

## Alpaka Hackathon Section-II: Heat Equation Solution

PLASMA-PEPSC Workshop on Alpaka and OPENPMD

October 22, 2024

# Workshop Schedule

## Section - I

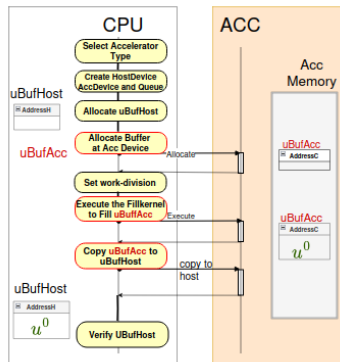
- 1 Introduction: What is alpaka, where it is used?
- 2 **Hands on 1:** Installing alpaka and running an example (LUMI)
- 3 Parallel programming concepts and portable parallelization by alpaka
  - Grid Structure and WorkDivision
  - Data Parallelism
  - Indexing
- 4 **Hands on 2:** HelloIndex kernel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

## Steps of Filling a Buffer in Parallel

- 1 Select the accelerator
- 2 Create host-device, acc-device and the queue
- 3 Allocate host and device memory using Buf and Vec data structures
- 4 Decide how to parallelize: Set the work-division
- 5 Create the kernel instance for filling the buffer
- 6 Execute the kernel
- 7 Copy the result from Acc (e.g GPU) back to the host buffer.



## Allocate memory at Host and at Device

### Define number of dim and index type

```
1 using Dim = alpaka::DimInt<2u>; // Number of dim: 2 as a type
2 using Idx = std::size_t; // Index type of the threads and buffers
```

### Define extents for buffers

```
1 // alpaka::Vec is a static array similar to std::array.
2 // Dim is a compile-time constant, which is 2.
3 constexpr alpaka::Vec<Dim, Idx> numNodes{64, 64};
4 constexpr alpaka::Vec<Dim, Idx> haloSize{2, 2};
5 constexpr alpaka::Vec<Dim, Idx> extent = numNodes + haloSize;
```

### Allocate memories at host and accelerator

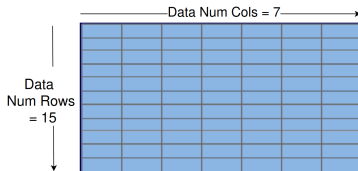
```
1 // Allocate memory for host-buffer. Creates alpaka::Buf.
2 auto uBufHost = alpaka::allocBuf<double, Idx>(devHost, extent);
3
4 // Allocate memory for accelerator buffer
5 auto uBufAcc = alpaka::allocBuf<double, Idx>(devAcc, extent);
```

### Pass acc pointer uBufAcc to kernel and execute kernel

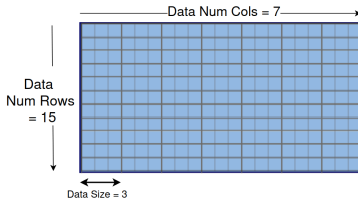
```
1 alpaka::exec<Acc>(queue, workDiv, initBufKernel, uBufAcc.data(), ...);
```

Aim: Finding the adress of a data and fill it with a specific value

Lets assume that the 2D data is a 15 rows x 7 columns matrix



Assume that data-size is 3

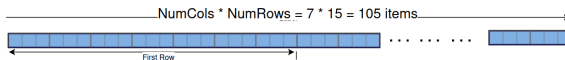


## Structure of allocated data at the memory

To fill the acc data buffer at the kernel the allocated memory structure should be known.

