alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

Plasma-PEPSC Workshop
23 October 2024

alpaka Parallel Programming Library

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF
TECHNISCHE UNIVERSITÄT DRESDEN
CBG Max Planck Institute of Molecular Cell Biology and Genetics
UFZ HELMHOLTZ Centre for Environmental Research
Uniwersytet Wrocławski

SPONSORED BY THE
Federal Ministry of Education and Research
STAATSMINISTERIUM FÜR WISSENSCHAFT KULTUR UND TOURISMUS
Freistaat SACHSEN

Section - I

1. **Introduction:** What is alpaka, where it is used?
2. Hands on 1**:** Installing alpaka and running an example (LUMI)
3. **Parallel programming concepts and portable parallelization by alpaka**
   - Grid Structure and WorkDivision
   - Data Parallelism
   - Indexing
4. Hands on 2**:** *HelloIndex* kernel which prints indexes

Section – II

1. **Memory management** for 1D and 2D data
2. **Filling buffers in parallel**
3. Hands on 3: Kernel to fill initial conditions of heat equation
4. **Heat Equation**
5. **Preparing stencil kernel**
6. Hands on 4: *Heat Equation* stencil kernel
7. **Programming features and data-structures of alpaka**
8. **Usability and Optimization**
   - **Using alpaka mdspan** for easier indexing
     - Hands on 5
   - **Domain Decomposition**
     - Hands on 6 (Day2)
   - **Using async queues** for performance increase
     - Hands on 7 (Day2)
   - **Using shared memory** for performance increase
     - Hands on 8 (Day2)

alpaka – Abstraction Library for Parallel Kernel Acceleration

alpaka is...

- A parallel programming library: Accelerate your code by exploiting your hardware‘s parallelism!
- An abstraction library independent of hardware ecosystem: Create portable code that runs on CPUs and GPUs!
- Open-source software & open-development

Problem of HPC Systems?

Heterogenous Hardware Ecosystem!

TOP500

1 Frontier(USA) 1.194 Exaflop/s,
AMD EPYC CPU + AMD Instict GPU

2 Aurora(USA) 585 Petaflop/s,
Intel Xeon CPU + Intel GPU Max

3 Eagle(USA) 561 Petaflop/s,
Intel Xeon CPU + Nvidia GPU H100

4 Fugaku(Japan) 442 Petaflop/s,
Fujitsu A64FX CPU

5 Lumi(Finland) 380 Petaflop/s,
AMD EPYC CPU + AMD Instinct GPU

www.top500.org

THE LIST

11/2023 Highlights

The 62nd edition of the TOP500 shows five new or upgraded entries in the top 10 but the Frontier system still remains the only true exascale machine with an HPL score of 1.194 Exaflop/s.

The Frontier system at the Oak Ridge National Laboratory, Tennessee, USA remains the No. 1 system on the TOP500 and is still the only system reported with an HPL performance exceeding one Exaflop/s. Frontier brought the pole position back to the USA one year ago on the June 2022 listing and has since been remeasured with an HPL score of 1.194 Exaflop/s.

Frontier is based on the latest HPE Cray EX235a architecture and is equipped with AMD EPYC 64C 2GHz processors. The system has 8,699,904 total cores, a power efficiency rating of 52.59 gigaflops/watt, and relies on HPE's Slingshot 11 network for data transfer.

The Aurora system at the Argonne Leadership Computing Facility, Illinois, USA is currently being commissioned and will at full scale exceed Frontier with a peak performance of 2 Exaflop/s. It was submitted with a measurement on half of the final system achieving 585 Petaflop/s on the HPL benchmark which secured the No. 2 spot on the TOP500.

Aurora is built by Intel based on the HPE Cray EX - Intel Exascale Compute Blade which uses Intel Xeon CPU Max Series processors and Intel Data Center GPU Max Series accelerators which communicate through HPE's Slingshot-11 network interconnect.

The Eagle system installed in the Microsoft Azure cloud in the USA is newly listed as No. 3. This Microsoft NDv5 system is based on Intel Xeon Platinum 8480C processors and NVIDIA H100 accelerators and achieved an HPL score of 561 Pflop/s.

read more »

List Statistics

Vendors System Share

1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE**
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, **Intel**
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, **Microsoft**
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu**
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE**
6	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN
7	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-

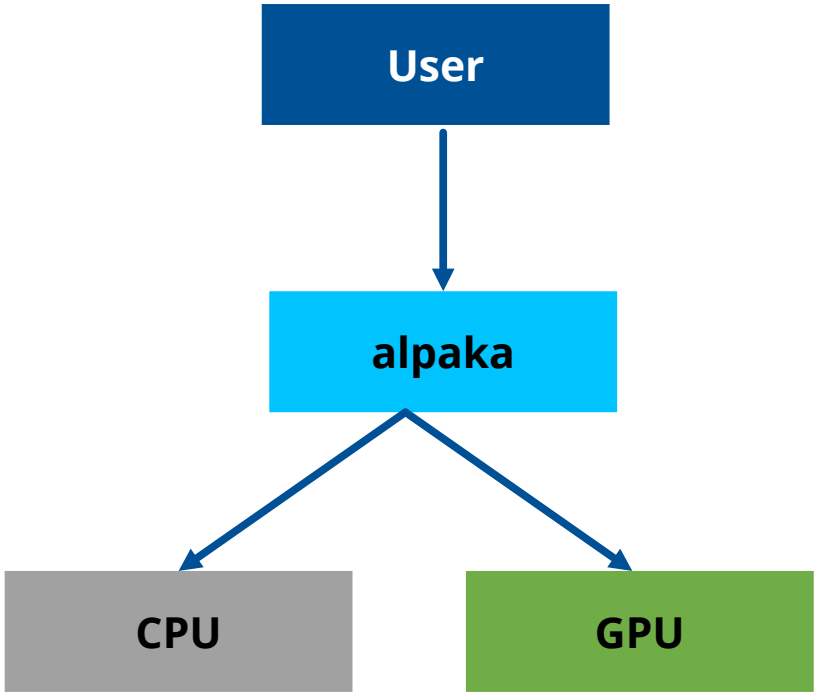
Without alpaka

- Hardware ecosystem is heterogenous, platforms are not inter-operable → parallel programs not easily portable

alpaka: single API to rule them all

- **Abstraction** (not hiding!) of the underlying hardware, compiler and OS
  - No default device, built-in functions, language extentions
- **Easy change of the backend in code**

- **Direct usage of vendor APIs**
  - GPU Backends: Hip (AMD), Cuda (NVidia), SYCL (Intel GPUs)
  - One can use vendor profilers and debuggers (Cuda,HIP…) for alpaka code!
  - CPU Backends: OpenMp, Threads, TbbBlocks
- **Zero abstraction overhead for Kernel execution!**
- **Heterogenous Programming**: Using different backends in a synchronized manner.



Find us on GitHub!

alpaka library: https://www.github.com/alpaka-group/alpaka

- Full source code and many examples, Issue tracker

The documents: https://alpaka.readthedocs.io/en/latest/

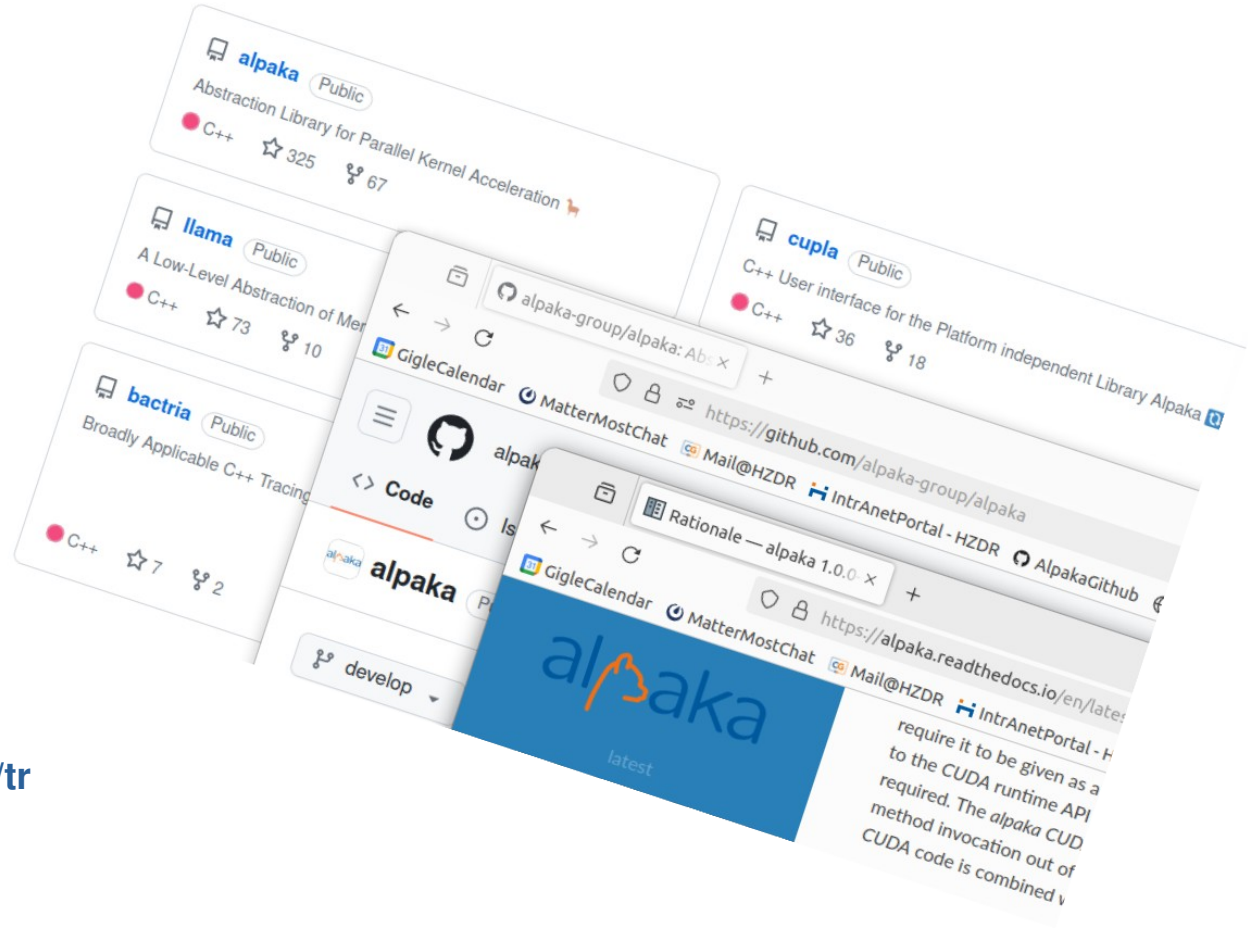
- Installation guide
- Cheatsheet
- Abstraction model and the rationale behind alpaka

Project group: https://www.github.com/alpaka-group

- Contains all alpaka-related projects such as vikunja, cupla ...

PLASMA-PEPSC Workshop Presentations and Hands-ons:

- https://github.com/alpaka-group/alpaka-workshop-slides/tree/oct2024_workshop



Programming with alpaka

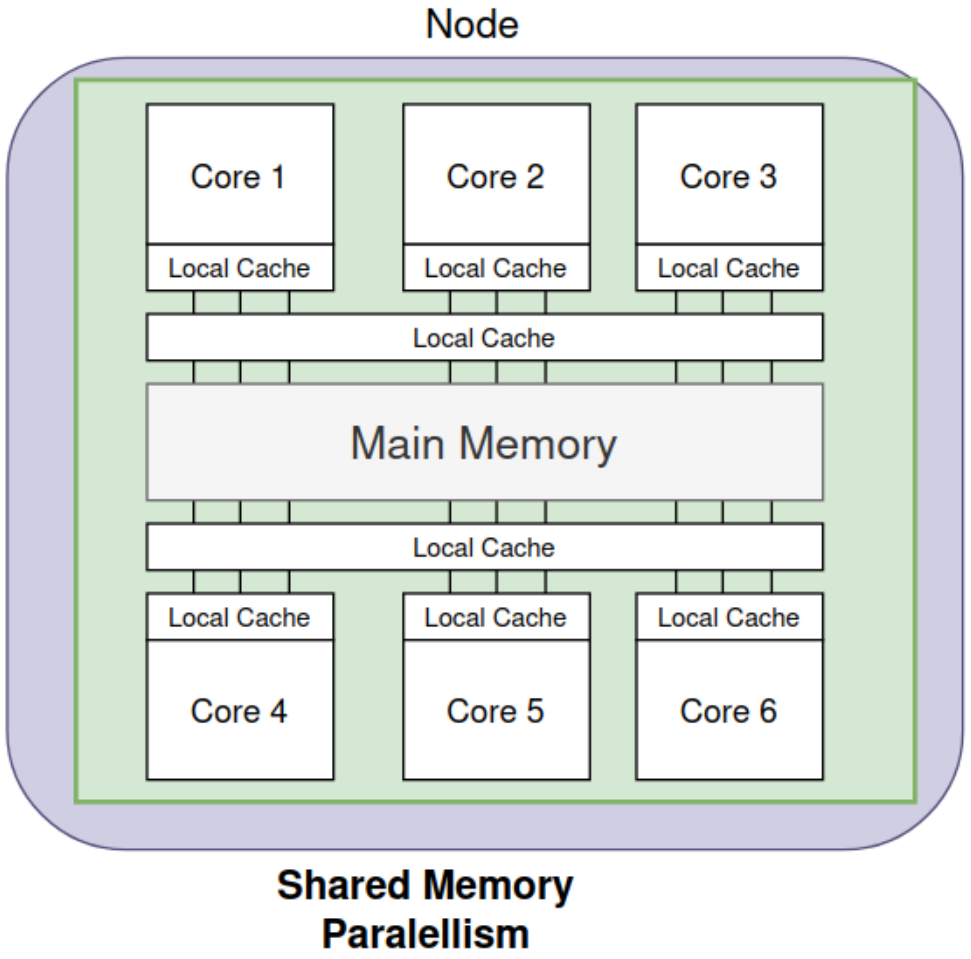
- C++ only!
- alpaka is written entirely in C++17 and C++20.
- Header-only library. No additional runtime dependency.

  `#include <alpaka/alpaka.hpp>` is enough!
- Supports a wide range of modern C++ compilers (g++, clang++, Apple LLVM, MSVC)
- Portable across operating systems: Linux, macOS, Windows

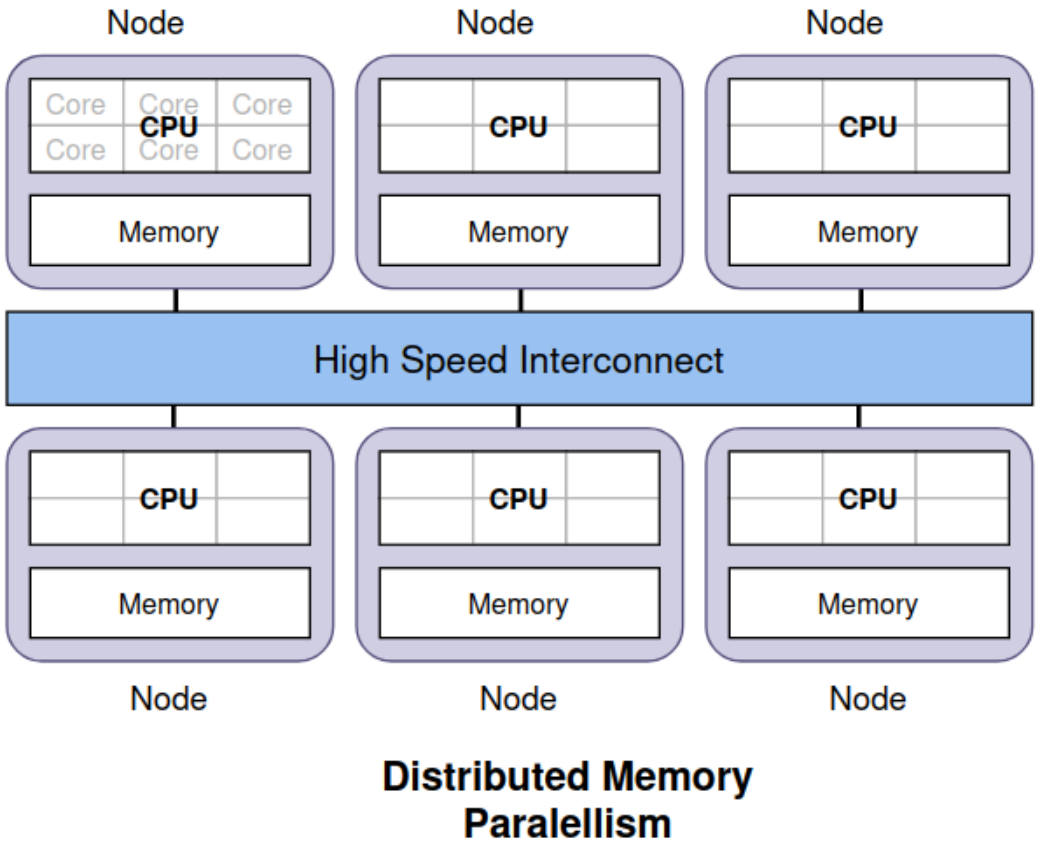
alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Alpaka uses Shared-Memory Paralelism (Node-level parallelism)



Shared Memory Paralellism



Distributed Memory Paralellism

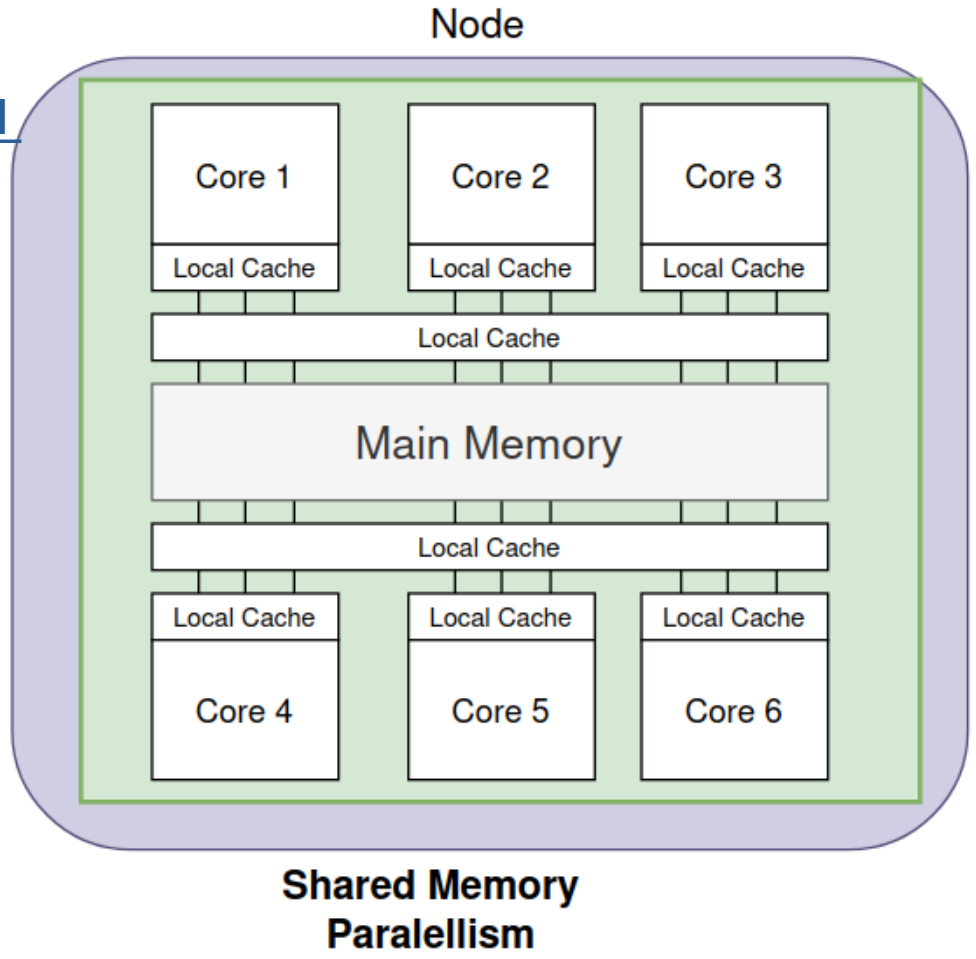
alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Alpaka uses Shared-Memory Paralelism (Node-level parallelism)

