | parallel (lock-step within warps) |

# Alpaka hello world kernel

```cpp
#include <alpaka/alpaka.hpp>
//! Prints "[x, y, z][gtid] Hello World" where tid is the global thread number.
struct HelloWorldKernel
{
  template<typename TAcc>
  ALPAKA_FN_ACC auto operator()(TAcc const& acc) const -> void
  {
    using Dim = alpaka::Dim<TAcc>;
    using Idx = alpaka::Idx<TAcc>;
    using Vec = alpaka::Vec<Dim, Idx>;
    using Vec1 = alpaka::Vec<alpaka::DimInt<1u>, Idx>;

    Vec const globalThreadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc);
    Vec const globalThreadExtent = alpaka::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc);
    Vec1 const linearizedGlobalThreadIdx = alpaka::mapIdx<1u>(globalThreadIdx, globalThreadExtent);

    printf(
      "[z:%u, y:%u, x:%u][linear:%u] Hello World\n",
      static_cast<unsigned>(globalThreadIdx[0u]),
      static_cast<unsigned>(globalThreadIdx[1u]),
      static_cast<unsigned>(globalThreadIdx[2u]),
      static_cast<unsigned>(linearizedGlobalThreadIdx[0u]));
  }
};
```

**Listing:** Alpaka Hello World.

# Should you use Alpaka?

It depends on the priorities of your project:

- It provides portability across different platforms.
- Easier to maintain.

- There is a delay between appeareance of new features and Alpaka support.
- It remains a thin library, but doesn't give you the same control and flexibility as a low-level language (CUDA, HIP).
- Requires learning a different language extension which is not so widely adopted and departs from others (learning curve).

Materials on Alpaka:

- Repository, documentation.
- Online tutorial.

# Kokkos

*Kokkos Core implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms.*

- Very similar to Alpaka, also a header-only library.
- C++ Middle layer that targets CPU, NVIDIA, AMD platforms.

# Kokkos Core Capabilities

| Concept | Example |
|---|---|
| Parallel Loops | **parallel_for**( N, **KOKKOS_LAMBDA** (int i) { ...BODY… }); |
| Parallel Reduction | **parallel_reduce**( **RangePolicy**<ExecSpace>(0,N), **KOKKOS_LAMBDA** (int i, double& upd) {<br>…BODY...<br>upd += ...<br>}, Sum<>(result)); |
| Tightly Nested Loops | **parallel_for**(**MDRangePolicy**<**Rank**<3> > ({0,0,0},{N1,N2,N3},{T1,T2,T3},<br>**KOKKOS_LAMBDA** (int i, int j, int k) {…BODY...}); |
| Non-Tightly Nested Loops | **parallel_for**( **TeamPolicy**<**Schedule**<**Dynamic**>>( N, TS ), **KOKKOS_LAMBDA** (Team team) {<br>… COMMON CODE 1 ...<br>**parallel_for**(**TeamThreadRange**( team, M(N)), [&] (int j) { ... INNER BODY... });<br>… COMMON CODE 2 ...<br>}); |
| Task Dag | **task_spawn**( **TaskTeam**( scheduler , priority), **KOKKOS_LAMBDA** (Team team) { … BODY }); |
| Data Allocation | **View**<double**, Layout, MemSpace> a("A",N,M); |
| Data Transfer | **deep_copy**(a,b); |
| Atomics | atomic_add(&a[i],5.0); View<double*,MemoryTraits<AtomicAccess>> a(); a(i)+=5.0; |
| Exec Spaces | Serial, Threads, OpenMP, Cuda, HPX (experimental), ROCm (experimental) |

# Kokkos reduction kernel

```
1  struct squaresum {
2    using value_type = int;
3    KOKKOS_INLINE_FUNCTION
4    void operator()(const int i, int& lsum) const {
5      lsum += i * i;  // compute the sum of squares
6    }
7  };
8
9  int n = 10, sum = 0;
10 Kokkos::parallel_reduce(n, squaresum(), sum);
```

**Listing:** Kokkos reduction kernel.

# Should you use Kokkos?

- It provides portability and maintainability.
- It provides some higher level functionality (eg. parallel_reduction, execution policies).

- Documentation is scarce, learning curve.
- It provides a subset of low-level functionality of vendor-driven languages (CUDA, HIP, SYCL).

Materials on Kokkos:

- Repository.
- GTC presentation, video.

# Bonus: Portability in CUDA / HIP?

- CUDA and HIP are very similar standards.
- *Kernel language* is practically the same for most use-cases.
- For hardware-specific optimizations (eg. tensor cores), you would have to implement a portable version by hand for portability.