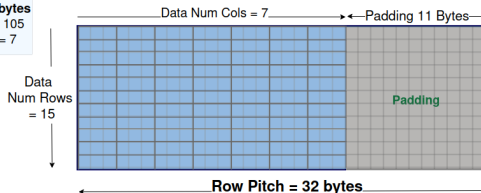
### 1D Data

Data item size = 3 bytes  
Number of Items = 105



### 2D Data

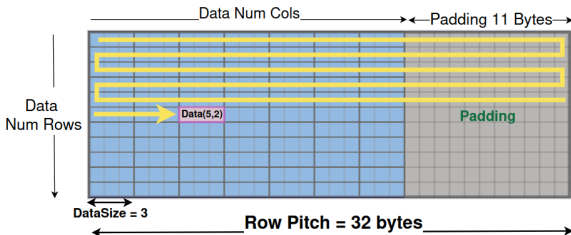
Data item size = 3 bytes  
Number of Items = 105  
Rows = 15, Cols = 7



# How to access data given the pointer and the pitch?

How to access data at index (5,2) given the pointer ptr and pitch?

pitch = {32bytes, 3 bytes} as {row-pitch, datasize}



```
dataPosition = (char *)ptr + 5*pitch[0] + 2*pitch[1]
dataPosition = (char *)ptr + 5*32 + 2*3
dataPtr = (T *)dataPosition
// shorter alternative if pitch[0] is divisible by pitch[1];
// which is the usual case
dataPtr = ptr + (5*pitch[0] + 2*pitch[1])/sizeof(T)
```

## Passing multi dimensional buffer to the kernel

### ■ Pass 3 variables for a buffer: pointer, "row-pitch", and datasize

If a pointer of a 2D buffer is passed to the kernel as a pointer; 2 additional values **pitch** and item **data-size** should also be passed.

```
1 // Signature of function operator of the Kernel
2 template<typename TAcc, typename TDim, typename TIdx>
3 ALPAKA_FN_ACC auto operator() (
4     TAcc const& acc,
5     double* const bufData,
6     // 2 variables row-pitch and data-type size
7     alpaka::Vec<TDim, TIdx> const pitch,
8     double dx,
9     double dy) const -> void
```

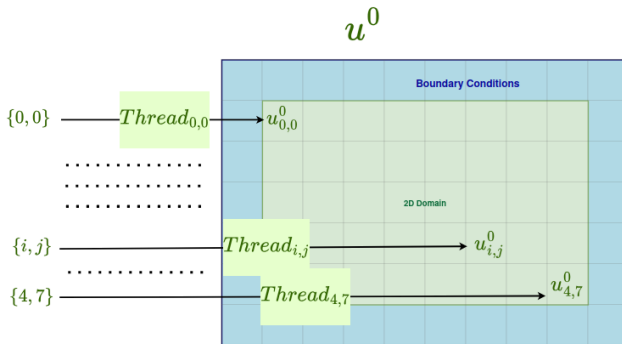
### ■ Simple Alternative: Pass an alpaka mdspan object

```
1 template<typename TAcc, typename TDim, typename TIdx, typename TMdSpan>
2 ALPAKA_FN_ACC auto operator() (
3     TAcc const& acc,
4     TMdSpan uAccMdSpan
5     ...) const -> void
```



## The Kernel to Initialize Heat Values

Calculate and set initial heat values, the  $u^0$  matrix, by running a grid of threads.



## Hands on 3: The Kernel to Initialize Heat Values

**The InitializeBufferKernel fills the given buffer at the accelerator device (e.g GPU)**  
Prepare kernel to set initial heat values

- **Thread Index:** Find thread index in the kernel to be used as index to set 2D buffer.
- **Initial Condition at the point:** Find analytically the heat value at the point which has coordinates equal to the 2D thread index.
- **Memory Address in Buffer:** Calculate the corresponding memory address in buffer using thread index. Take into account the row-pitch and data-size
- **Set Value at the Address:** Set the data at the memory position to the calculated initial condition.

```

1  template<typename TAcc, typename TDim, typename TIdx>
2  ALPAKA_FN_ACC auto operator()(
3      TAcc const& acc, double* const bufData,
4      alpaka::Vec<TDim, TIdx> const pitch, double dx, double dy) const -> void {
5      // Get 2D thread index using alpaka index function
6      .....
7      // Calculate analytical solution at point
8      auto heatAtPointValue = analyticalSolution(acc, gridThreadId[1] * dx, gridThreadId[0] * dy,
9          0.0);
10     // Calculate data position in buffer, from thread index and pitches
11     auto ptr = getElementPtr(bufData, gridThreadId, pitch);
12     // Set the value using the address
13     *ptr = heatPointValue;
14 } // function operator