Node-Level Parallelism

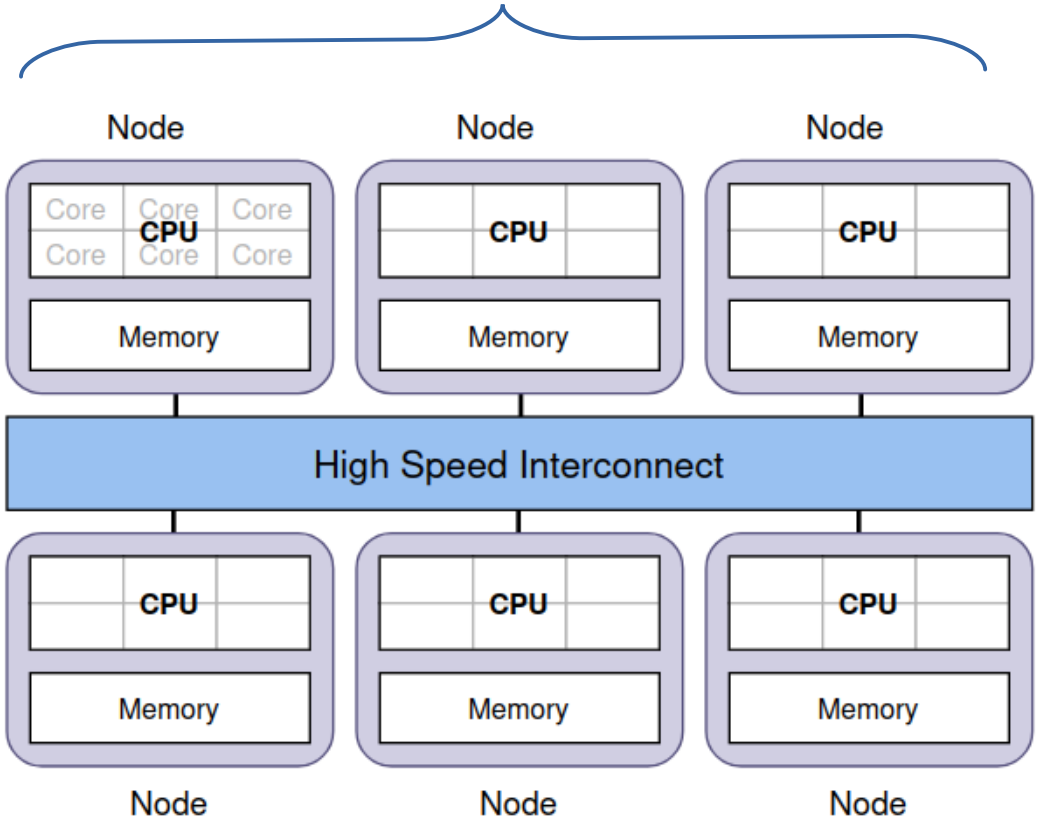
HIP
Cuda
OpenMP
SYCL
Tbb
....



Shared Memory Paralellism

Distributed Memory Parallelism

MPI (Message Passing Interface)

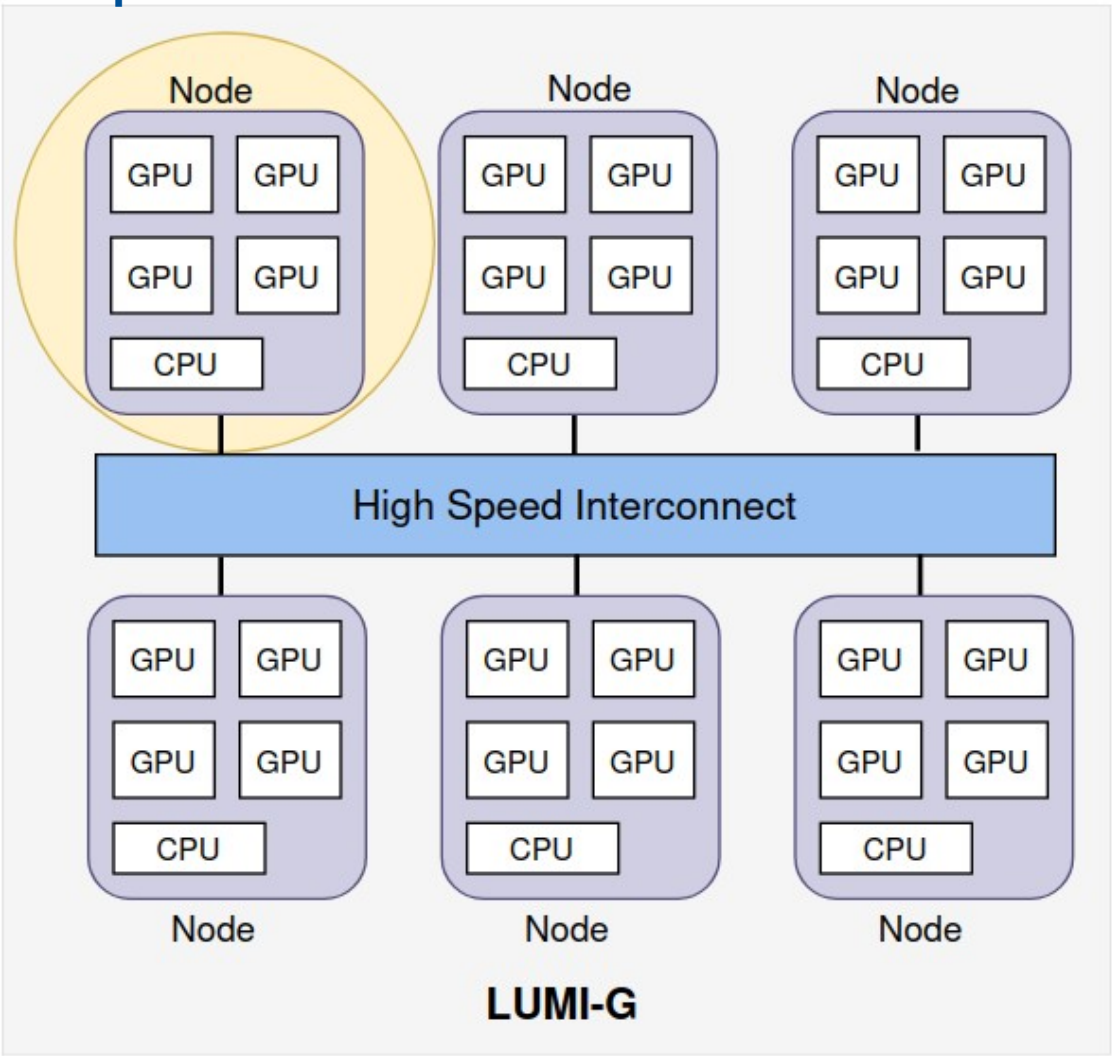


Distributed Memory Paralellism

Node-Level Parallelism on LUMI Supercomputer

LUMI-G: 55,296 GPU-cores per node

4 AMD MI250X GPUs
× 2 Graphics Compute Dies (GCDs)
× 108 Compute Units (CUs)
× 64 Processing Elements (PEs)
= 55,296 GPU cores



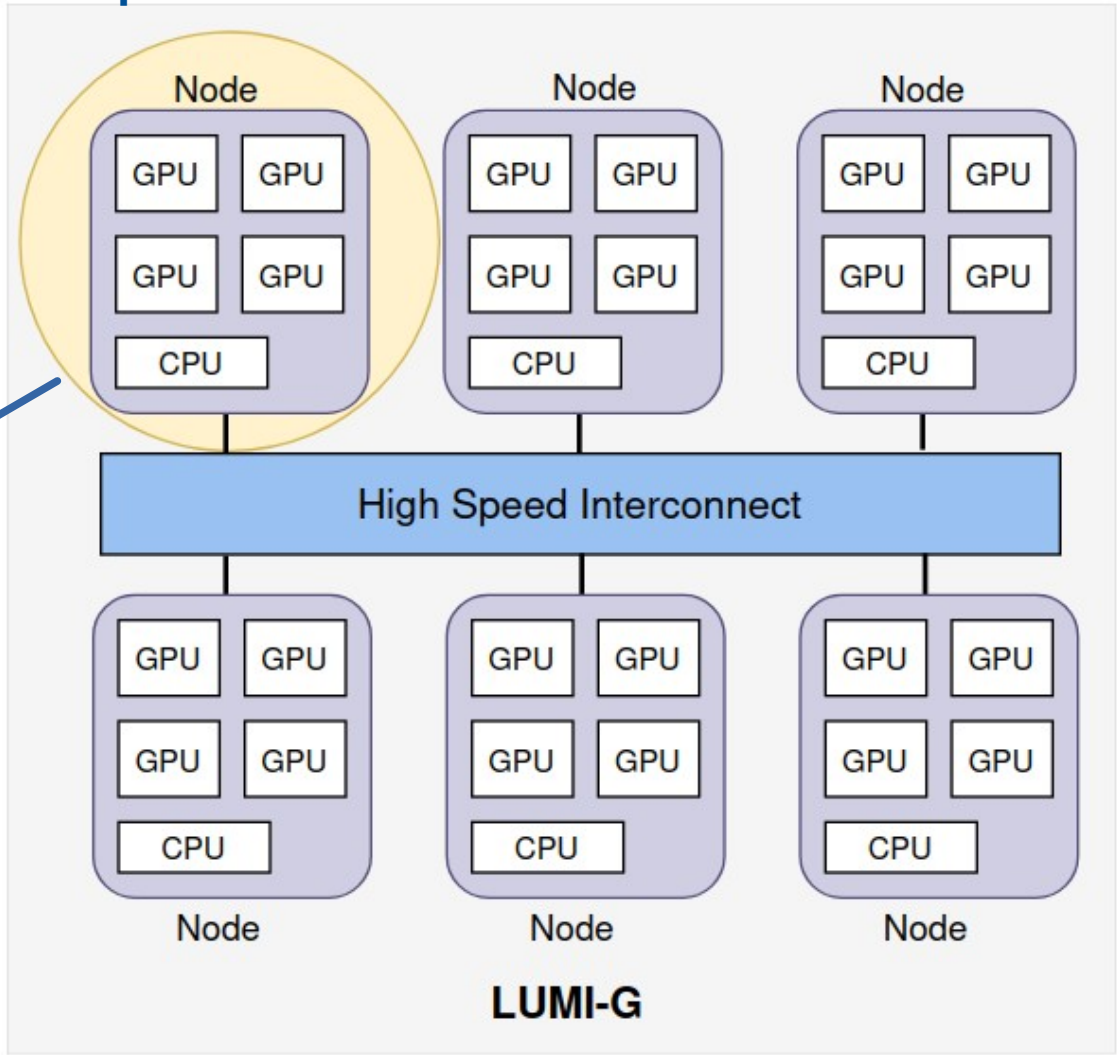
Node-Level Parallelism on LUMI Supercomputer

LUMI-G: 55,296 GPU-cores per node

4 AMD MI250X GPUs
× 2 Graphics Compute Dies (GCDs)
× 108 Compute Units (CUs)
× 64 Processing Elements (PEs)
= 55,296 GPU cores

Node-Level Parallelism

Alpaka



alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

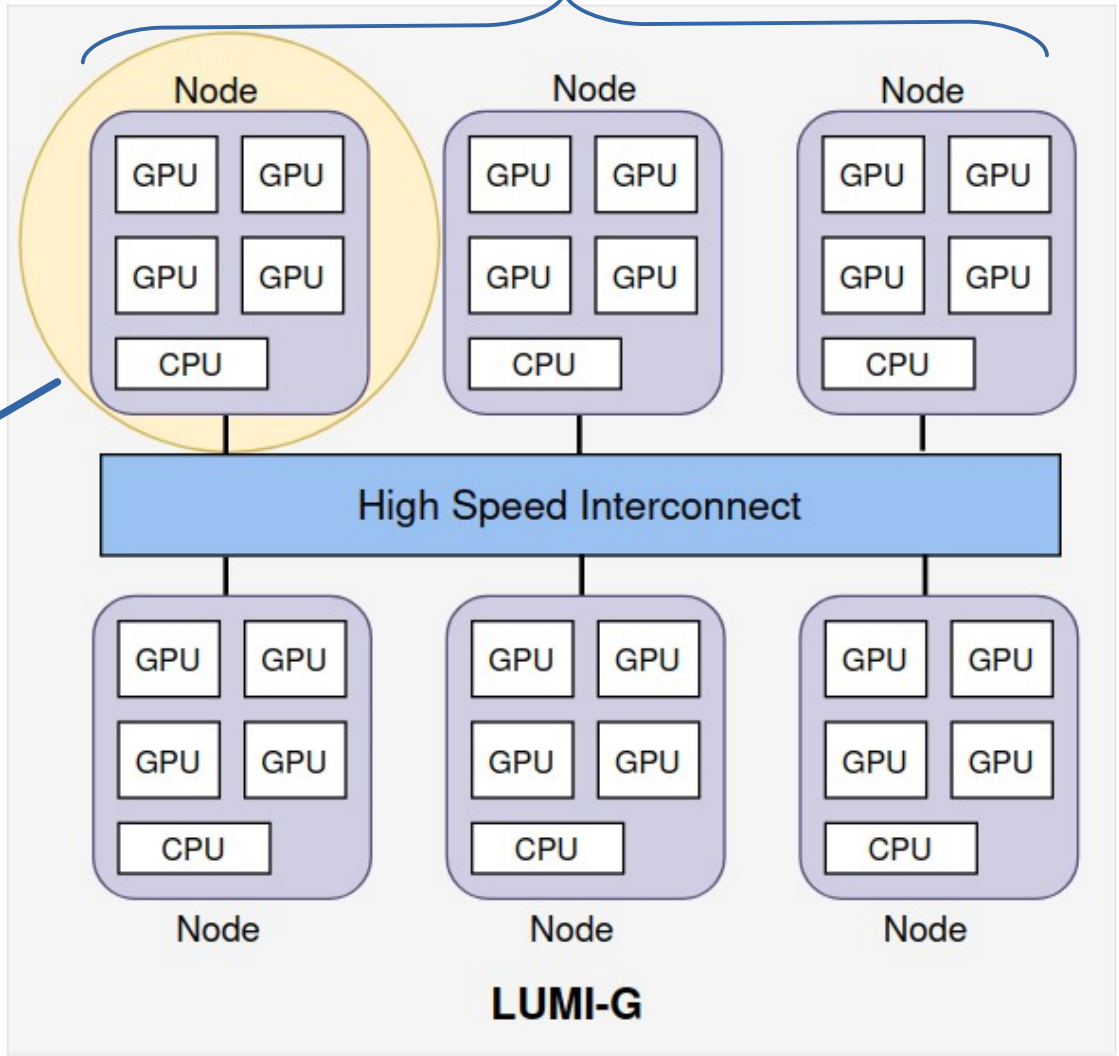
Full Parallelism on LUMI Supercomputer!

Distributed Memory Parallelism
MPI (Message Passing Interface)

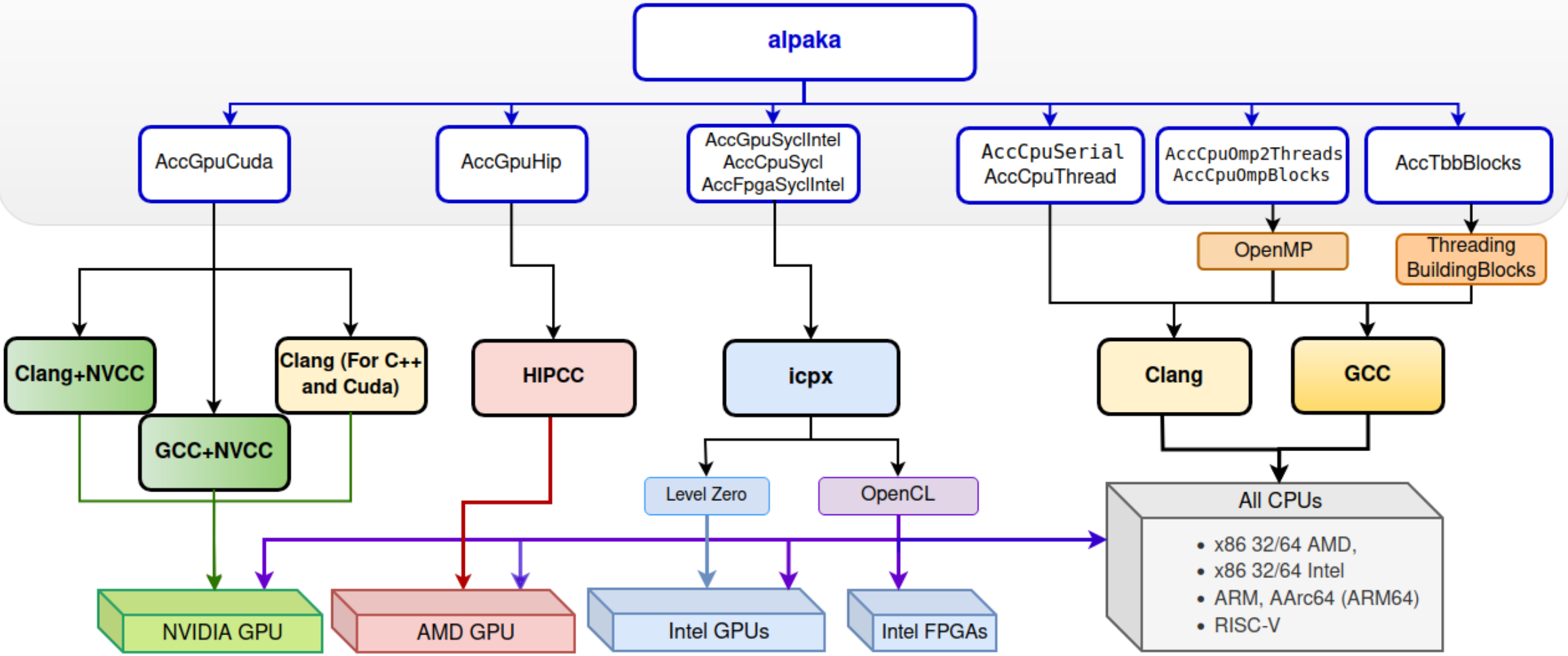
LUMI-G: 55,296 GPU-cores per node
4 AMD MI250X GPUs
× 2 Graphics Compute Dies (GCDs)
× 108 Compute Units (CUs)
× 64 Processing Elements (PEs)
= 55,296 GPU cores

Node-Level Parallelism

Alpaka



Alpaka abstraction tree



alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Profiling and Debugging Alpaka Code

Vendor tools can be used on Alpaka applications!
Because alpaka builds down to same machine-code as the vendor solutions.