- *Utility functions* (memcpy, memset, malloc, kernel invocation...) can be defined with macros or a hand-made middle language.
- In practice, making your own middleware just for utility functions is very little work.

- It is therefore possible to have one codebase with a low-maintainance self-developed wrapper just covering your utility / kernel needs.
- If targeting performance, it is always better to use the native solution.
- What about CPU execution?

# Bonus: CUDA / HIP running on CPU?

If the CUDA code satisfies that it produces the same result when invoked with a block dimension of `{1, 1, 1}` – or in other words:

- `for`-loops over threads are *block-dimension strided*.
- `if`-statements for a single thread refer to threads of index 0.

Then, with some macros and function definitions it is possible to compile the code for CPUs.

See the following presentation for details.

# So, standard or middleware?

- Standards offer the best performance for their native platform.
- Middlewares offer portability.

- It is possible to obtain good performance on a middleware.
- Low-level functions *not supported by the middleware* will require your own implementation across vendors (high effort).

- It is possible to achieve portability between CUDA / HIP / CPU.
- You may want to focus on a single CPU backend if you do this (as opposed to the many variants offered by Alpaka for instance).
- You will maintain the portability layer.

*What is your application's main target?*

# Table of Contents

Good practices

Other standards: OpenCL, HIP, SYCL

Middleware libraries: Alpaka, Kokkos

**Parallel design patterns**

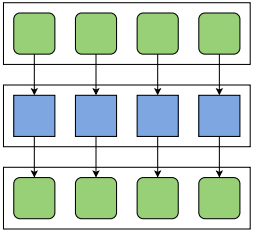
Summary

Resources

# Data parallel vs streaming patterns

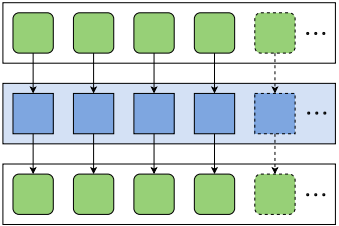
Data parallel patterns

- Map
- Farm
- Reduction
- Stencil



Streaming patterns

- Farm
- Pipeline



# Data parallel vs streaming parallel patterns

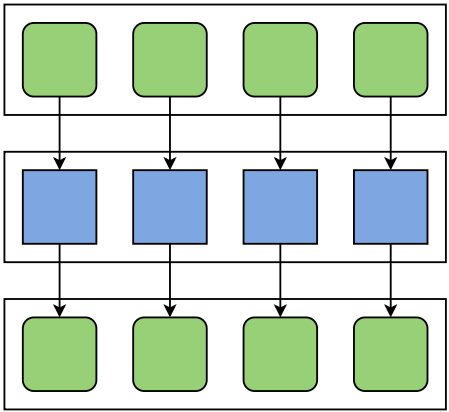
- Size of the input + dependencies between items define which pattern to use.
- Data parallel patterns may not be efficient in streaming scenarios, and the other way around.
- For streaming patterns, there is usually one (or more) input items that distributes the input elements to working items as they come.

# Map (parallel)

- Used on embarrassingly parallel collections of items.
- Same function applied to every item, all at the same "time".
- Applicable if all items are independent.
- Usually good candidate for SIMD abstractions.

**Used in**

Ray tracing, Monte Carlo simulations.

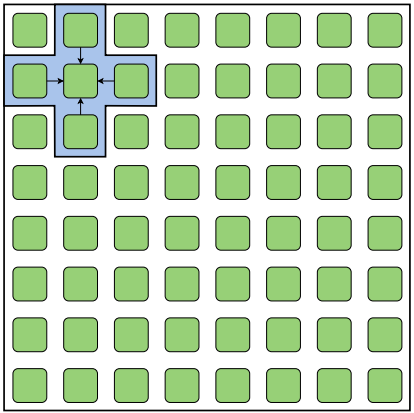


# Stencil (parallel)

- When for every item of a collection, we need data from the neighbourhood items.
- Usually a fixed number of neighbourhood is accessed.
- Boundary conditions have to be taken into account.
- Data reuse in the implementation (cache).

**Used in**

Signal filtering, image processing, grid methods.

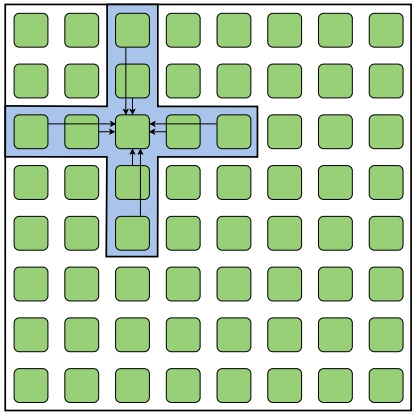


# Stencil (parallel)

- When for every item of a collection, we need data from the neighbourhood items.
- Usually a fixed number of neighbourhood is accessed.
- Boundary conditions have to be taken into account.
- Data reuse in the implementation (cache).