```

## Hands-on Session3: Filling an accelerator buffer paralelly

# The Heat Equation

- The Heat Diffusion over time:

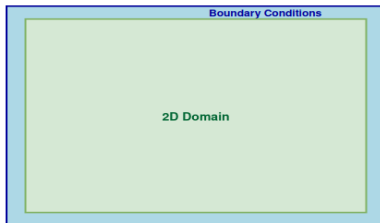
$$\frac{\partial u(x, y, t)}{\partial t} = \alpha \left( \frac{\partial^2 u(x, y, t)}{\partial x^2} + \frac{\partial^2 u(x, y, t)}{\partial y^2} \right)$$

Time and Spatial Derivative Approximations:

$$\left. \frac{\partial u(x, y, t)}{\partial t} \right|_{t=t^n} \approx \frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} \quad \left. \frac{\partial^2 u(x, y, t)}{\partial x^2} \right|_{x=x_i, y=y_j} \approx \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2}$$

- Resulting difference equation:

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha \Delta t \left( \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} \right)$$



## The Heat Equation- Cont.

- The difference equation:

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha \Delta t \left( \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} \right)$$

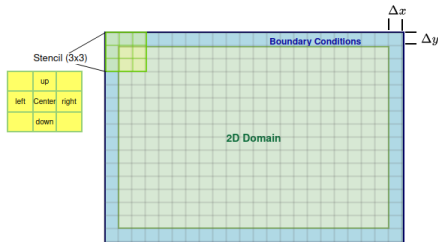
- Substitute:  $\alpha = 1$ ,  $r_X = \frac{\Delta t}{\Delta x^2}$ ,  $r_Y = \frac{\Delta t}{\Delta y^2}$ . Then  $u_{i,j}^{n+1}$  is:

$$u_{i,j}^{n+1} = u_{i,j}^n + r_X (u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) + r_Y (u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n)$$

By regrouping the terms related to  $u_{i,j}^n$ , the equation can be rewritten as:

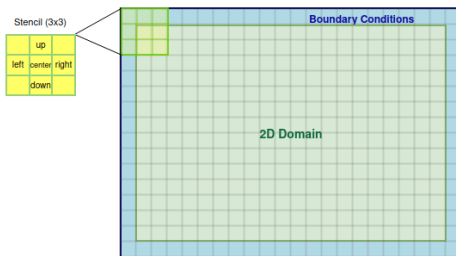
$$u_{i,j}^{n+1} = u_{i,j}^n (1 - 2r_X - 2r_Y) + r_X (u_{i+1,j}^n + u_{i-1,j}^n) + r_Y (u_{i,j+1}^n + u_{i,j-1}^n)$$

$$S = \begin{pmatrix} 0 & r_Y & 0 \\ r_X & 1 - 2r_X - 2r_Y & r_X \\ 0 & r_Y & 0 \end{pmatrix}$$



# Parallel Heat Equation Solution

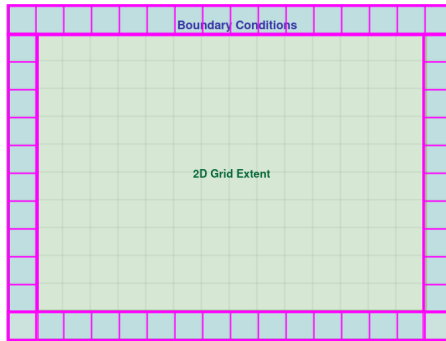
- **Data Parallelism:** Each point on the grid can be updated independently based on its neighbors, enabling parallel computation.
- **Independency:** Next values of heat at a point does not depend on previous heat values.
- **Stencil Operations:** Stencil is a core computational pattern in PDE solvers. Updates a grid point in time using its immediate neighbors (left, right, up, down) according to the difference equation.