- **Perfomance Analzers:** Timeline of events, system-wide analysis, bottleneck detection for softwares

- **Pofilers:** Analysis of memory usage, instruction statistics, kernel execution times threading, vectorization.

- **Sanitizers**: Detect the memory issues, data races, threading issues at runtime. Integer overflow, memory leak etc are typical errors found by sanitizers

- **Debuggers**: Interactive developer tools to debug.

ROCm/**omnitrace**

Omnitrace: Application Profiling, Tracing, and

AMD ROCm

ROCm/**omniperf**

Advanced Profiling and Analytics for AMD Hardware

AMD ROCm

Application Profiling with Intel® VTune™ and PTI-GPU

Intel® Advisor
Offload Modelling and Analysis

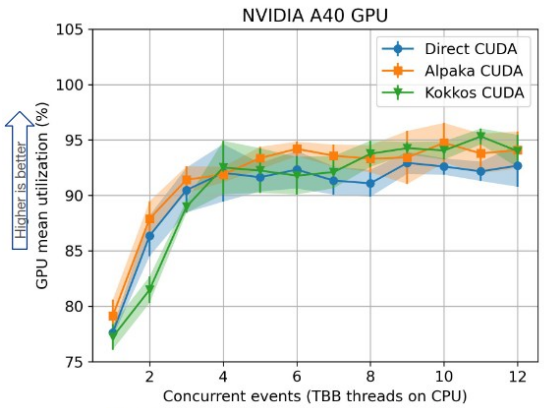
- **HIP Tools:** Rocprof, Omniperf, Omnitrace, AMD uProf, Rocgdb ...
- **SYCL Tools:** Intel VTune Profiler, Intel Advisor, Inspector debugger ...
- **Cuda Tools:** Nsight Systems, Nsight Compute, Cuda-gdb ...

NVIDIA Nsight™ Compute

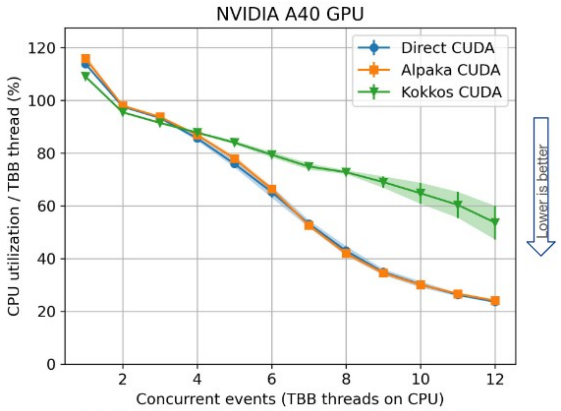
NVIDIA Nsight Systems

Performance of Alpaka

Mean GPU and CPU utilization on NVIDIA A40 GPU

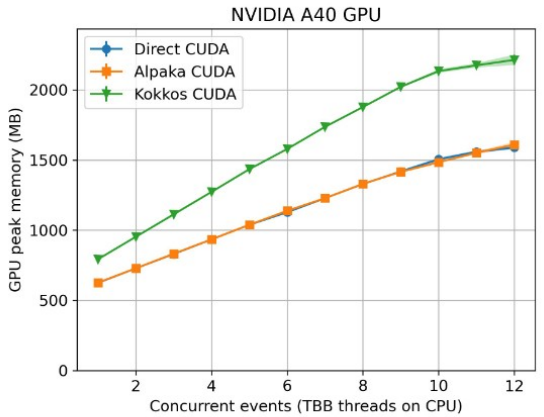


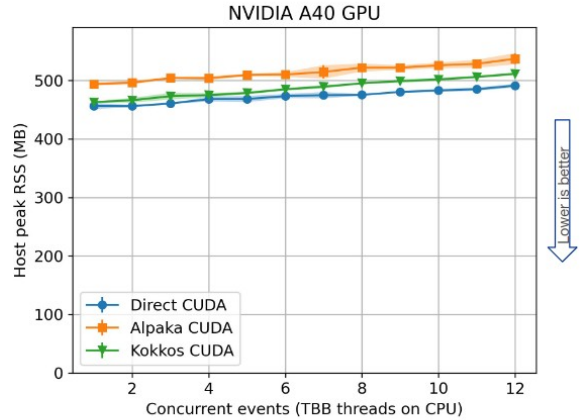
As reported by `nvidia-smi`, mean from samples



CPU efficiency normalized by number of threads

Peak memory usage on NVIDIA A40 GPU





As reported by `nvidia-smi` and `/proc/<PID>/status`. A100 shows similar behavior.

alpaka

Kokkos

Kokkos C++ Performance Portability Programming Ecosystem

195 followers https://kokkos.org crtrott@sandia.gov

NVIDIA CUDA

Source: Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code

N. Andriotis[1], A. Bocci[2], E. Cano[2], L. Cappelli[3], M. Dewing[4], T. Di Pilato[5,6], J. Esseiva[7], L. Ferragina[8], G. Hugo[2], **M. Kortelainen**[9], M. Kwok[9], J. J. Olivera Loyola[10], F. Pantaleo[2], A. Perego[11], W. Redjeb[2,12]

[1]BSC [2]CERN [3]INFN Bologna [4]ANL [5]CASUS [6]University of Geneva [7]LBNL [8]University of Bologna [9]FNAL [10]ITESM [11]University of Milano Bicocca [12] RWTH

CHEP 2023

https://indico.jlab.org/event/459/contributions/11824/attachments/9281/14171/20230511-CHEaP23_CMSPortability.pdf

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

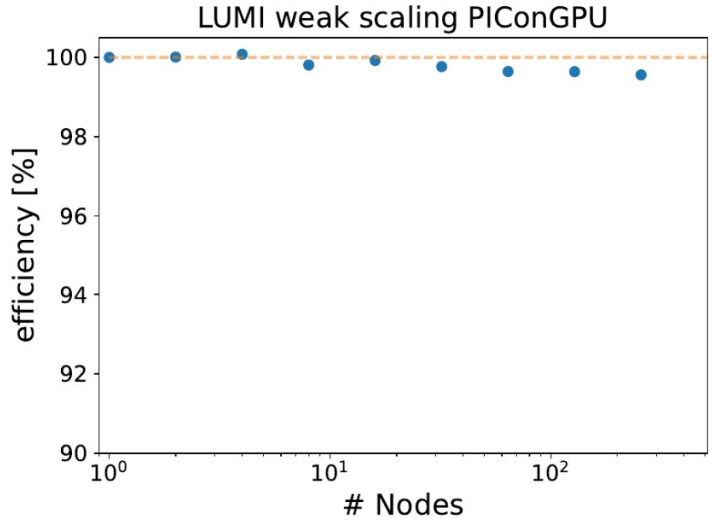
alpaka in the wild...

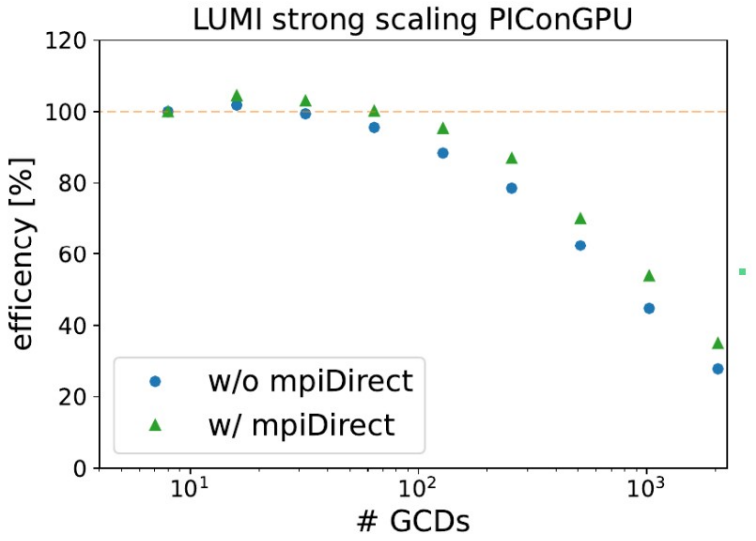
PIConGPU:

- Fully relativistic, manycore, 3D3V particle-in-cell (PIC) code
- Implements central algorithms in plasma physics
- Scalable to more than 18,000 GPUs
- Developed at Helmholtz-Zentrum Dresden-Rossendorf

PIConGPU

https://github.com/ComputationalRadiationPhysics/picongpu





Community and Long Term Support

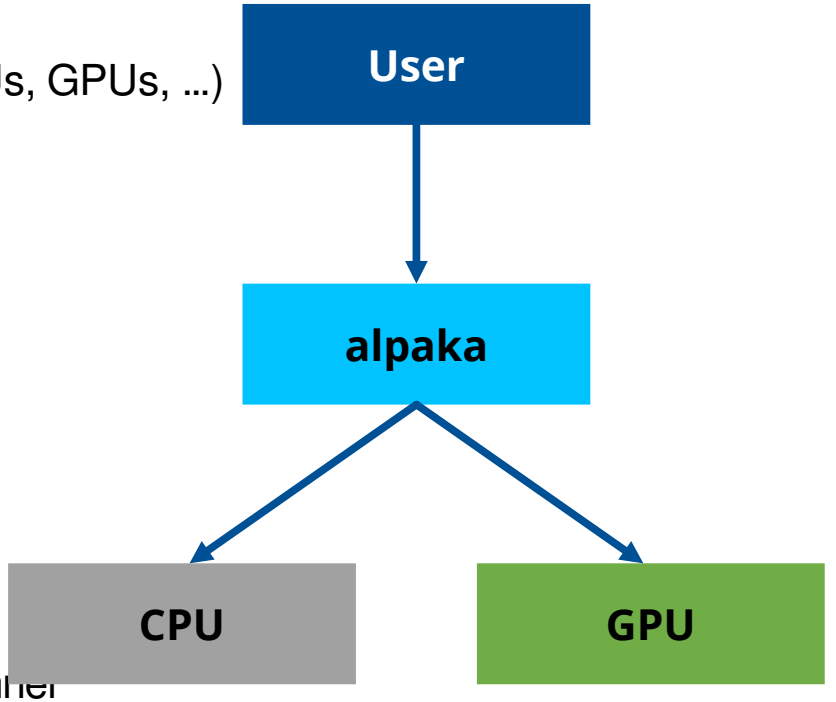
- Partners using and contributing to alpaka

- alpaka is a part of Strategic Helmholtz Program-Oriented Funding Roadmap **2027-2034**

As a summary

Without alpaka

- In HPC world, Multiple hardware types are available from different vendors (CPUs, GPUs, …)
- Platforms are not inter-operable → parallel programs are not easily portable



alpaka: one API to rule them all

- **Abstraction** (not hiding!) of the underlying hardware & software platforms
  - AMD, Nvidia, Intel GPUs, Different CPU parallelisations like TbbBlocks, OpenMP, Threads
- **Easy change of the backend in Code**
- **Builts down to the same machine code with the vendor solutions**
- **Zero abstraction overhead for Kernel execution!**
- **Heterogenous Programming**: Using different backends in a synchronized manner

If you use alpaka for your research, please cite one of the following publications:

Matthes A., Widera R., Zenker E., Worpitz B., Huebl A., Bussmann M. (2017): Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the alpaka Library. In: Kunkel J., Yokota R., Taufer M., Shalf J. (eds) High Performance Computing. ISC High Performance 2017. Lecture Notes in Computer Science, vol 10524. Springer, Cham, DOI: **10.1007/978-3-319-67630-2_36**.

E. Zenker et al., “alpaka – An Abstraction Library for Parallel Kernel Acceleration”, 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, 2016, pp. 631 – 640, DOI: **10.1109/IPDPSW.2016.50**.

Worpitz, B. (2015, September 28). Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures. Zenodo. DOI: **10.5281/zenodo.49768**.

Thank you!
You can contact us for any of your questions about alpaka!

Links

Repositories of the Workshop

- Hands-on Exercises
- Presentations

alpaka Documentation:

- Main Page: https://alpaka.readthedocs.io/en/latest/index.html
- Installation Guide
- Cheat Sheet
- CMake Variables
- API Docs

Webinar May 2024

- Webinar Slides (pdf)

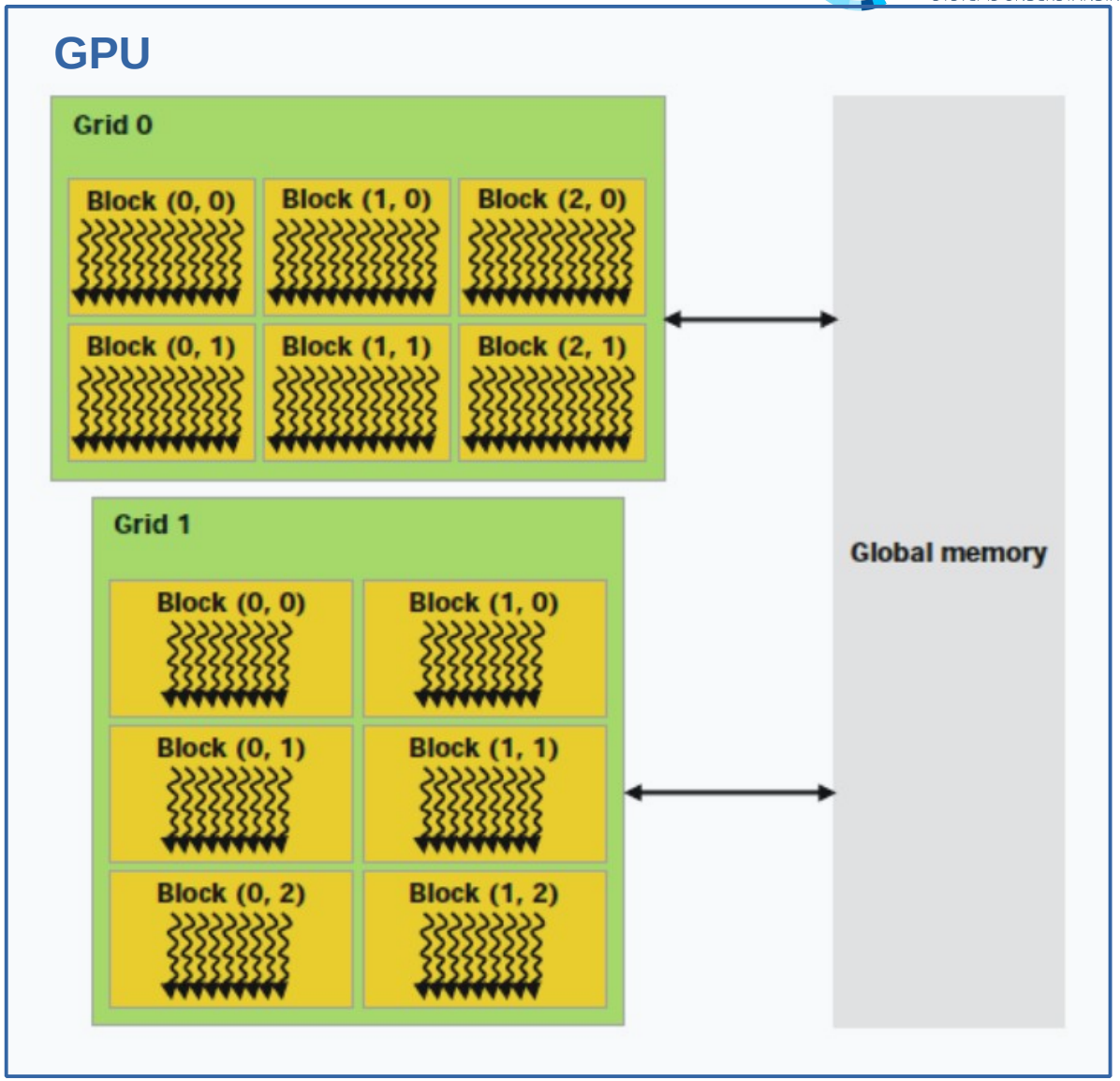
Hands-on Session

Hands-On Session1: Install alpaka and run VectorAdd example

Parallel Programming Concepts and Portable Parallelization by alpaka

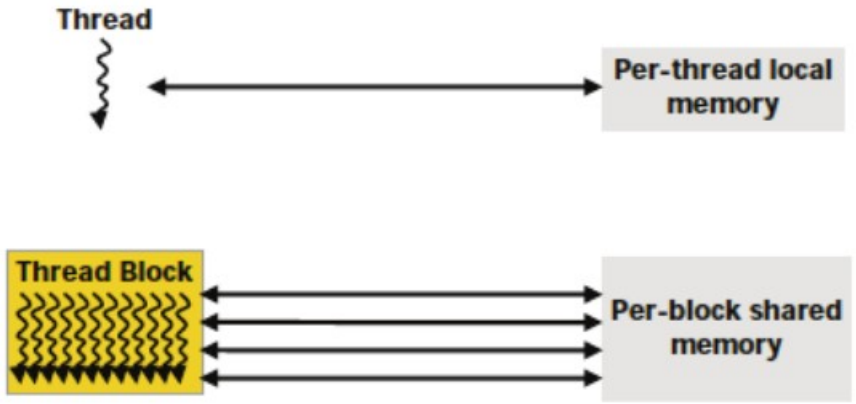
Thread Parallel Programming depends on Grid hierarchy