**Used in**

Signal filtering, image processing, grid methods.

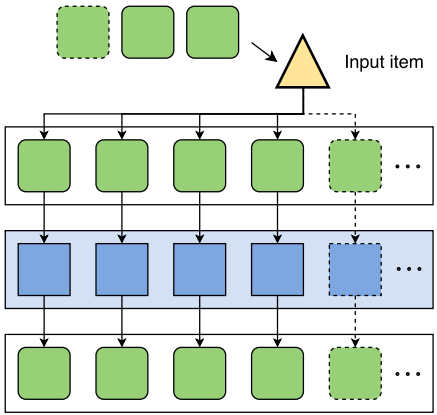


# Farm (parallel streaming)

- Similar to map, but size of collection is not known in advance.
- Used for embarrassingly parallel computations in stream computations.
- There is at least one producer item.

**Used in**

Used in HEP online trigger software.



# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

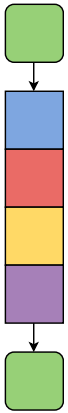
image filtering, signal processing, game engines.

# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.

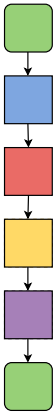


# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.

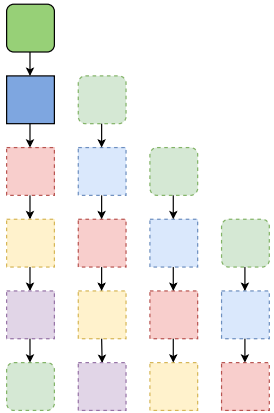


# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.

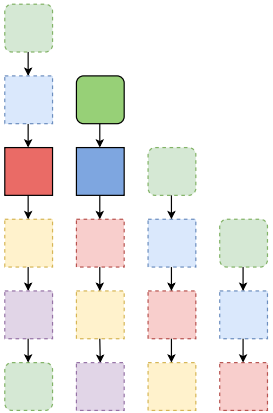


# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.

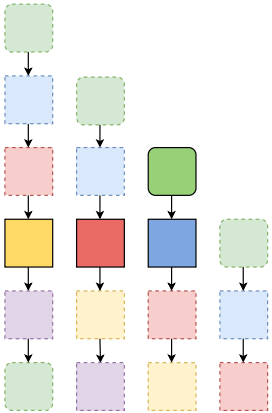


# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.

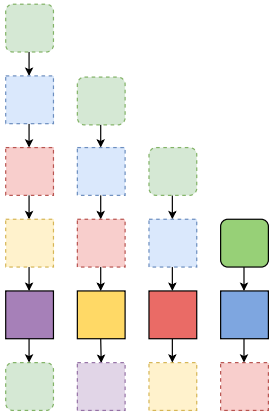


# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.

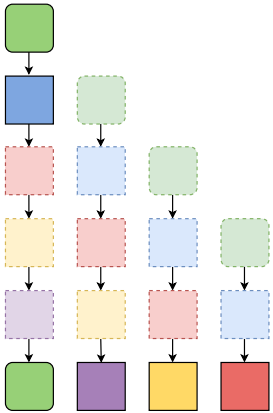


# Pipeline (streaming)

- Size of collection not needed in advance.
- Different steps run in parallel, but others may not be able to run in parallel.
- Different functions are applied in different steps, where the order is important.

**Used in**

image filtering, signal processing, game engines.



# Reduction (sequential)

- Combines a collection of items into one, with a defined operation.
- Many different partition options.
- Elements depend on each other, but are associative.

**Used in**

Matrix operations, computing of statistics on datasets.

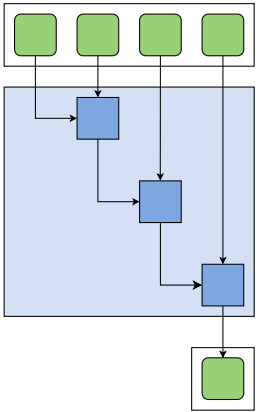


**Figure:** Sequential reduction

# Reduction (parallel)

- Combines a collection of items into one, with a defined operation.
- Many different partition options.
- Elements depend on each other, but are associative.

**Used in**

Matrix operations, computing of statistics on datasets.