- **Halo Region for BC:** A layer of grid cells added to the core-domain surrounding the problem domain for Boundary Conditions.

# Halo Region

2D Grid Extent + Halo

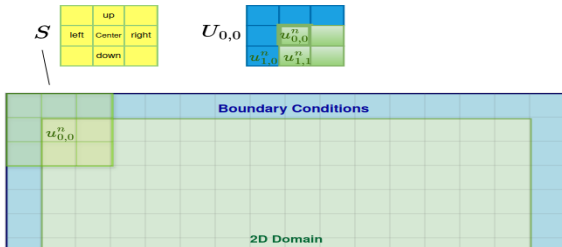


# Calculation of $u_{i,j}^{n+1}$ from $u_{i,j}^n$

- Kernel execution by thread  $i,j$  calculates  $u_{i,j}^{n+1}$  using  $u_{i,j}^n$
- Hence, Each heat point is separately calculated by a thread using **Frobenious Inner Product** (FIP) of  $S$  and matrix  $U_{i,j}$ :

$$u_{i,j}^{n+1} = \langle S, U_{i,j}^n \rangle_F = \sum_{m=1}^M \sum_{k=1}^K s_{m,k} u_{m,k}$$

- $S$  and  $U_{0,0}$  is used by a thread to calculate  $u_{0,0}^{n+1}$  using FIP





- Another thread calculates  $u_{0,1}^{n+1}$  using  $S$  and  $U_{0,1}^n$

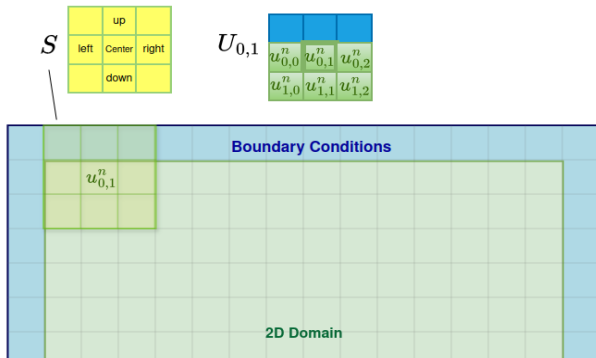
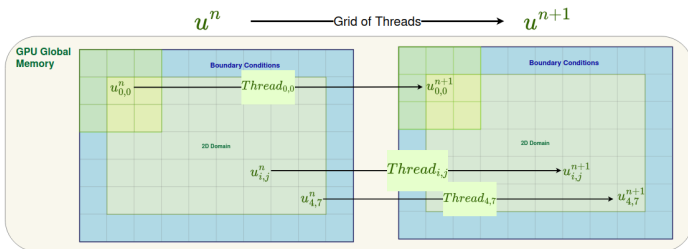


Figure: Another thread calculates  $u_{0,1}^{n+1}$

A grid of threads is used to find next 2D heat values



- Stencil kernel will update only core nodes not the border
- The workdiv for stencil-kernel can be calculated by setting grid-thread extent to nodes domain
- The workdiv for the borders-kernel would need larger extents, but will only work on borders of this extended domain.

# Complete Heat Equation Solution

## ■ Initialization:

- Define the "host device" and "accelerator device". The "Host" and "Device" in short.
- Set initial conditions and boundary conditions.
- Allocate data buffers to host and device.
- Copy data from host to device buffer to pass to the kernel.
- Define parallelisation strategy (determine block size).