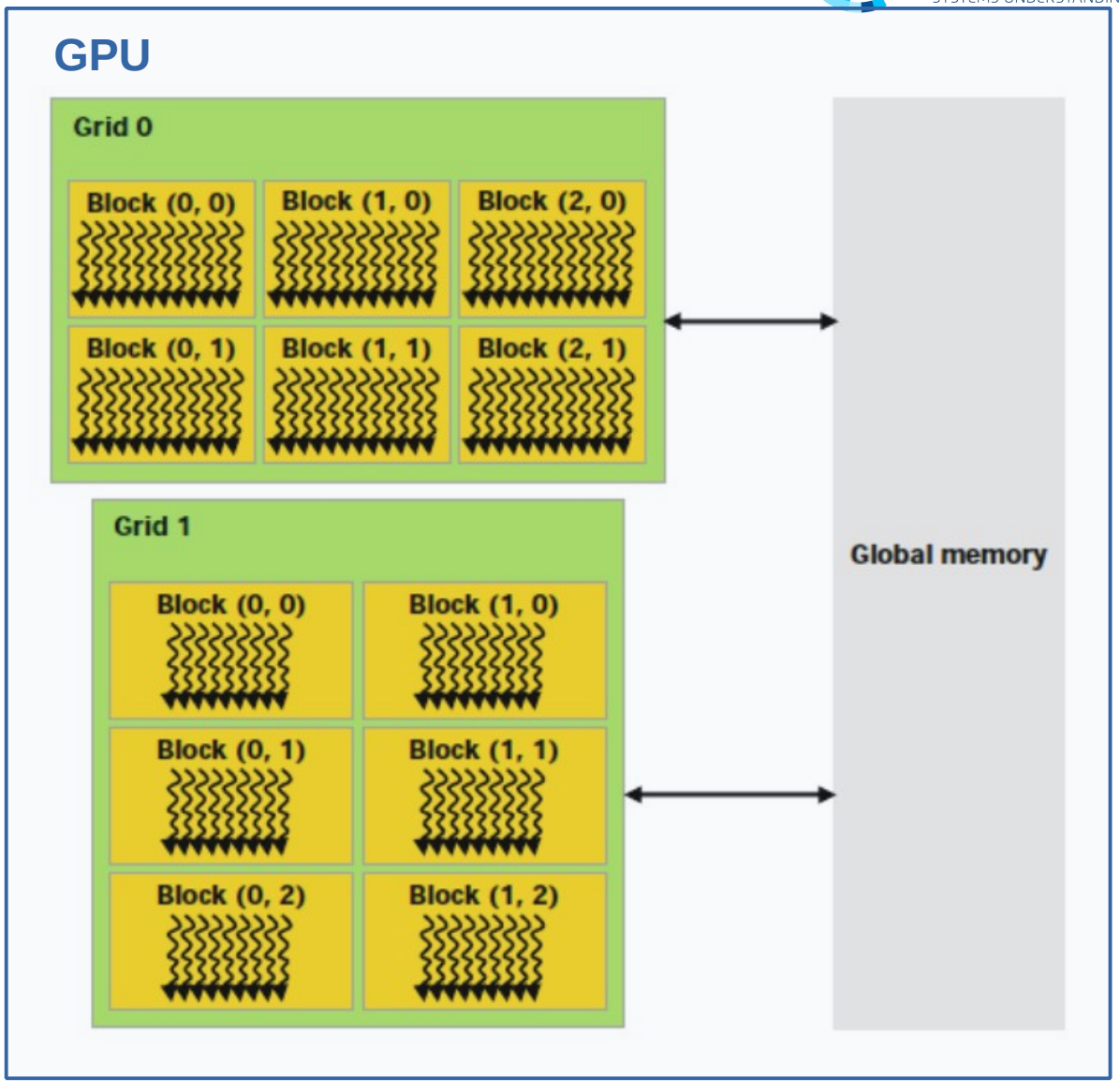
- In Parallel Programming, kernels are executed on a "grid" of multiple threads that run on a GPU
- 1D, 2D, and 3D grids are supported
- Each dimension of the grid is partitioned into equal sized "blocks" of threads
- Each block is made up of multiple "threads"
- The threads are the things that do the work



Thread Parallel Programming
depends on **Grid hierarchy**

- In Parallel Programming, kernels are executed on a "grid" of threads that run on a GPU
- 1D, 2D, and 3D grids are supported
- Each dimension of the grid is partitioned into equal sized "blocks" of threads
- Each block is made up of multiple "threads"
- The threads are the things that do the work





The architecture of a GPU

Threads are distributed among CUs
(**Compute Units or Streaming MultiProcessors**)

- Main determinants of mapping **threads** to the **CUs**:
  - Number of cores per **CU**,
  - Shared memory usage of each thread,
  - Register and local memory (lm) usage of each thread,
  - Limits on Threads per CU
  - **Memory latencies**
  - **Memory sizes**

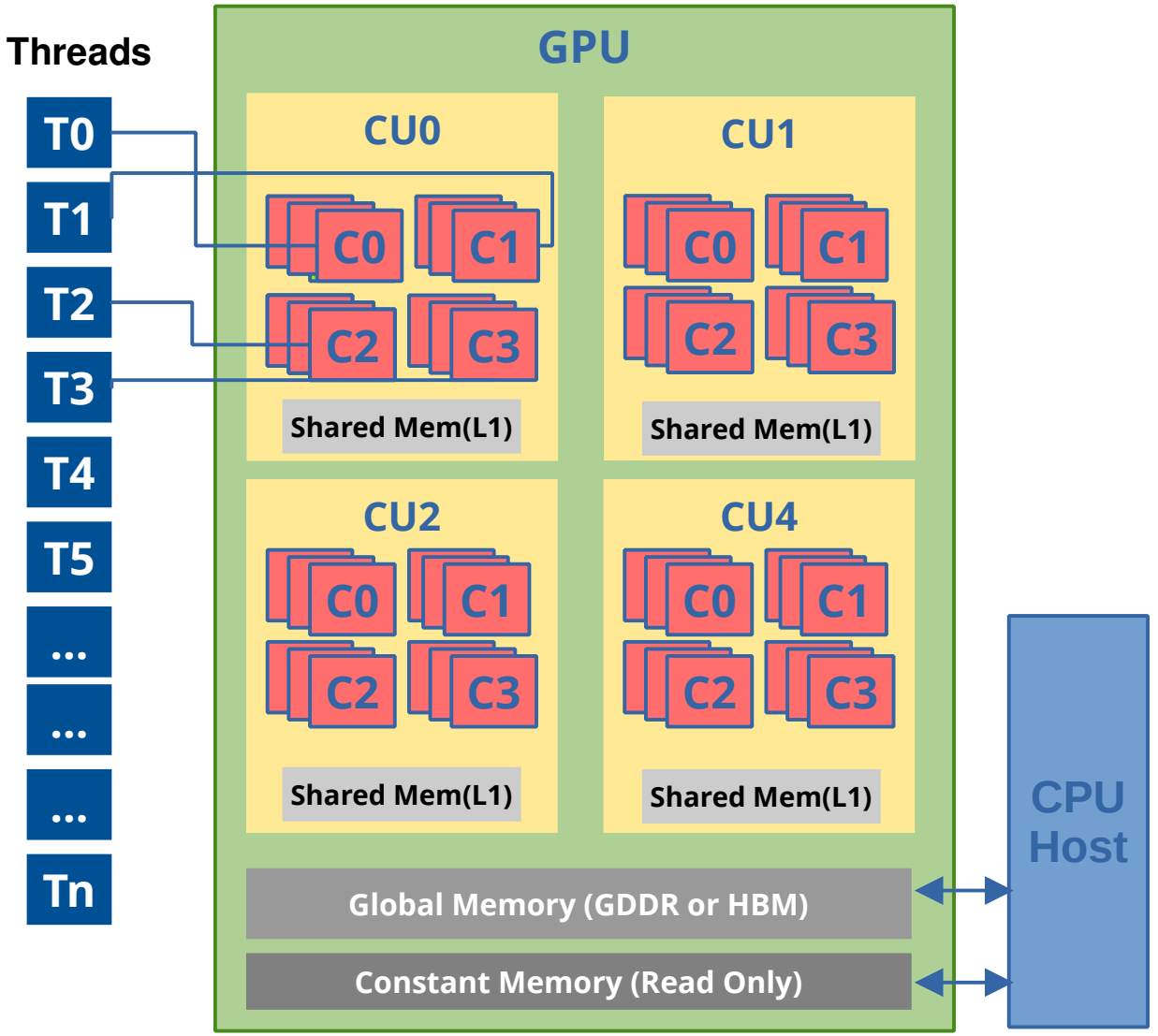
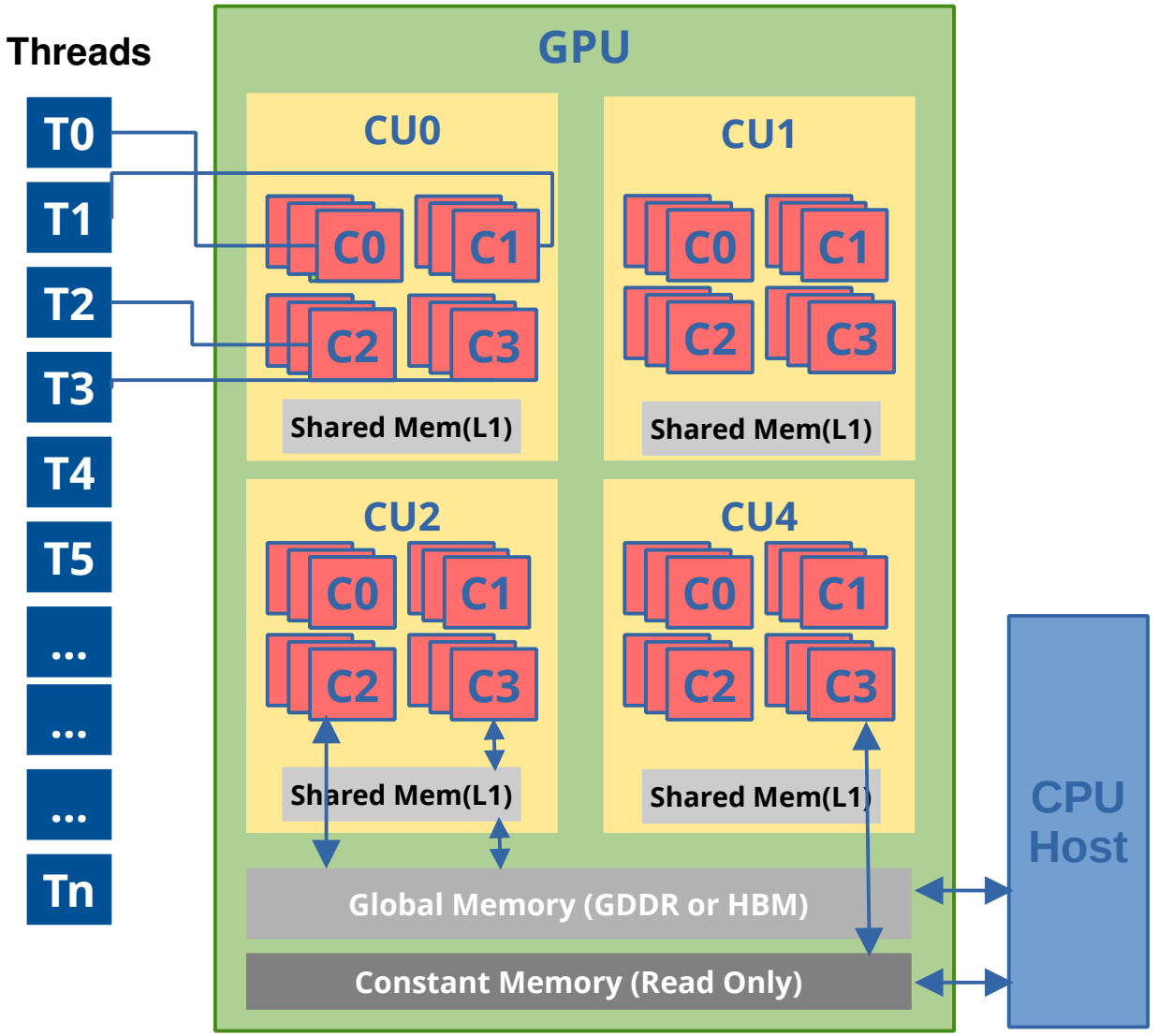


Table: Summary of All Memory Types, Latency, Bandwidth, Size, and PCIe Data Transfer for AMD Instinct MI100 and AMD Radeon RX 6800 XT

Memory Type / Operation	Bandwidth	Latency (Approximate)	Memory Size	Notes
Global Memory Access	~224 GB/s to ~1.23 TB/s (HBM2 or GDDR6)	100s to 1000s of cycles[1]	16 GB (GDDR6) to 32 GB (HBM2)	HBM2 for HPC workloads; GDDR6 for gaming workloads.
Constant Memory Access	Potentially hundreds of GB/s (cached)	10s to 100s of cycles (cached)[1]	Typically 64 KB	Cached in L1 or L2; efficient for shaders or frequently accessed constants. Uncached access has higher latency.
L1 (Shared Memory) Access	Very High (On-chip memory)	10-50 cycles[1]	128 KB per CU	On-chip shared memory for inter-thread communication; very fast, with latency depending on bank conflicts and access patterns.
Copying Data (CPU to GPU via PCIe)	Up to 16 GB/s (PCIe 4.0 x16)	Several µs to ms (depending on data size)	Limited by system RAM and VRAM	Limited by PCIe bandwidth; slower than on-chip memory operations. Involves transfer setup overhead.

| Footnote 1. One cycle is 0.5 nanoseconds at a 2 GHz frequency. |



Thread-Parallel Programming Terminology

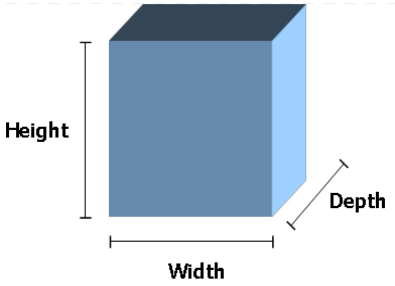
Indexing of threads. Each thread has an index accessible in kernel.

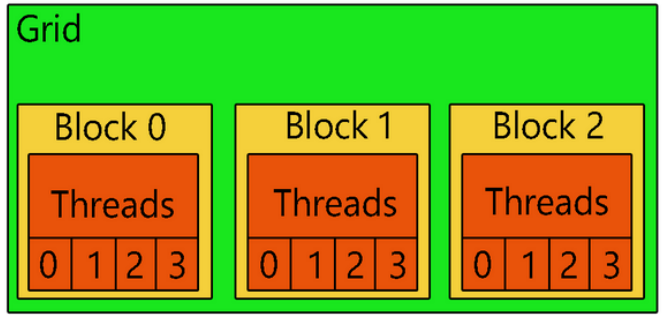
In other words kernel implicitly knows the index of thread which runs it.

Dimensions: Set of dimension names. **{Z-dimension, Y-dimension, X-dimension}**

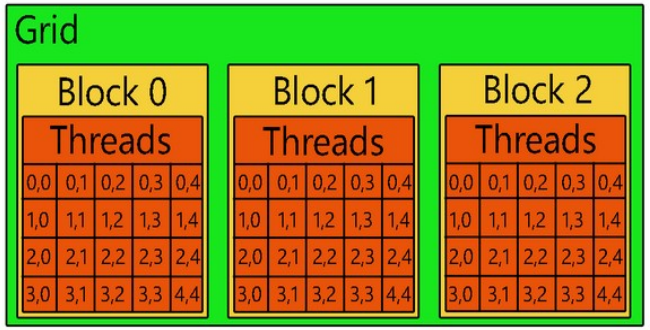
Extent: A vector representing the sizes along dimensions.

- In 3D **{Depth,Height, Width}. The order is important.**





Grid-Block Extent: {3}
Block-Thread Extent: {4}

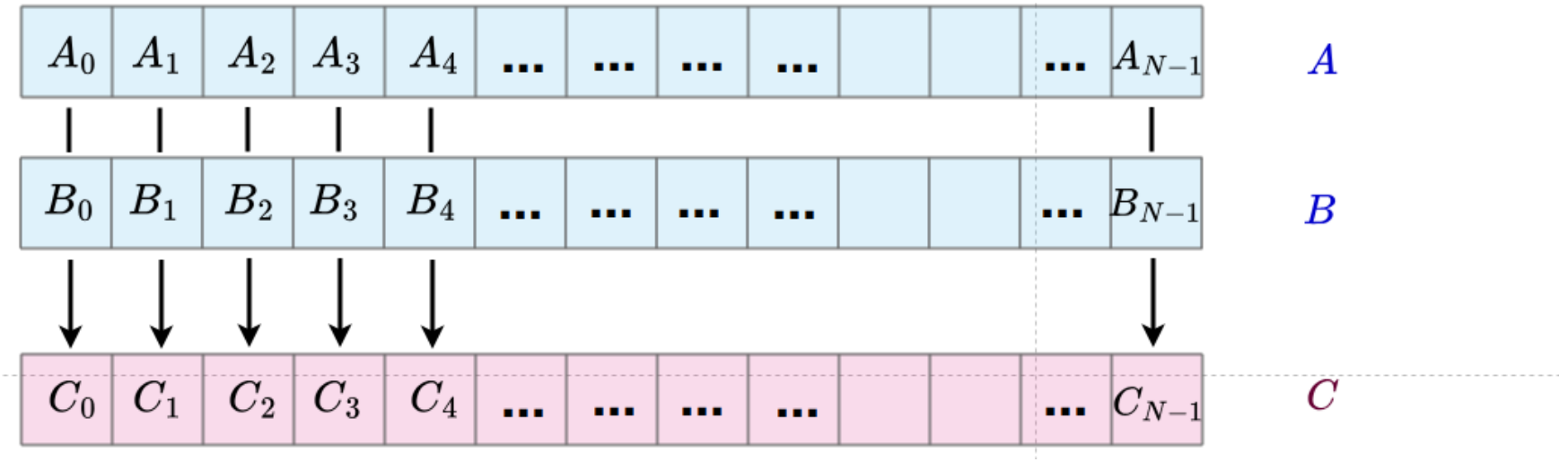


Grid-Block Extent: {1,3}
Block-Thread Extent: {4,5}

Alpaka	HIP (AMD)	SYCL (Intel)	Cuda (NVIDIA)
Grid	Grid	NDRange/Range	Grid
Block	Workgroup	Workgroup	Block
Warp	Wavefront	Subgroup	Warp
Thread	Work-item	Work-item	Thread