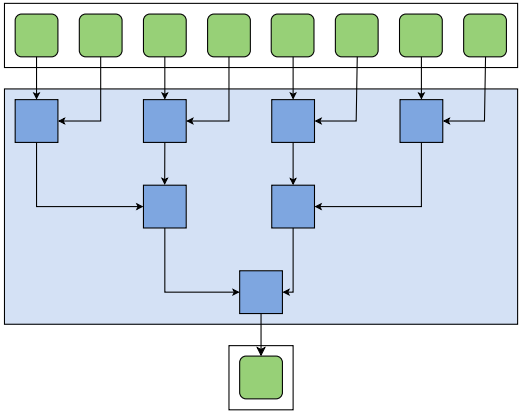


**Figure:** Parallel reduction

# Prefix sum example

The *prefix sum* is a problem that consists in calculating the accumulated sum at every element of an array. For example:

- Number of tracks per event: $[10, 15, 32, 45, 24]$
- Prefix sum: $A = [0, 10, 25, 57, 102, 126]$

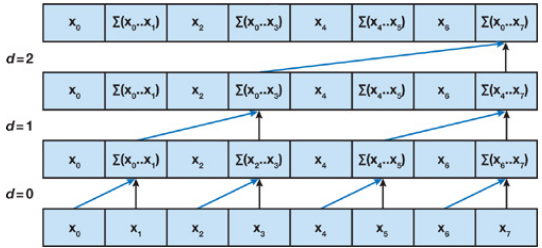
The prefix sum of an array of numbers is extremely useful. It provides:

- The accumulated sum of the entire array (last element).
- The offset of each element (on element $A[i]$).
- The size of each element ($A[i+1] - A[i]$).

# Efficient GPU prefix sum: Blelloch scan

The Blelloch scan consists in performing two *sweeps* of the data.

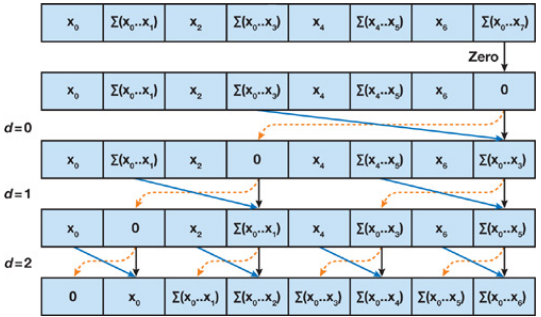
- *up-sweep* – the tree is traversed from leaves to root computing partial sums.




This is actually a reduction!

# Efficient GPU prefix sum: Blelloch scan (2)

- *up-sweep* – the tree is traversed from leaves to root computing partial sums.
- *down-sweep* – the tree is traversed from root to leaves. Each node of the current level passes its value to the element on the left, and the sum of the former and current value on the right.



# LHCb HLT1

The prefix sum is an essential tool of the LHCb HLT1 reconstruction.

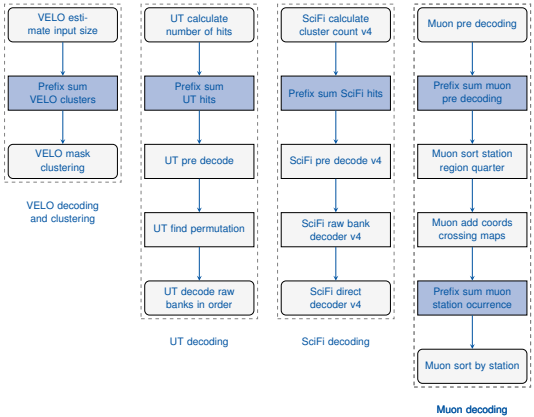



**Figure:** Decoding sequences detail.

# Table of Contents

Good practices

Other standards: OpenCL, HIP, SYCL

Middleware libraries: Alpaka, Kokkos

Parallel design patterns

**Summary**

Resources

# Summary

- Precision affects performance, especially in GPUs.
- GPUs implement IEEE754 standard, deviations are to be expected from compiler / architecture variability.
- Be mindful about register spilling.
- Prefer single-source kernels.

- Choose your standard wisely if targeting best performance.
- Consider middlewares if targeting portability.
- It is possible to obtain portability with a standard, and to obtain performance with a middleware.

- Design patterns are a powerful high-level design tool.
- Know your patterns, design algorithms better.

# Table of Contents

Good practices

Other standards: OpenCL, HIP, SYCL

Middleware libraries: Alpaka, Kokkos

Parallel design patterns

Summary

**Resources**

# Resources used in the talk

- GPU Teaching Kit on Accelerated Computing
- Local Memory and Register Spilling by Paulius Micikevicius
- Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs, white paper
- From sequential to parallel programming with patterns by Placido Fernandez