## ■ Simulation Loop Over Time:

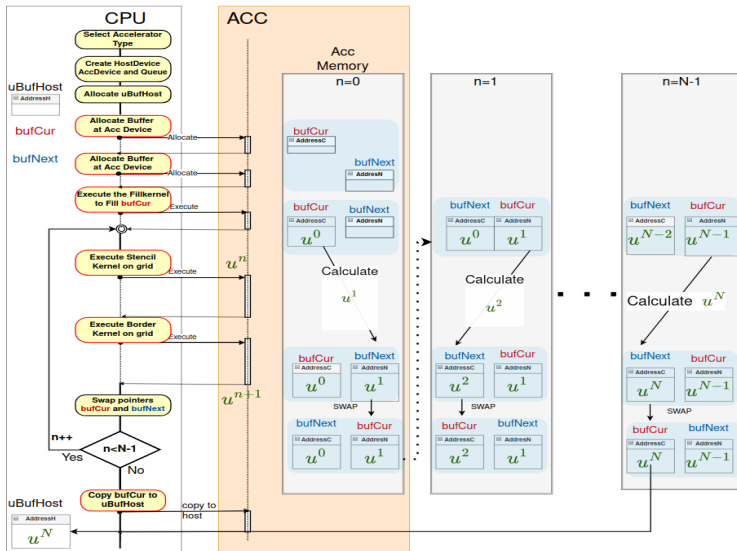
- **Step 1:** Execute `StencilKernel` to calculate next heat values for 2D domain.
- **Step 2:** Apply boundary conditions using `BoundaryKernel`.
- **Step 3:** Swap buffers for the next iteration so that calculated  $u_{i,j}^{n+1}$  becomes the  $u_{i,j}^n$  for the next step.

## ■ Copy result from Acc to Host:

- Use `alpaka::memcpy` to copy latest 2D heat buffer  $u^n$  to host back.

## ■ Validation

# Complete Heat Equation solution



## Hands On 4: The Stencil Kernel

What kind of parallelization needed to calculate  $u_{i,j}^{n+1}$  using  $u_{i,j}^n$

- StencilKernel needs a Work-division to work on domain of all nodes (without halo)
- Boundary kernel needs a Workdivision which covers nodes + halo region
- Boundary kernel will only use threads corresponding to halo region

### Coding StencilKernel

- **Input:** Check the size of the input buffer, it should include halo region as well
- **Thread Index:** Find thread index in the kernel. This index will be used as the center of 3x3 stencil.
- **Memory Address in Buffer:** Calculate the corresponding memory address in buffer using thread index. Take into account pitch and data-size
- **Find Stencil matrix** Use dx, dy, dt to calculate the stencil matrix
- **Calculate new heat value:** Calculate  $u_{i,j}^{n+1}$  using **Frobenious Inner Product** of 3x3 matrices
- **Set Value at the Address:** Set the data at the memory address.

```

1  struct StencilKernel
2  {
3      template<typename TAcc, typename TDim, typename TIdx>
4      ALPAKA_FN_ACC auto operator() (
5          TAcc const& acc,
6          double const* uCurrBuf, double* const uNextBuf,
7          alpaka::Vec<TDim, TIdx> const chunkSize,
8          alpaka::Vec<TDim, TIdx> const pitchCurr, alpaka::Vec<TDim, TIdx> const pitchNext,
9          double const dx, double const dy, double const dt) const -> void
10     {
11         ...
12     }
13 };

```

# Hands-on Session4: Stencil Kernel to Calculate $u^{n+1}$

# Setting up the stage to run kernels

- 1 Selecting the accelerator and host devices
- 2 Allocating and setting host and accelerator device memory
- 3 Alpaka Vector, Buffer and View
- 4 Passing data to the accelerator
- 5 WorkDiv
- 6 Define Queue

## Accelerator, Device and Host

### Define number of dim and index type

```
1 using Dim = alpaka::DimInt<2u>; // Number of dim: 2 as a type
2 using Idx = std::size_t; // Index type of the threads and buffers
```

### Define the accelerator

```
1 // AccGpuCudaRt, AccGpuHipRt, AccCpuThreads, AccCpuSerial,
2 // AccCpuOmp2Threads, AccCpuOmp2Blocks, AccCpuTbbBlocks
3 using Acc = alpaka::AccGpuHipRt<Dim, Idx>;
4 using DevAcc = alpaka::Dev<Acc>;
```