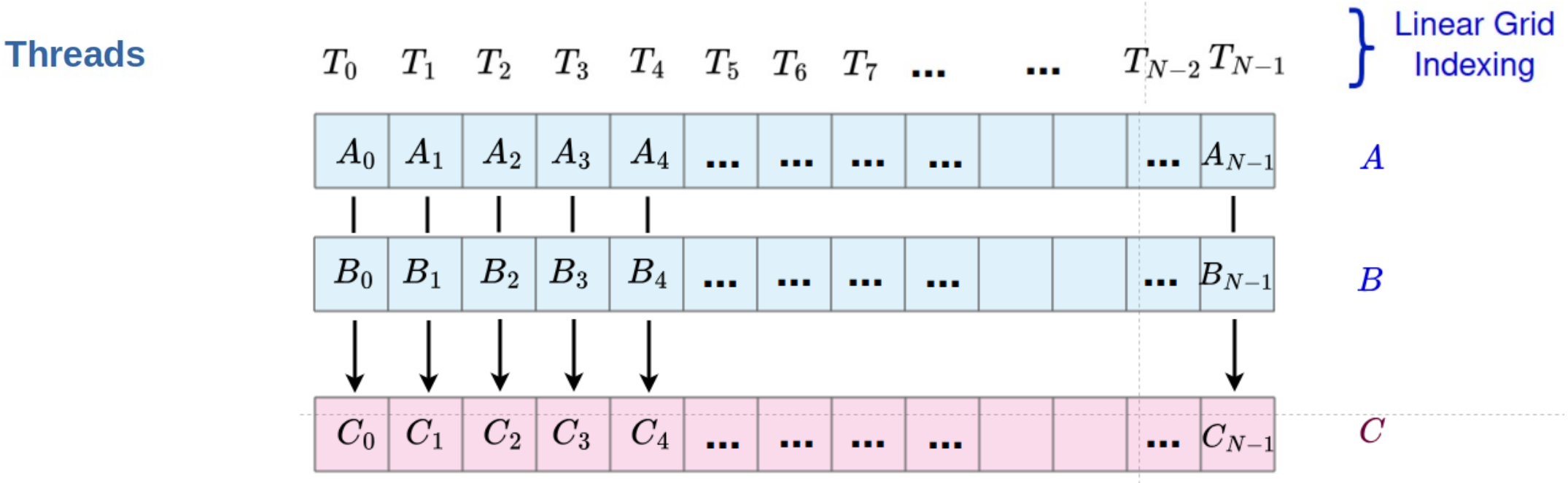
How to paralelize Vector addition problem

$$\vec{A} + \vec{B} = \vec{C}$$



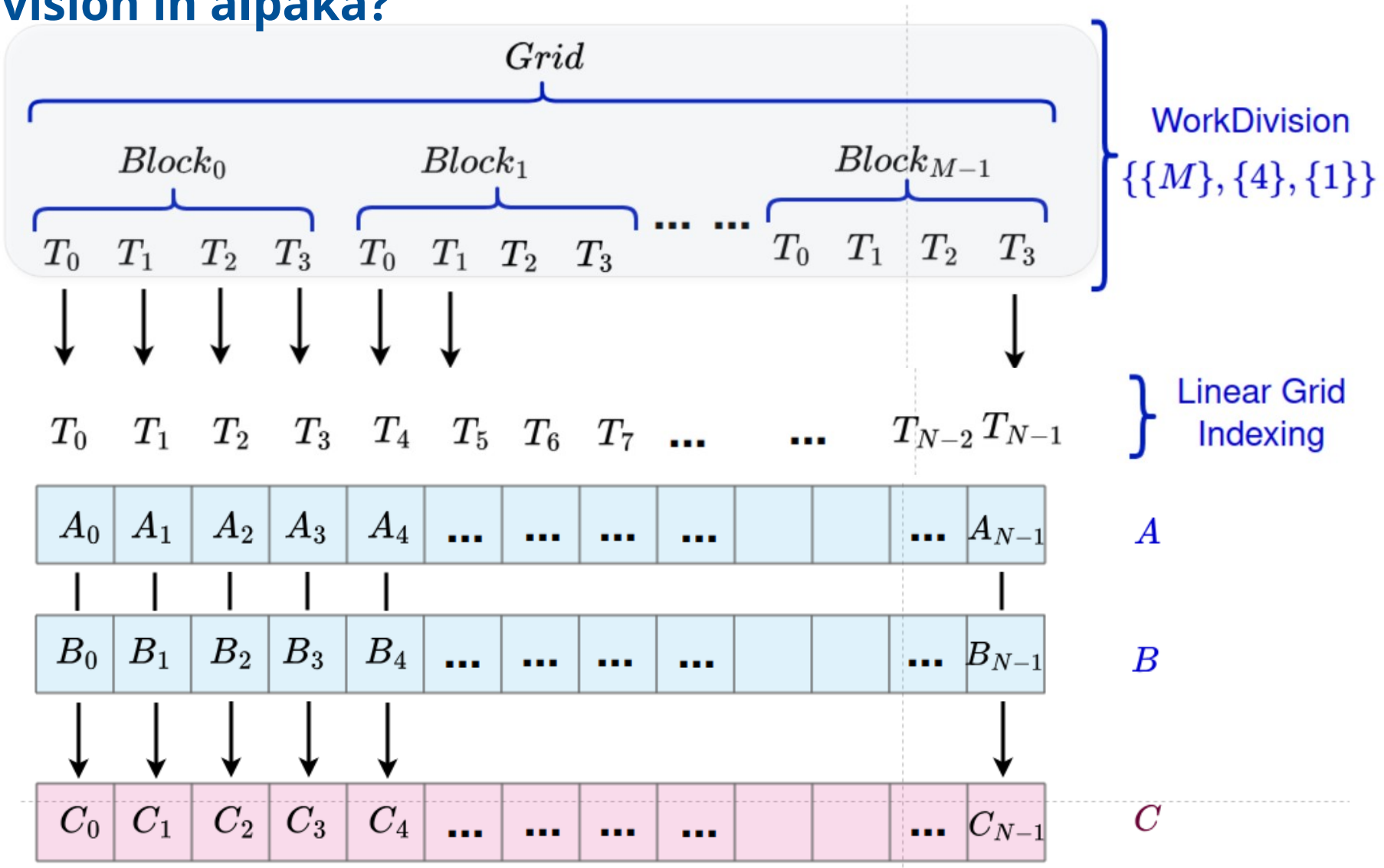
How to paralelize Vector addition problem

$$\vec{A} + \vec{B} = \vec{C}$$



What is work division in alpaka?

$$\vec{A} + \vec{B} = \vec{C}$$



alpaka

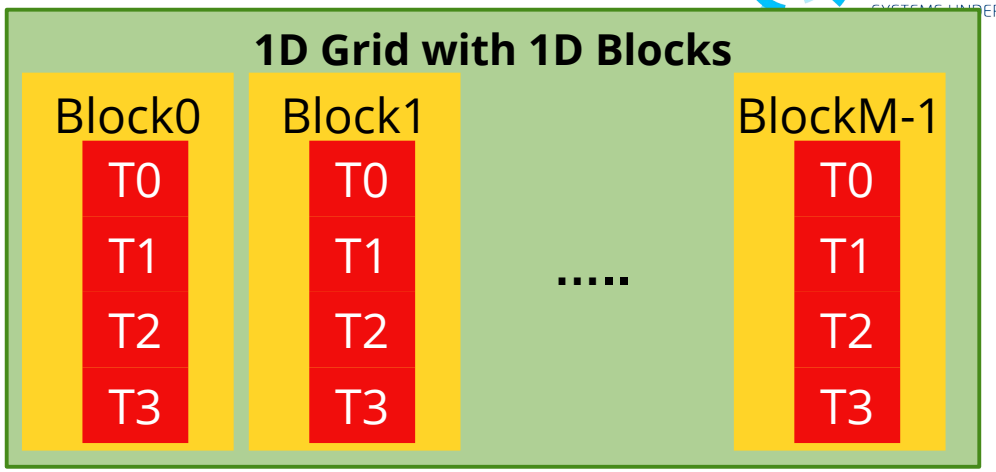
CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

WorkDivision in Alpaka has 3 vectors

1D WorkDivision

- Grid-Block Extent 1D vector = {M}
- Block-Thread Extent 1D vector {4u}
- Thread-Elem Extent 1D vector = {1u}

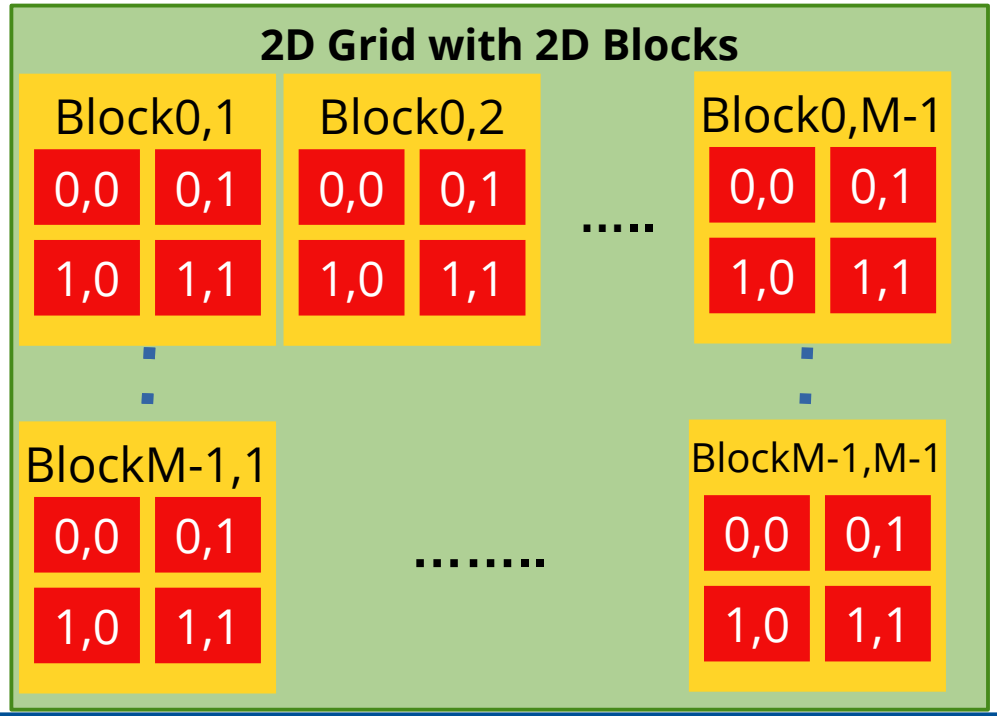
1D WorkDivision {{M}, {4u}, {1u}}



2D WorkDivision

- Grid-Block Extent 2D vector = {M,M}
- Block-Thread Extent 2D vector = {2u,2u}
- Thread-Elem Extent 2D vector = {1u,1u}

2D WorkDivision {{M,M}, {2u,2u}, {1u,1u}}



Sequentiential and Parallel Vector Sums

// **Sequential Code for CPU**

```
std::vector<int> A = {...};
std::vector<int> B = {...};
std::vector<int> C;   C.resize(N);
for (int i = 0; i < N; ++i) {
  C[i] = A[i] + B[i];
}
```

// **OpenMp Code for CPU**

```
std::vector<int> A = {...};
std::vector<int> B = {….};
std::vector<int> C;   C.resize(N);

#pragma omp parallel for
   for (int i = 0; i < N; ++i) {
       C[i] = A[i] + B[i];
   }
```

```
// Define kernel as a lambda or function object
 auto addKernel = [] ALPAKA_FN_ACC (TAcc const& acc,
    int * A, int* B, int* C, unsigned const& N) -> void
{
    // Get the thread index
     auto const threadIdx = alpaka::getIdx<alpaka::Grid,
alpaka::Thread>(acc)[0];
    // Vector sum
     C[threadIdx] = A[threadIdx] + B[threadIdx];
};

// Call the kernel
// 1. Let alpaka calculate good block and grid sizes given our full problem
extent
 auto const workDiv = alpaka::getValidWorkDiv(kernelCfg, devAcc,
addKernel, ptrA, ptrB, ptrC, N);
// 2. Call kernel
 using Acc = alpaka::AccGpuHipRt<Dim, Idx>;
 alpaka::exec<Acc>(queue, workDiv, addKernel, ptrA, ptrB, ptrC,
N );
```

alpaka Kernel and thread indices

- Contains the algorithm that is run by each thread
- Can be thought as “**code inside a hypothetical parallel for loop** where **indices are magically provided**”
- Kernel can acces to the thread index currently running the kernel
- Usually thread indexes are used to access to the specific part of the data
- alpaka Kernels are functors or lambdas
- Arguments can be pointers and trivially copyable types
- Returns void
- Has an argument of type Accelerator

function object as kernel

```
struct HelloWorldKernel {

    template <typename Acc>
    ALPAKA_FN_ACC void operator()(Acc const & acc) const {

        uint32_t threadIdx = alpaka::getIdx<Grid, Threads>(acc)[0];
        printf("Hello, World from alpaka thread %u!\n", threadIdx);
    }
};
     using Acc = acc::AccGpuHipRt<Dim, Idx>;
    HelloWorldKernel helloWorlKernel;
    alpaka::exec<Acc>(queue, workDiv, helloWorlKernel);
```

Function object as kernel

Select backend type

Execute kernel in Grid

lambda function as kernel

```
    auto kernelLambda = [] ALPAKA_FN_ACC(Acc const& acc, size_t const nExclamationMarksAsArg) -> void
    {
        auto globalThreadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc);
        auto globalThreadExtent = alpaka::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc);
        auto linearizedGlobalThreadIdx = alpaka::mapIdx<1u>(globalThreadIdx, globalThreadExtent);

        printf(
            "[z:%u, y:%u, x:%u][linear:%u] Hello world from a lambda",
            static_cast<unsigned>(globalThreadIdx[0]),
            static_cast<unsigned>(globalThreadIdx[1]),
            static_cast<unsigned>(globalThreadIdx[2]),
            static_cast<unsigned>(linearizedGlobalThreadIdx[0]));

        for(size_t i = 0; i < nExclamationMarksAsArg; ++i)
        {  printf("!"); }

        printf("\n");
    };
    using Acc = acc::AccGpuHipRt<Dim, Idx>;
    // Run kernel

    alpaka::exec<Acc>(queue, workDiv, kernelLambda, nExclamationMarks);
```

Lambda as kernel

Select backend type

Execute kernel in Grid

Alpaka Function Qualifiers

- ALPAKA_FN_ACC
  - For functions called directly from accelerator code
  - For kernels
- ALPAKA_FN_HOST_ACC
  - For functions which can be called from accelerator code or host code
- ALPAKA_FN_HOST
  - shows the code will only run on the host side.
- No Qualifier:
  - Same as using ALPAKA_FN_HOST

- **Kernel**

```
class VectorAddKernel
{ public:
   template<typename TAcc>
   ALPAKA_FN_ACC auto operator()( TAcc const& acc,   // the accelerator
         int * A, int*  B,  int*  C,  unsigned const& N) const -> void
   {  // Get the thread index
        auto const threadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Thread>(acc)
[0];
        C[threadIdx] = A[threadIdx] + B[threadIdx];
   }  };
```

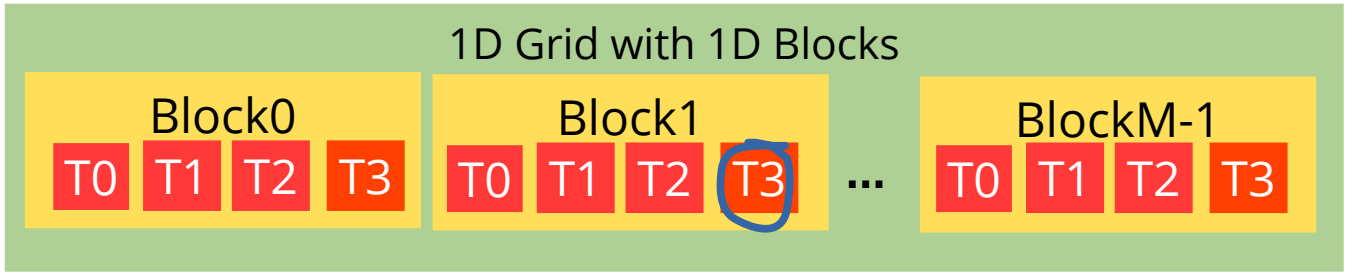
- **Function, callable from Host and Acc**

```
template<typename TAcc, typename T>
   ALPAKA_FN_HOST_ACC auto divides(TAcc&, T const& arg1, T const& arg2)
   {
         return arg1 / arg2;
   }
```

- **Function, only callable from Host**

```
ALPAKA_FN_HOST static auto getAccDevProps(DevCpu const& dev)
{ .... }
```

Getting thread index in Kernel



Obtaining the indices of threads

```
// Hip or Cuda way  4 * 1 + 3
int threadIdx = blockIdx.x * blockDim.x + threadIdx.x;
```

- Index of Thread on the Grid:

```
auto gridThreadIndex = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc);
// gridThreadIndex is {7}, alpaka::Grid and alpaka::Threads are alpaka defined types
```

- Index of Thread on a Block:

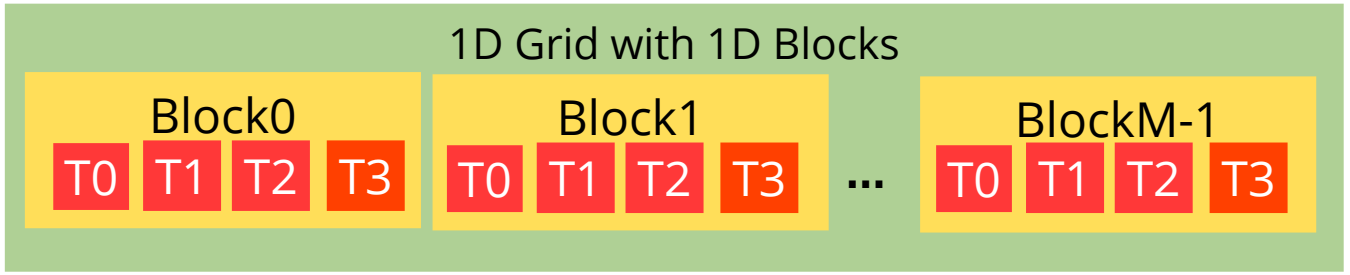
```
auto theradBlockIndex = alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc);
```

- `// threadBlockIndex is {3}, alpaka::Block is an alpaka type`

- Index of Block on the Grid:

```
auto blockGridIndex = alpaka::getIdx<alpaka::Grid, alpaka::Blocks>(acc);
// the blockGridIndex is {1}
```

Getting the extents



Obtaining the extents of grids and blocks

- Extent of Grid in number of threads:

```
auto gridThreadExtent = workdiv::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc);
// gridThreadExtent is {M*4}, alpaka::Grid and alpaka::Threads are alpaka defined types
```

- Extent of Block in number of threads:

```
auto blockThreadExtent = alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(acc);
// blockThreadExtent is {4}
```

- Extents of Grid in number of blocks:

```
auto gridBlockExtent = alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(acc);
// gridBlockExtent is {M}
```

Getting Thread Index in Kernel

Obtaining the indices of threads

- Index of Thread on the Grid:

```
auto gridThreadIndex = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc);
// gridThreadIndex is {1,9}
```

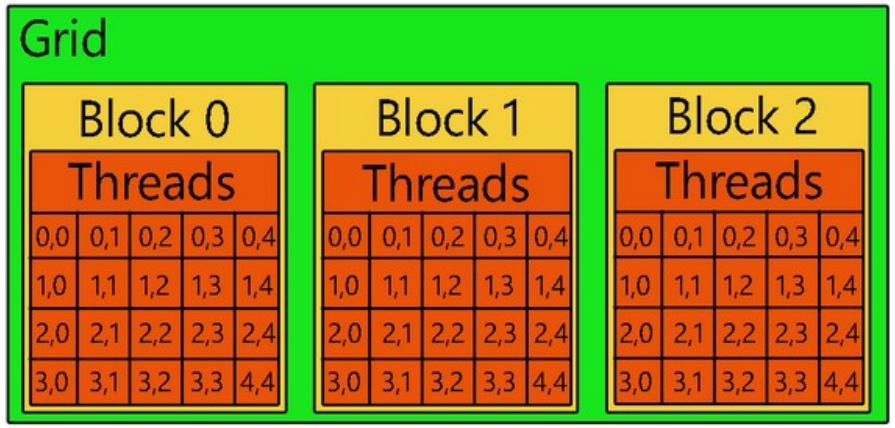
- Index of Thread on a Block:

```
auto theradBlockIndex = alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc);
// threadBlockIndex is {1,4}
```

- Index of Block on the Grid:

```
auto blockGridIndex = alpaka::getIdx<alpaka::Grid, alpaka::Blocks>(acc);
// the blockGridIndex is {1}
```

Getting the extents



Obtaining the extents of grids and blocks

- Extents of Grid in terms of threads:

```
auto gridThreadExtent = workdiv::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc);
// gridThreadExtent is {4,15}, alpaka::Grid and alpaka::Threads are alpaka defined types
```

- Extents of Block in terms of threads:

```
auto blockThreadExtent = alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(acc);
// blockThreadExtent is {4,5}
```

- Extents of Grid in terms of blocks:

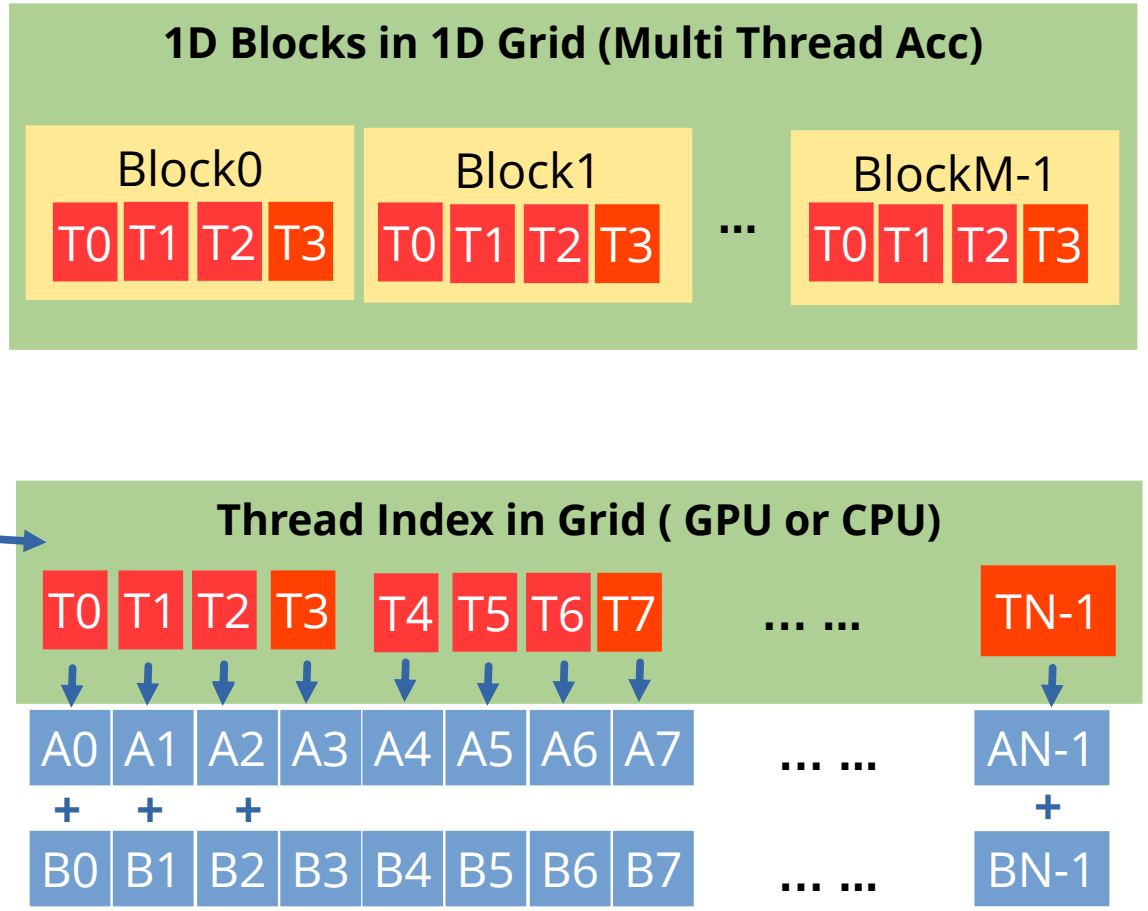
```
auto gridBlockExtent = alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(acc);
// gridBlockExtent is {1,3}
```

Alpaka Kernel

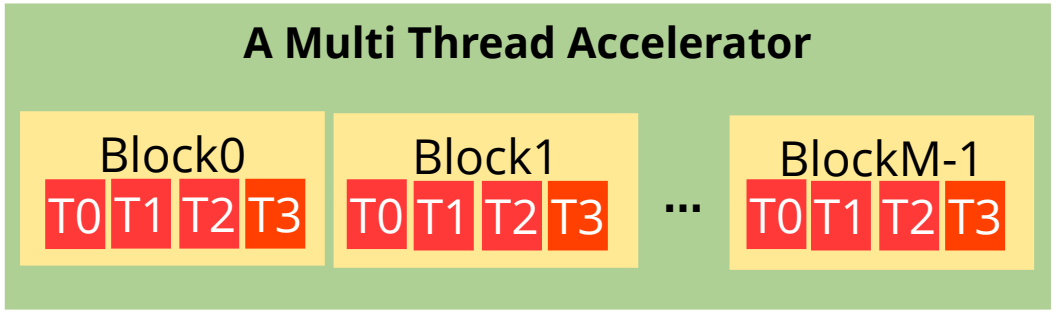
```
class VectorAddKernel
{ public:
   template<typename TAcc>
   ALPAKA_FN_ACC auto operator()( TAcc const& acc,   // the accelerator
      int * A, int*  B,  int*  C,  unsigned const& N) const -> void
   {  // Get the thread index
   auto const threadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Thread>(acc)[0];
   C[threadIdx] = A[threadIdx] + B[threadIdx];
   }  };
```

Alpaka Kernel Call

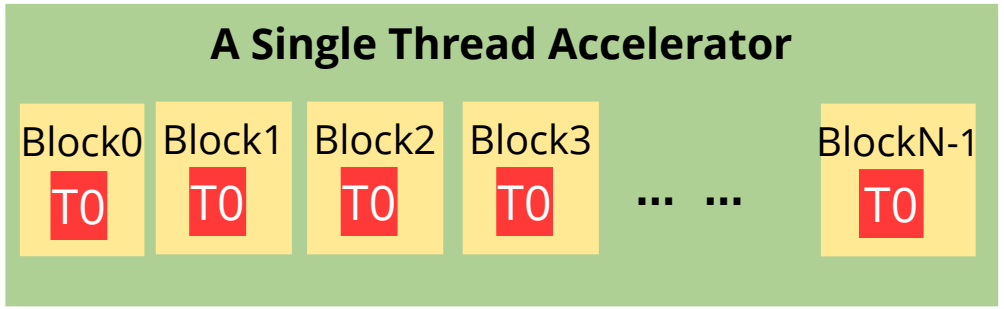
```
    auto const workDiv = alpaka::WorkDivMembers<Dim, Idx>({M}, {4}, {1});
  // Launch the vector addition kernel
   VectorAddKernel vectorAddKernel;
   alpaka::exec<Acc>(queue, workDiv, vectorAddKernel, bufA, bufB, bufC);
```



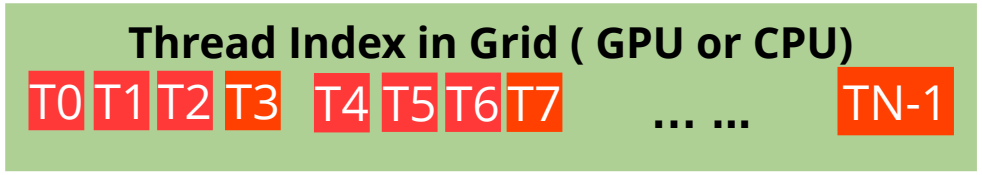
Should I change work-division in code if I change the accelerator?



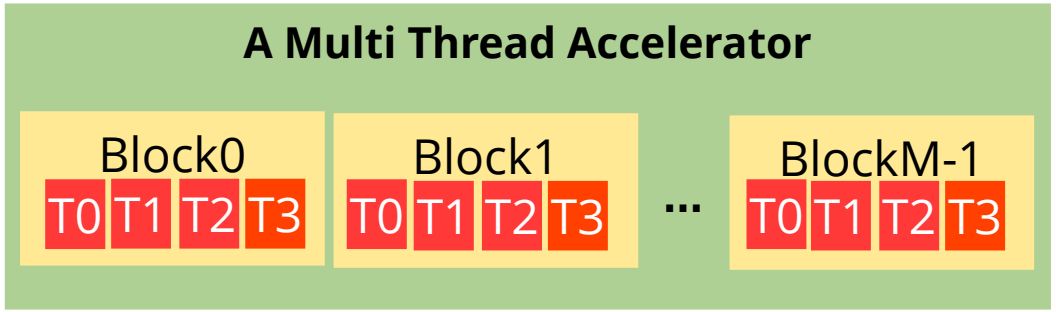
auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({M}, {4}, {1});



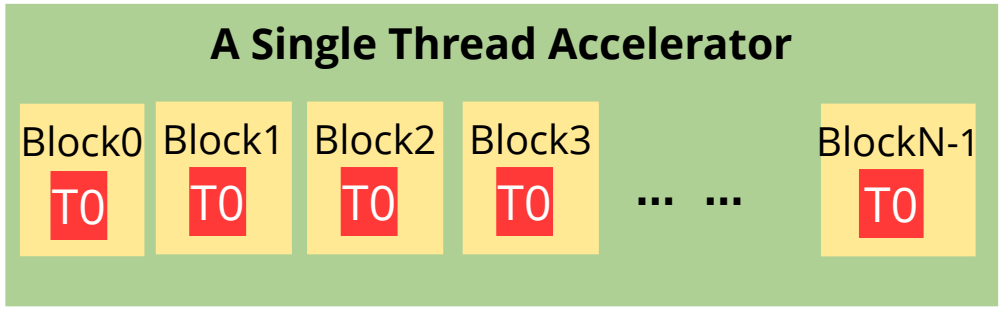
auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({N}, {1}, {1});



Should I change work-division in code if I change the accelerator?

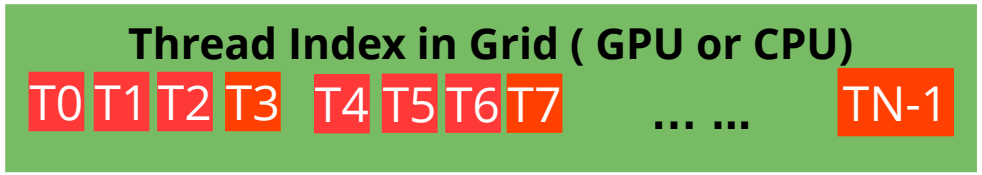


auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({M}, {4}, {1});



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({N}, {1}, {1});

Alpaka Kernel

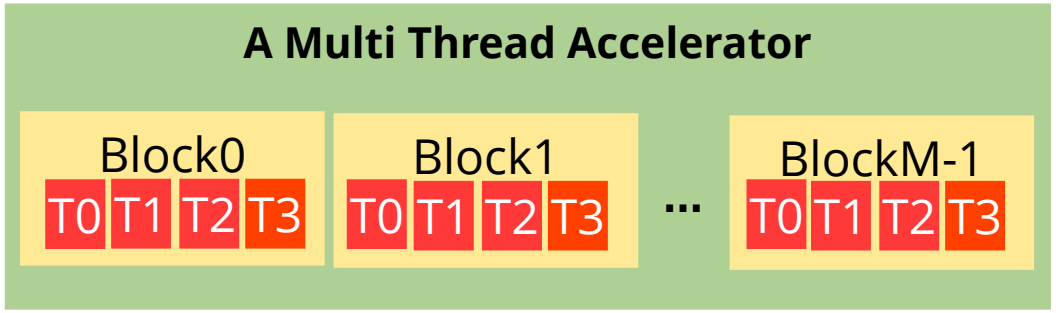



```
class VectorAddKernel
{ public:
   template<typename TAcc>
   ALPAKA_FN_ACC auto operator()( TAcc const& acc,   // the accelerator
      int * A, int*  B,  int*  C,  unsigned const& N) const -> void
   {  // Get the thread index, only first element of the vector needed since we are in 1D
   auto const threadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Thread>(acc)[0];
   C[threadIdx] = A[threadIdx] + B[threadIdx];
   }  };
```

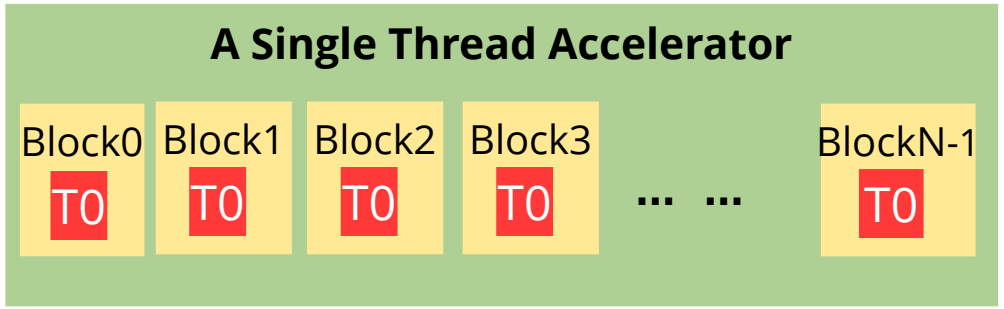
Alpaka Kernel Call

```
// since accelerator can be a CPU or GPU don't use constant workDiv
 auto workDiv = ?
  //  Launch the vector addition kernel
  VectorAddKernel vectorAddKernel;
  alpaka::exec<Acc>(queue, workDiv, vectorAddKernel, bufA, bufB, bufC);
```

Should I change work-division in code if I change the backend?



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({M}, {4}, {1});



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({N}, {1}, {1});

Alpaka Kernel

class **VectorAddKernel**

{ public:

template<typename TAcc>

ALPAKA_FN_ACC auto **operator**()(TAcc const& acc, // the accelerator

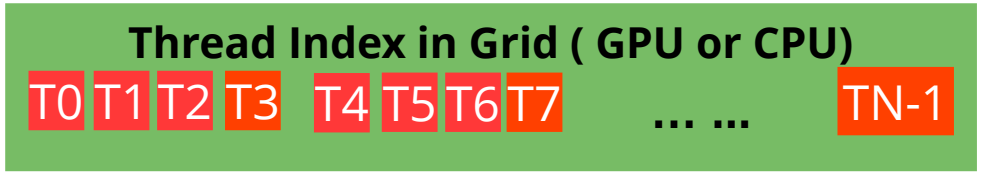
int * A, int* B, int* C, unsigned const& N) const -> void

{ // Get the thread index, only first element of the vector needed since we are in 1D

auto const threadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Thread>(acc)[0];

C[threadIdx] = A[threadIdx] + B[threadIdx];

} };



Alpaka Kernel Call

// since accelerator can be a CPU or GPU don't use constant workDiv

auto workDiv = ? **??**

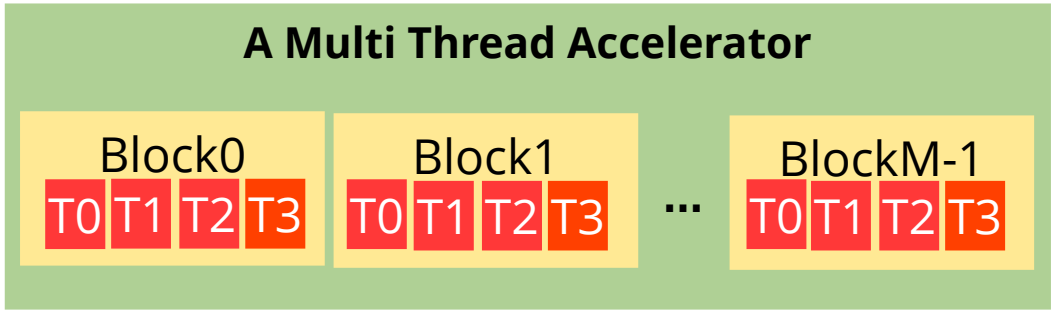
// Launch the vector addition kernel

VectorAddKernel vectorAddKernel;

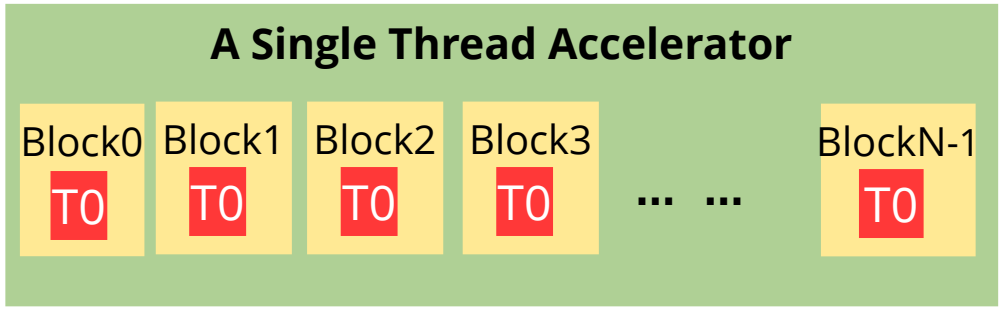
alpaka::exec<Acc>(queue, workDiv, vectorAddKernel, bufA, bufB, bufC);

OK

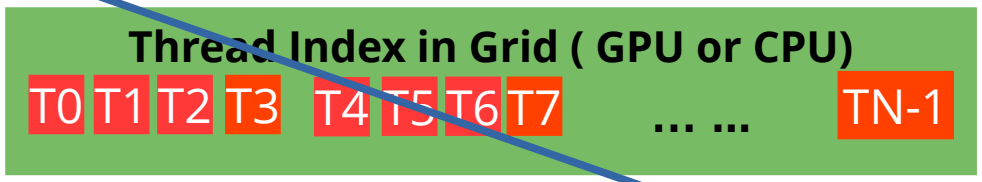
alpaka::getValidWorkDiv



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({M}, {4}, {1});