### Select a device from platform of Acc

```
1 auto const platform = alpaka::Platform<Acc>{};
2 auto const devAcc = alpaka::getDevByIdx(platform, 0);
```

### Select a host and hostype to allocate memory for data

```
1 // Get the host device for allocating memory on the host.
2 auto const platformHost = alpaka::PlatformCpu{};
3 auto const devHost = alpaka::getDevByIdx(platformHost, 0);
4 // Host device type is needed, still not known
5 using DevHost = alpaka::DevCpu;
```



## What is Accelerator

**Accelerator** hides hardware specifics behind alpaka's abstract API

- **On Host:** Accelerator is a type. A Meta-parameter for choosing correct physical device and dependent types

```
1 using Acc = acc::AccGpuHipRt<Dim, Idx>;
```

- **Inside Kernel:** Accelerator is a variable. Contains thread state, provides access to alpaka's device-side API

- The Accelerator provides the means to access to the indices

```
1 // get thread index on the grid
2 auto gridThreadId = alpaka::getIdx<Grid, Threads>(acc);
```

- The Accelerator gives access to alpaka's shared memory (for threads inside the same block)

```
1 // allocate a variable in block shared static memory
2 auto& sdata = alpaka::declareSharedVar<double[T_SharedMemSize1D], __COUNTER__>(acc);
```

- Enables synchronization on the block level

```
1 // synchronize all threads within the block
2 alpaka::syncBlockThreads(acc);
```

- Maps all device-side functions to their native counterparts

```
1 alpaka::sqrt(acc, val); // or atomics, create distribution
```

## What is alpaka Buffer, Vector and View

- **alpaka::Buf** is multi-dimensional dynamic (runtime sized) container.
  - Contains **memory address**, **extent**, **datatype** and the **device** that memory belongs to!
  - Device to device copy is easy
  - Supports `[]` operator but not `[][]`.

```
1 // Allocate buffers
2 auto bufCpu = alpaka::allocBuf<float, Idx>(devCpu, extent);
3 auto bufGpu = alpaka::allocBuf<float, Idx>(devGpu, extent);
4 ....
5 // Copy buffer from CPU to GPU. Notice: destination comes first!
6 alpaka::memcpy(gpuQueue, bufGpu, bufCpu);
7 // cuda way: cudaMemcpy(b_d, b_host, sizeof(float)*N, cudaMemcpyHostToDevice)
```

- **alpaka::Vec** is a static 1D array.

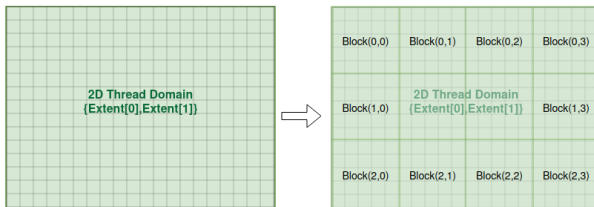
```
1 alpaka::Vec<SizeOfArrayAsType, DataT> myVec;
```

- **alpaka::View** is a non-owning view to an already allocated memory, so that it can be used in `alpaka::memcpy`

## Determining WorkDiv using getValidWorkDiv function

**getValidWorkDiv** function calculates blocksize using **full grid extent!**

```
1 // All kernel inputs are needed because work-division depends on the kernel
2 InitializeBufferKernel initBufferKernel;
3 // Elements per thread needed to determine work-div
4 constexpr alpaka::Vec<Dim, Idx> elemPerThread{1, 1};
5 // Give full-grid thread extent as input!
6 alpaka::KernelCfg<Acc> const kernelCfg = {extent, elemPerThread};
7 // Determine the work-div using kernel and kernel arguments
8 auto workDiv = alpaka::getValidWorkDiv(kernelCfg, devAcc, initBufferKernel,
    uBufAcc.data(), pitchCurrAcc, dx, dy);
```