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({N}, {1}, {1});



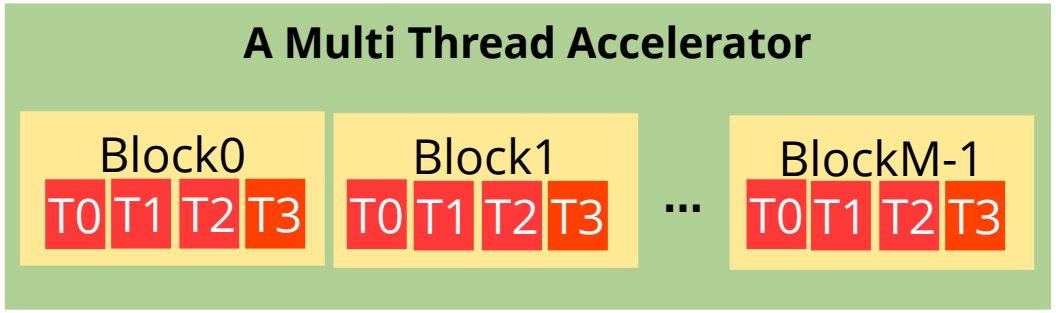
Alpaka Kernel

```
class VectorAddKernel
{ public:
   template<typename TAcc>
   ALPAKA_FN_ACC auto operator()( TAcc const& acc,   // the accelerator
      int * A, int*  B,  int*  C,  unsigned const& N) const -> void
   {  // Get the thread index, only first element of the vector needed since we are in 1D
   auto const threadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Thread>(acc)[0];
   C[threadIdx] = A[threadIdx] + B[threadIdx];
   }  };
```

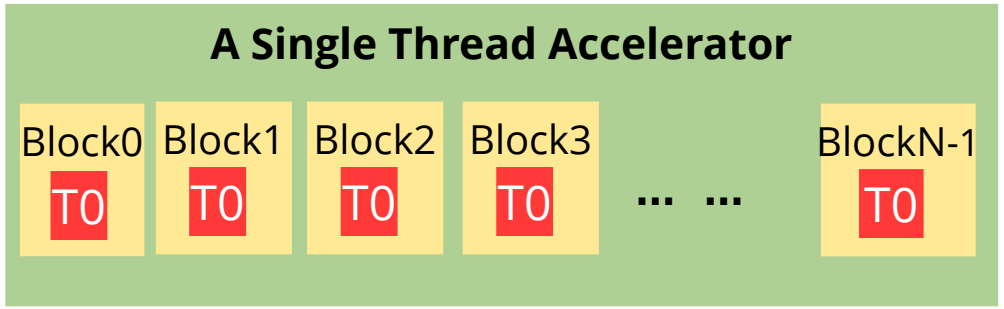
Alpaka Kernel Call

```
// since accelerator can be a CPU or GPU don't use constant workDiv
auto workDiv = alpaka::getValidWorkDiv(kernelCfg, devAcc,ptrA, ptrB, ptrC);
// Launch the vector addition kernel
VectorAddKernel vectorAddKernel;
alpaka::exec<Acc>(queue, workDiv, vectorAddKernel, bufA, bufB, bufC);
```

alpaka::getValidWorkDiv



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({M}, {4}, {1});



auto const **workDiv** = alpaka::WorkDivMembers<Dim, Idx>({N}, {1}, {1});

Alpaka Kernel

```
class VectorAddKernel
{ public:
   template<typename TAcc>
   ALPAKA_FN_ACC auto operator()( TAcc const& acc,   // the accelerator
      int * A, int*  B,  int*  C,  unsigned const& N) const -> void
   {  // Get the thread index, only first element of the vector needed since we are in 1D
   auto const threadIdx = alpaka::getIdx<alpaka::Grid, alpaka::Thread>(acc)[0];
   C[threadIdx] = A[threadIdx] + B[threadIdx];
   }  };
```

Alpaka Kernel Call

```
// since accelerator can be a CPU or GPU don't use constant workDiv
auto workDiv = alpaka::getValidWorkDiv(kernelCfg, devAcc,ptrA, ptrB, ptrC);
// Launch the vector addition kernel
VectorAddKernel vectorAddKernel;
alpaka::exec<Acc>(queue, workDiv, vectorAddKernel, bufA, bufB, bufC);
```

Valid does not mean "optimal"

Hands-on Session

Hands-On Session2:
Hello Index Example

Thank you for attending!

APPENDICES

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Appendix 1

2. Select the Accelerator, Platform and Device

- alpaka provides a number of pre-defined **Accelerators**.
  - `Acc`**`Gpu`**`CudaRt` for Nvidia GPUs
  - `Acc`**`Gpu`**`HipRt` for AMD GPUs
  - Acc**Gpu**SyclIntel for AMD, Intel and Nvidia GPUs
  - `Acc`**`Cpu`**`Omp2Blocks` based on OpenMP 2.x
  - `Acc`**`Cpu`**`TbbBlocks` based on TBB
  - `Acc`**`Cpu`**`Threads` based on `std::thread`
  - `Acc`**`Cpu`**`Sycl`
  - `Acc`**`Fpga`**`SyclIntel`

```
auto main() -> int
{
    using Dim = alpaka::DimInt<1u>; // Number of dimensions as a type, 1 for 1D index domain
    using Idx = std::size_t; // Index type of the threads and buffers
    using DataType = std::uint32_t; // Define the buffer element type

   // Define the accelerator: AccGpuCudaRt, AccGpuHipRt,
   // AccCpuThreads, AccCpuOmp2Threads, AccCpuOmp2Blocks, AccCpuTbbBlocks AccCpuSerial
    using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
    using DevAcc = alpaka::Dev<Acc>;

    // Select a device from platform of Acc
    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);
```

- **Device** instance represents a single physical device

Appendix 2

3. How to parallelise?

I- Get a valid work division from alpaka

Use **`getValidWorkDiv`** function which **devides the full grid-thread extent into blocks.**

- Inputs:
  - **Full grid-thread extent** (total number of threads needed) and **Elements per thread extent**
- The probable output:

`{Vec{alpaka::core::divCeil(numElements,1024)}, Vec{1024}, Vec{1} }`

II - Determine the workdivision manually

- WorkDivision data structure consists 3 vectors:
  - Grid block extent.
    - `Vec{alpaka::core::divCeil(numElements,1024)}` or `Vec{1,1,alpaka::core::divCeil(numElements,1024)}` depending on the number of dimensions.
  - Block thread extent: `Vec{1024}` or `Vec{1,1,1024}`
  - Elements per thread is `Vec{1}` or `Vec{1,1,1}`

```
using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
using DevAcc = alpaka::Dev<Acc>;

// Define the work division depending on the data
Idx const numElements(100000);
Idx const elementsPerThread(1u);
alpaka::Vec<Dim, Idx> const extent(numElements);

// Let alpaka calculate good block and grid sizes given our full problem extent
alpaka::WorkDivMembers<Dim, Idx> const workDiv = alpaka::getValidWorkDiv<Acc>(
    devAcc, // device
    extent, // {length, height, depth} of grid. For 1D only legth of the vector!
    elementsPerThread, false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

.....
// Instantiate the kernel function object
VectorAddKernel kernel;
alpaka::exec<Acc>( // Run the kernel execution task
    queue,
    workDiv,
    kernel,
    std::data(bufAccA),
```

```
Total amount of constant memory:                65536 bytes
Total amount of shared memory per block:        49152 bytes
Total number of registers available per block: 65536
Warp size:                                      32
Maximum number of threads per multiprocessor:   2048
Maximum number of threads per block:            1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Nvidia Tesla K20 deviceQuery

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Appendix 3

4. Allocate data vectors on host and device.

- **alpaka::Buf** is multi-dimensional dynamic (runtime sized) container.

  It contains

  - *memory*,
  - *size*,
  - the *device,* the memory is allocated in!
- **alpaka::allocBuf()** allocates memory to the given device.

```
using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
using DevAcc = alpaka::Dev<Acc>;

// Select a device from platform of Acc
auto const platform = alpaka::Platform<Acc>{};
auto const devAcc = alpaka::getDevByIdx(platform, 0);

// Define the work division depending on the data
Idx const numElements(100000);
Idx const elementsPerThread(1u);
alpaka::Vec<Dim, Idx> const extent(numElements);
.....

// Get the host device for allocating memory on the host.
auto const platformHost = alpaka::PlatformCpu{};
// Get the device directly from CPU platform not from the platform of Acc
auto const devHost = alpaka::getDevByIdx(platformHost, 0);

// Host device type is needed, still not known
using DevHost = alpaka::DevCpu;
// Allocate 3 host memory buffers
using BufHost = alpaka::Buf<DevHost, DataType, Dim, Idx>;
BufHost bufHostA(alpaka::allocBuf<DataType, Idx>(devHost, extent));
BufHost bufHostB(alpaka::allocBuf<DataType, Idx>(devHost, extent));
BufHost bufHostC(alpaka::allocBuf<DataType, Idx>(devHost, extent));

// Fill the host buffers
for(Idx i(0); i < numElements; ++i)
{
    bufHostA[i] = randomA; bufHostB[i] = randomB; bufHostC[i] = 0;
}

// Allocate 3 buffers on the accelerator
using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
BufAcc bufAccA(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
BufAcc bufAccB(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
BufAcc bufAccC(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
```

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Appendix 4

5.1 Create the Queue for memcpy and kernel task

- alpaka::Queue is “a queue of tasks”
- Queue is always FIFO, everything is sequential inside the queue.
- *and more*
  - *Different queues run in parallel for many devices*
  - *Used for synchronization*
  - *Accelerator back-ends can be mixed (used in interleaves) within a device queue.*
  - *...*

```
// Create a queue on the device, define the synchronization behaviour
alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

// Copy from Host to Acc
alpaka::memcpy(queue, bufAccA, bufHostA);
alpaka::memcpy(queue, bufAccB, bufHostB);
alpaka::memcpy(queue, bufAccC, bufHostC);

// Instantiate the kernel function object
VectorAddKernel kernel;
alpaka::exec<Acc>( // Run the kernel execution task
    queue,
    workDiv,
    kernel, alpaka::getPtrNative(bufAccA), alpaka::getPtrNative(bufAccB),
    alpaka::getPtrNative(bufAccC),
    numElements);
// Copy back the result
alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
```

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Appendix 5

5.2 Copy data vectors to the Device

- **alpaka::memcpy** copies the data from one buffer/view to another buffer or view.
- **alpaka::Buf** knows the device it belongs to.
- Alternatively **alpaka::View** is used to adapt already allocated memory.

  if we already have a C++ **std::vector** at host; we don't need to create an **alpaka::Buf** to copy it between different devices. Converting it to an **alpaka::View** is enough to copy it using alpaka::memcpy.

```
// Allocate 3 buffers on the accelerator
using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
BufAcc bufAccA(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
BufAcc bufAccB(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
BufAcc bufAccC(alpaka::allocBuf<DataType, Idx>(devAcc, extent));

// Define the synchronization behavior of a queue
// choose between Blocking and NonBlocking
// Create a queue on the device
alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

// Copy from Host to Acc
alpaka::memcpy(queue, bufAccA, bufHostA);
alpaka::memcpy(queue, bufAccB, bufHostB);
alpaka::memcpy(queue, bufAccC, bufHostC);
```

Appendix 6

- **Call alpaka::exec** function
- The result is stored in an **alpaka::Buf**

7. Copy result back

- Copy the result in device to the host

```
// Instantiate the kernel function object
VectorAddKernel kernel;

alpaka::exec<Acc>( // Run the kernel execution task
    queue,
    workDiv,
    kernel,
    alpaka::getPtrNative(bufAccA),
    alpaka::getPtrNative(bufAccB),
    alpaka::getPtrNative(bufAccC),
    numElements);
// Copy back the result
alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
```

alpaka

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Appendix 7

Use ***mdspan*** which is a **multi-dimensional** and **non-owning view**.

std::mdspan standard

```
   //  std::mdspan
std::vector v{1,2,3,4,5,6,7,8,9,10,11,12};
  // View data as 2D
auto ms2 = std::mdspan(v.data(), 2, 6);
  // View the same data as 3D
auto ms3 = std::mdspan(v.data(), 2, 3, 2);

  // Write data into v using 2D view
for(std::size_t i = 0; i!=ms2.extent(0); i++)
 for(std::size_t j = 0; j!=ms2.extent(1); j++)
          ms2[i, j] = i * 1000 + j;
  // Read data using 3D view
```

alpaka::experimental::mdspan

```
// Define the 2D extents (dimensions)
Vec const extentA(M, K);
// Allocate host memory
auto bufHostA = alpaka::allocBuf<DataType, Idx>(devHost, extentA);

// Create mdspan view for bufHostA and bufHostB using alpaka::experi
// to fill the host buffers
auto mdHostA = alpaka::experimental::getMdSpan(bufHostA);

auto const numColumns = mdHostA.extent(1);
for(Idx i = 0; i < mdHostA.extent(0); ++i)
{
    for(Idx j = 0; j < mdHostA.extent(1); ++j)
    {
        // fill with some data
        mdHostA(i, j) = randomValue;
    }
}
```

Create an mdspan view

Easy set

Appendix 8

Pass alpaka ***mdspan*** to the +heat+ kernel

```
Vec const extentA(M, K);
// Allocate device memory
auto bufDevA = alpaka::allocBuf<DataType, Idx>(devAcc, extentA);

// Copy data to device, use directly host buffers (not mdspans)
alpaka::memcpy(queue, bufDevA, bufHostA);
alpaka::wait(queue);

// Create mdspan views for device buffers using alpaka::experimenta
auto mdDevA = alpaka::experimental::getMdSpan(bufDevA);

MatrixAddKernel kernel;
// ....
// Execute the kernel
alpaka::exec<Acc>(queue, workDiv, kernel, mdDevA, mdDevB, mdDevC);
```

Create an mdspan view

Pass to kernel

Use alpaka ***mdspan*** in the kernel

```
struct MatrixAddKernel
{
    template<typename TAcc, typename TMdSpan>
    ALPAKA_FN_ACC void operator()(TAcc const& acc, TMdSpan A, TMdSpan B, TMdSpan C) const
    {
        // compile time checks
        static_assert(is_mdspan<TMdSpan>, "The type TMdSpan should be an std mdspan");
        static_assert(TMdSpan::rank() == 2);

        auto const i = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0];
        auto const j = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[1];

        if(i < C.extent(0) && j < C.extent(1))
        {
            C(i, j) = A(i, j) * B(i, j);
        }
    }
};
```

set/get values

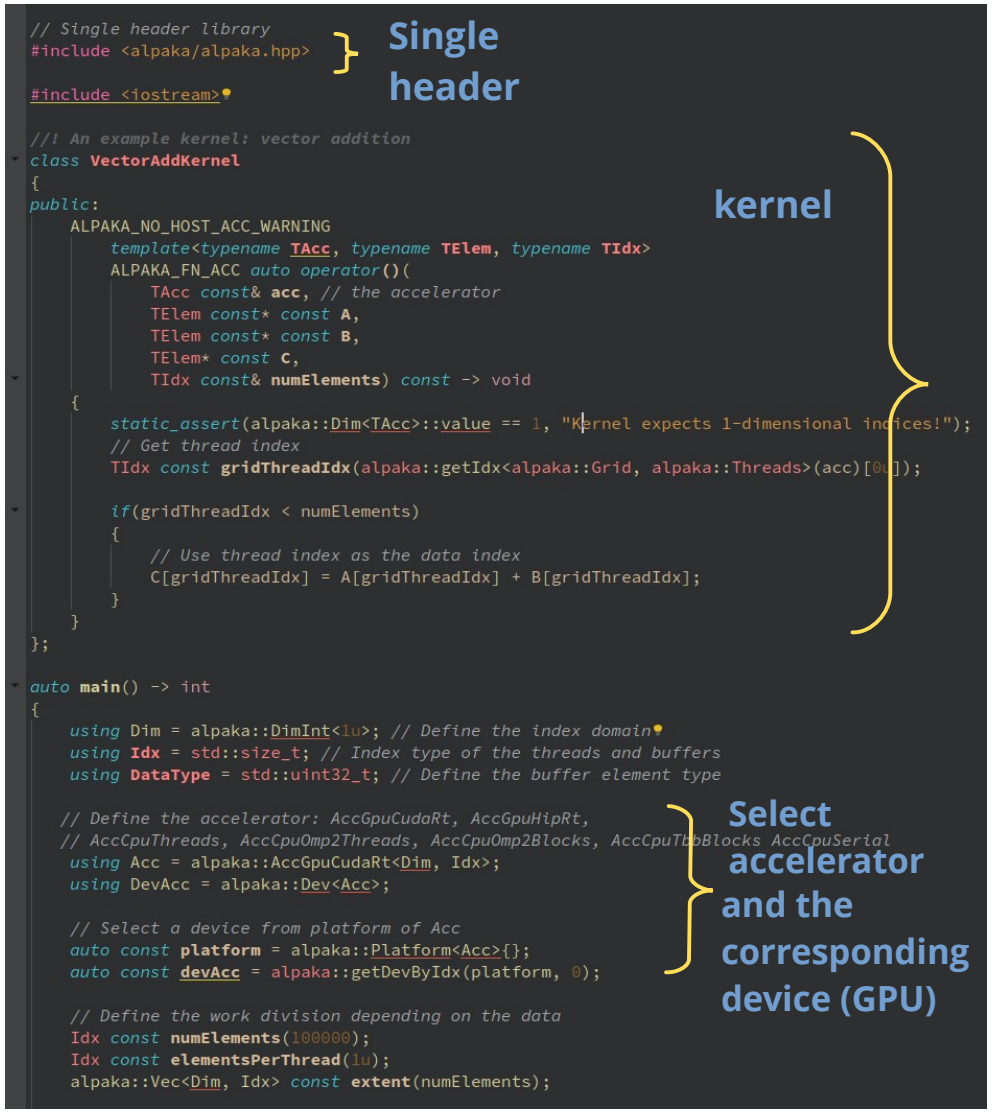
Parallel vector addition code

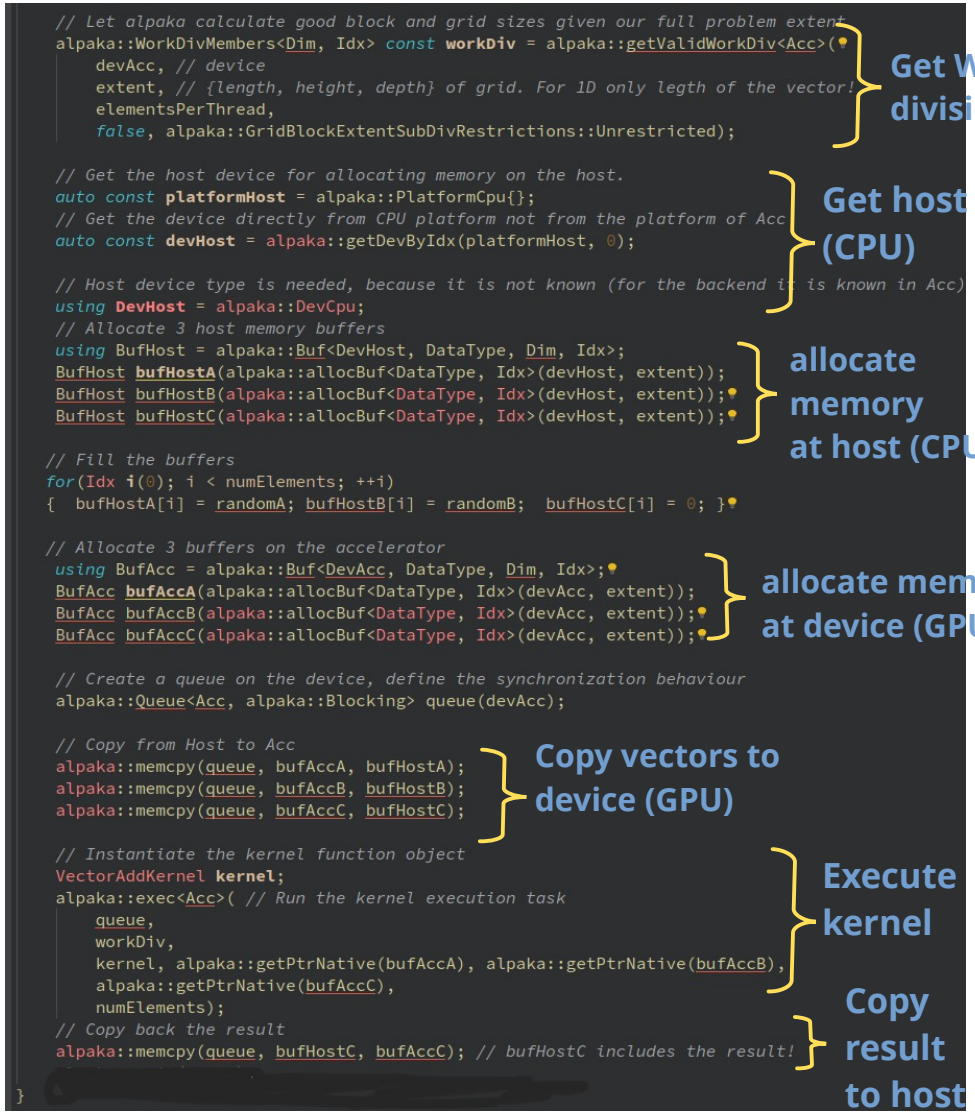
Appendix 9

```cpp
// Single header library
#include <alpaka/alpaka.hpp>

#include <iostream>

//! An example kernel: vector addition
class VectorAddKernel
{
public:
    ALPAKA_NO_HOST_ACC_WARNING
        template<typename TAcc, typename TElem, typename TIdx>
        ALPAKA_FN_ACC auto operator()(
            TAcc const& acc, // the accelerator
            TElem const* const A,
            TElem const* const B,
            TElem* const C,
            TIdx const& numElements) const -> void
    {
        static_assert(alpaka::Dim<TAcc>::value == 1, "Kernel expects 1-dimensional indices!");
        // Get thread index
        TIdx const gridThreadIdx(alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u]);

        if(gridThreadIdx < numElements)
        {
            // Use thread index as the data index
            C[gridThreadIdx] = A[gridThreadIdx] + B[gridThreadIdx];
        }
    }
};

auto main() -> int
{
    using Dim = alpaka::DimInt<1u>; // Define the index domain
    using Idx = std::size_t; // Index type of the threads and buffers
    using DataType = std::uint32_t; // Define the buffer element type

   // Define the accelerator: AccGpuCudaRt, AccGpuHipRt,
   // AccCpuThreads, AccCpuOmp2Threads, AccCpuOmp2Blocks, AccCpuTbbBlocks AccCpuSerial
    using Acc = alpaka::AccGpuCudaRt<Dim, Idx>;
    using DevAcc = alpaka::Dev<Acc>;

    // Select a device from platform of Acc
    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);

    // Define the work division depending on the data
    Idx const numElements(100000);
    Idx const elementsPerThread(1u);
    alpaka::Vec<Dim, Idx> const extent(numElements);
```

```cpp
    // Let alpaka calculate good block and grid sizes given our full problem extent
    alpaka::WorkDivMembers<Dim, Idx> const workDiv = alpaka::getValidWorkDiv<Acc>(
        devAcc, // device
        extent, // {length, height, depth} of grid. For 1D only legth of the vector!
        elementsPerThread,
        false, alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);

    // Get the host device for allocating memory on the host.
    auto const platformHost = alpaka::PlatformCpu{};
    // Get the device directly from CPU platform not from the platform of Acc
    auto const devHost = alpaka::getDevByIdx(platformHost, 0);

    // Host device type is needed, because it is not known (for the backend it is known in Acc)
    using DevHost = alpaka::DevCpu;
    // Allocate 3 host memory buffers
    using BufHost = alpaka::Buf<DevHost, DataType, Dim, Idx>;
    BufHost bufHostA(alpaka::allocBuf<DataType, Idx>(devHost, extent));
    BufHost bufHostB(alpaka::allocBuf<DataType, Idx>(devHost, extent));
    BufHost bufHostC(alpaka::allocBuf<DataType, Idx>(devHost, extent));

   // Fill the buffers
   for(Idx i(0); i < numElements; ++i)
   {  bufHostA[i] = randomA; bufHostB[i] = randomB;  bufHostC[i] = 0; }

   // Allocate 3 buffers on the accelerator
    using BufAcc = alpaka::Buf<DevAcc, DataType, Dim, Idx>;
    BufAcc bufAccA(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
    BufAcc bufAccB(alpaka::allocBuf<DataType, Idx>(devAcc, extent));
    BufAcc bufAccC(alpaka::allocBuf<DataType, Idx>(devAcc, extent));

    // Create a queue on the device, define the synchronization behaviour
    alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

    // Copy from Host to Acc
    alpaka::memcpy(queue, bufAccA, bufHostA);
    alpaka::memcpy(queue, bufAccB, bufHostB);
    alpaka::memcpy(queue, bufAccC, bufHostC);

    // Instantiate the kernel function object
    VectorAddKernel kernel;
    alpaka::exec<Acc>( // Run the kernel execution task
        queue,
        workDiv,
        kernel, alpaka::getPtrNative(bufAccA), alpaka::getPtrNative(bufAccB),
        alpaka::getPtrNative(bufAccC),
        numElements);
    // Copy back the result
    alpaka::memcpy(queue, bufHostC, bufAccC); // bufHostC includes the result!
}
```

Parallel vector addition code





Appendix 10

Programing Tips

- If you want to pass multi-dimensional data to kernel, use mdspan (enable it via cmake option)

  (If you don’t use mdspan; you will need to take care of alignment/pitch values. Pass the pointer, extents and the pitch.)
- To incease perfomance; using **shared memory and constant memory** of GPUs are among alpaka features.
- A kernel can be run directly by **exec** function or can be **enqueue**d as a task.
- Vendor specific profiling and debugging tools (e.g. **nsys, rocprof** …) can be used on compiled alpaka code.
- If you debug GPU code try to compile your code for CPU; and use CPU debugger tools

  (Change accelerator type to CPU accelerators, then debug using **gdb** and similar tools.)
- Inside alpaka Kernel, you can use printf; but you should **not use** std::cout for GPU backends.
- For unused number of dimensions in workdiv; use 1, for that dimension.

  auto blockThreadExtent = **alpaka::Vec**<TDim3D,Idx>{1u,1u,128u};

Appendix-11 Summary of alpaka Structures

- **Accelerator** provides abstract view of all capable physical devices
- **Device** represents a single physical device
- **Queue** Enables synchronisation of tasks on different queues, enables communication between the host and a single Device, e
- **Platform** is a union of Accelerator, Device and Kernel
- **Task** is a device-side operation (e.g kernel, memory operation)
- Others: **Event, Buffer** (runtime sized contiguous container), **Vector** (static array)

Appendix-12: Programming Heterogeneous Systems-I

How to use multiple backends in parallel?

- Acquire at least one Device per Accelerator
- Create one Queue per Device

```
// Define Accelerators
using AccCpu = alpaka::AccCpuOmp2Blocks<Dim, Idx>;
using AccGpu = alpaka::AccGpuCudaRt<Dim, Idx>;

// Acquire at least one Device per Accelerator
auto devCpu = alpaka::getDevByIdx<AccCpu>(0u);
auto devGpu = alpaka::getDevByIdx<AccGpu>(0u);

// Create one queue per device

using QueueProperty = alpaka::NonBlocking;
using QueueCpu = alpaka::Queue<AccCpu, QueueProperty>;
using QueueGpu = alpaka::Queue<AccGpu, QueueProperty>;
auto queueCpu = QueueCpu{devCpu};
auto queueGpu1 = QueueGpu{devGpu};
auto queueGpu2 = QueueGpu{devGpu};

// Run tasks in parallel
alpaka::enqueue(queueCpu, taskCpu);
alpaka::enqueue(QueueGpu1, taskGpu1);
alpaka::enqueue(QueueGpu2, taskGpu2);
// Make sure all non-blocking queue tasks finished
// before the main thread ends
alpaka::wait(queueCpu);
alpaka::wait(QueueGpu1);
alpaka::wait(QueueGpu2);
```

Appendix-13
Programming Heterogeneous Systems-II

Communication by Buffers

- Buffers are defined and created per Device
- Buffers can be copied between different Devices
- **Notice**: CPU to GPU copies (and vice versa) require GPU queue

```
// Allocate buffers
auto bufCpu = alpaka::allocBuf<float, Idx>(devCpu, extent);
auto bufGpu = alpaka::allocBuf<float, Idx>(devGpu, extent);

/* Initialization … */

// Copy buffer from CPU to GPU - destination comes first
alpaka::memcopy(gpuQueue, bufGpu, bufCpu, extent);

// Execute GPU kernel
alpaka::enqueue(gpuQueue, someKernelTask);

// Copy results back to CPU and wait for completion
alpaka::memcopy(gpuQueue, bufCpu, bufGpu, extent);

// Wait for GPU, then execute CPU kernel
alpaka::wait(cpuQueue, gpuQueue);
alpaka::enqueue(cpuQueue, anotherKernelTask);
```

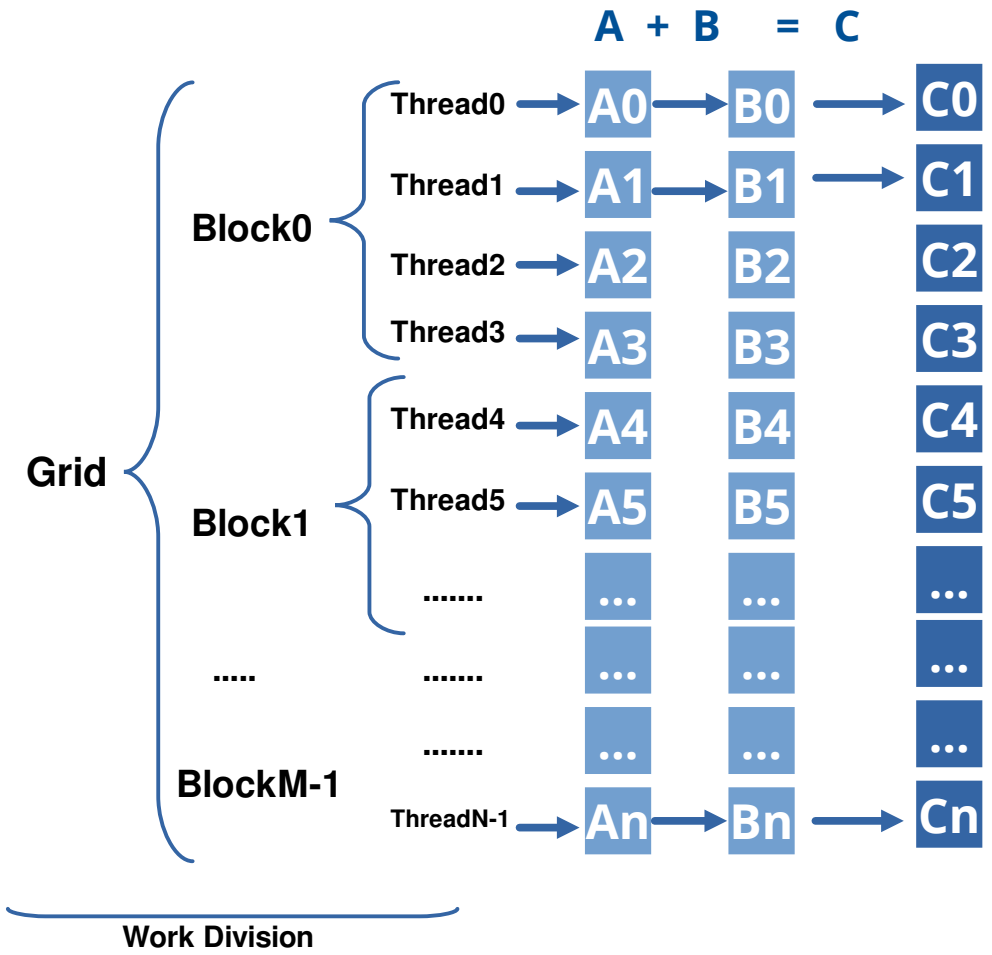
Appendix-14 Set the workdivision manually

- WorkDivision data structure consists 3 vectors:
  - Grid block extent.

    `Vec{M}` or `Vec{1,1,M}` depending on the number of dimensions.
  - Block thread extent.

    `Vec{4}` or `Vec{1,1,4}`
  - Elements per thread
- Setting work-div manually

```
using Dim1D = alpaka::DimInt<1>;//Set number of dims to 1
using Vec1D = alpaka::Vec<Dim1D, Idx>;//Define alias
auto workDiv1D = alpaka::WorkDivMembers(Vec1D{M},  Vec1D{4u},  Vec1D{1u});
// alternatively
using Dim3D = alpaka::DimInt<3>;//Set number of dims to 3
using Vec3D = alpaka::Vec<Dim3D, Idx>; //Define alias
auto workDiv3D = alpaka::WorkDivMembers(Vec3D{1,1,M},  Vec3D{1,1,4u},  Vec3D{1,1,1u});
```



Appendix-15 Tasks and Events

- Device-side related operations (kernels, memory operations) can be wrapped in tasks.
- Tasks are executed by **enqueue()** function.
- Tasks on the same queue are executed in order (FIFO principle)

```
alpaka::enqueue(queueA, task1);
alpaka::enqueue(queueA, task2); // task2 starts after task1 has finished, even queueA is non-blocking
```

- Order of tasks in different queues is unspecified
  -
```
alpaka::enqueue(queueA, task1);
alpaka::enqueue(queueB, task2); // task2 starts before, after or in parallel to task1
```

- For easier synchronization, alpaka Events can be inserted before, after or between Tasks:

```
auto myEvent = alpaka::Event<alpaka::Queue>(myDev);
alpaka::enqueue(queueA, myEvent);
alpaka::wait(queueB, myEvent); // queueB will only resume after queueA reached myEvent
```

```
// Create a queue on the device
QueueAcc queue(devAcc);

....
// Instantiate the kernel function object
VectorAddKernel kernel;

// Create the kernel execution task.
auto const taskKernel = alpaka::createTaskKernel<Acc>(
    workDiv,
    kernel,
    alpaka::getPtrNative(bufAccA),
    alpaka::getPtrNative(bufAccB),
    alpaka::getPtrNative(bufAccC),
    numElements);

alpaka::enqueue(queue, taskKernel);
alpaka::wait(queue); // wait in case we are using an asynchronous queue
```

Appendix-16 Accelerator Details

- Accelerator chosen by the programmer and **hides hardware specifics** behind alpaka's abstract API
  ```
  using Acc = acc::AccGpuCudaRt<Dim, Idx>;
  ```
- **Inside Kernel:** contains thread state, provides access to alpaka's device-side API
  - **The Accelerator provides the means to access to the indices**
    ```
    // get thread index on the grid
    auto gridThreadIdx = alpaka::getIdx<Grid, Threads>(acc);
    // get block index on the grid
    auto gridBlockIdx = alpaka::getIdx<Grid, Blocks>(acc);
    ```
  - **The Accelerator gives access to alpaka's shared memory** (for threads inside the same block)
    ```
    // allocate a variable in block shared static memory
    auto & mySharedVar = block::shared::st::allocVar<int, __COUNTER__>(acc);

    // get pointer to the block shared dynamic memory
    float * mySharedBuffer = block::shared::dyn::getMem<float>(acc);
    ```
  - **It also enables synchronization on the block level**
    ```
    // synchronize all threads within the block
    block::sync::syncBlockThreads(acc);
    ```
  - **Internally, the accelerator maps all device-side functions to their native counterparts**
    - Device-side functions require the accelerator as first argument:

      `math::sqrt(acc, /* … */);` `time::clock(acc);`

      `atomic::atomicOp<atomic::op::Or>(acc, /* … */, hierarchy::Grids);` (Atomics)
- **On Host:** Meta-parameter for choosing correct physical device and dependent types

APPENDIX-17 Device information and device management

- Each alpaka **Device** represents a single physical device;
- Contains device information:
  - `auto const name = alpaka::getName(myDev);  // Back-end-defined device name`
  - `auto const bytes = alpaka::getMemBytes(myDev);  // Size of device memory`
  - `auto const free = alpaka::getFreeMemBytes(myDev);  // Size of available device memory`
- Provides the means for device management:
  - `alpaka::reset(myDev);  // Reset GPU device state`
- Encapsulates back-end device:
  - `auto nativeDevice = alpaka::getDev(myDev);  // nativeDevice is not portable!`

APPENDIX-18 Queue operations

- Queues execute Tasks (see next slide):
  - `alpaka::enqueue(myQueue, taskRunKernel);`
- Check for completion:
  - `bool done = alpaka::empty(myQueue);`
- Wait for completion, Events (see next slide), or other Queues:
  - `alpaka::wait(myQueue);` `// blocks caller until all operations have completed`
  - `alpaka::wait(myQueue, myEvent);` `// blocks myQueue until myEvent has been reached`
  - `alpaka::wait(myQueue, otherQueue);` `// blocks myQueue until otherQueue's ops have completed`

Appendix-19 Changing the target platform by changing accelerator

```
using namespace alpaka;

using Dim = dim::DimInt<1u>;
using Idx = std::size_t;

/*** BEFORE ***/
using Acc = alpaka::AccCpuOmp2Blocks<Dim, Idx>;


/*** AFTER ***/
using Acc = alpaka::AccGpuHipRt<Dim, Idx>;


/* No change required - dependent types and variables are automatically changed
*/
auto myDev = alpaka::getDevByIdx<Acc>(0u);


using Queue = alpaka::Queue<Acc, queue::NonBlocking>;
auto myQueue = Queue{myDev};
```