## Setting WorkDiv Manually: Set 3 vectors of workdiv

```
1 // Set Dim and Index type
2 using Idx = uint32_t;
3 using Dim2D = alpaka::DimInt<2u>; // 2 as a type
4 // 2-element vectors
5 alpaka::Vec<Dim2D, Idx> gridBlockExtent{M,N}; // 2D grid
6 alpaka::Vec<Dim2D, Idx> blockThreadExtent{32,32}; // 2D block
7 alpaka::Vec<Dim2D, Idx> elementExtentPerThread{1,1};
8 // MxN blocks each has 32x32 threads, each level is 2D
9 alpaka::WorkDivMembers<Dim2D, Idx> workdiv2D{gridBlockExtent,
        blockThreadExtent, elementExtentPerThread};
```

```
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});
```

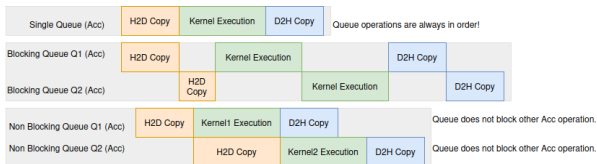
## What is alpaka::Queue

- **alpaka::Queue** is “a queue of tasks”.
- Used for synchronization of tasks like memcpy or kernel-execution
- Queue is always FIFO, everything is sequential (in-order) inside the alpaka::queue
- If the Queue is **non-blocking** the caller(host) is not blocked
- Different non-blocking queues can run in parallel
- Within a single queue accelerator back-ends can be mixed (used in interleaves)

```
1 // Create queue
2 using QueueAcc = alpaka::Queue<Acc, alpaka::NonBlocking>;
3 QueueAcc computeQueue{devAcc};
4 // Copy host -> device, use the queue
5 alpaka::memcpy(computeQueue, uCurrBufAcc, uBufHost);
6 alpaka::wait(computeQueue); // Not needed, we have single queue
7 // Create kernel instance
8 StencilKernel<sharedMemSize> stencilKernel;
9 // Execute kernel using queue
10 alpaka::exec<Acc>(computeQueue, workDiv_manual, stencilKernel...)
```

## Multiple Queues

### ■ Queues are used for synchronization



### ■ Copying and processing data synchronously

```

1  using QueueGpu = alpaka::Queue<AccGpu, alpaka::NonBlocking>;
2  StencilKernel<sharedMemSize> stencilKernel;
3  auto queueGpu1 = QueueGpu{devGpu};
4  auto queueGpu2 = QueueGpu{devGpu};
5  alpaka::memcpy(queueGpu1, uCurrBufAcc, uBufHost);
6  alpaka::wait(queueGpu1);
7  // Run 2 tasks in parallel: memcpy and exec
8  alpaka::memcpy(queueGpu2, uCurrBufAcc2, uBufHost2);
9  alpaka::exec<Acc>(queueGpu1, workDiv_manual, stencilKernel...)
10 alpaka::wait(queueGpu1);
11 alpaka::wait(queueGpu2);

```

## Queue Operations and Tasks

- Wrapping an operation as a **task** is possible.

```
1 auto const taskRunKernel = alpaka::createTaskKernel<Acc>(workDiv,  
    kernel, /* kernel args */);  
2 auto const taskMemcpy = alpaka::createTaskMemcpy<Acc>( uCurrBufAcc,  
    uBufHost);
```

- Tasks are executed by **enqueue()** function.

```
1 alpaka::enqueue(queueA, taskMemCopy);  
2 // taskRunKernel starts after taskMemCopy has finished, even queueA is  
    non-blocking  
3 alpaka::enqueue(queueA, taskRunKernel);
```

- Wait can be used to make queue finish all tasks enqueued

```
1 // wait until all tasks have completed  
2 alpaka::wait(queueA);  
3 // block queueA until otherQueue has completed  
4 alpaka::wait(queueA, otherQueue);
```

- Queues can be checked for completion of all tasks

```
1 bool done = alpaka::empty(myQueue);
```

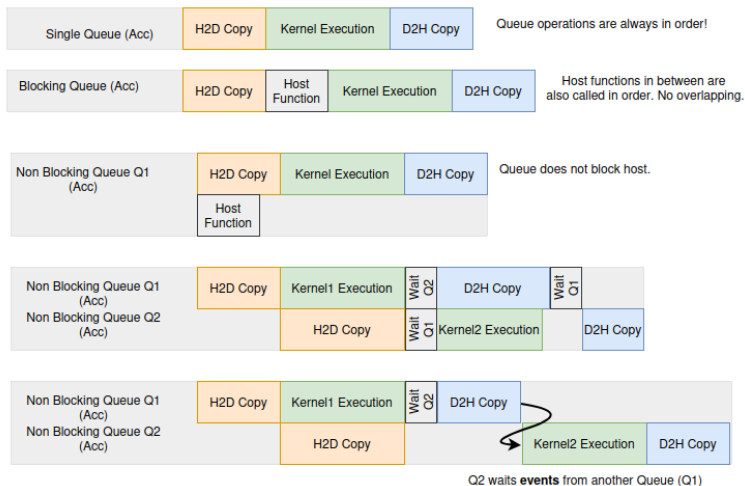
## Queue Operations and Events

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

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



## Some synchronization scenarios



## Going over Kernel execution and copying data

### ■ Create the queue and buffer using device

```
1 // Create queue,  
2 // queue is needed for kernel execution and copies to/from accelerator  
3 alpaka::Queue<Acc, alpaka::NonBlocking> queue{devAcc};  
4 auto uBufAcc = alpaka::allocBuf<float, Idx>(devAcc, extent);
```

### ■ Execute the kernel using the queue, the workdiv and kernel arguments:

```
1 alpaka::exec<Acc>(queue, workDiv, initBufferKernel, uBufAcc.data(),  
    pitchCurrAcc, dx, dy);
```

### ■ Copy the filled buffer back to the host

```
1 // Copy device -> host  
2 // Since buffers know their corresponding devices (host or acc) memcpy does  
    not need any device variable  
3 alpaka::memcpy(queue, uBufHost, uBufAcc);  
4 alpaka::wait(queue);
```

# alpaka Usability and Optimization Features

- 1 Use alpaka **mdspan** to set, get, pass data easily (Hands-On 5)
- 2 Use **Domain Decomposition**: Divide the domain into **chunks** (Hands-On 6)
- 3 Use **2 async queues** for performance increase (Hands-On 7)
- 4 Use **shared memory** for performance increase (Hands-On 8)

## alpaka::experimental::mdspan

### Mdspan a multi-dimensional and non-owning view

- Part of C++23 standard. Can be used with C++17.
- Consists pointer, pitch and data size
- Has member functions to get/set data and to get extents

### Mdspan Installation

- Set `alpaka_USE_MDSPAN` cmake variable to `FETCH` while installing alpaka
- Alternatively, set `alpaka_USE_MDSPAN` cmake variable to `FETCH` while configuring example if it is not already set while installation

```
1 // in build directory
2 cmake -Dalpaka_USE_MDSPAN=FETCH ..
```

### Passing mdspan to kernel

```
1 // Host code: Allocate device memory
2 auto bufDevA = alpaka::allocBuf<DataType, Idx>(devAcc, extentA);
3 // Create mdspan views for device buffers using alpaka::experimental::getMdSpan
4 auto mdDevA = alpaka::experimental::getMdSpan(bufDevA);
5 // Execute the kernel
6 alpaka::exec<Acc>(queue, workDiv, kernel, mdDevA, mdDevB, mdDevC);
```

## Kernel using mdspan instead of data pointer and pitch

### Accessing data at host or at accelerator

```

1  struct MatrixAddKernel
2  {   template<typename TAcc, typename TMdSpan>
3      ALPAKA_FN_ACC void operator()(TAcc const& acc, TMdSpan A, TMdSpan B, TMdSpan
        C) const
4      {
5          auto const i = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0];
6          auto const j = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[1];
7          if(i < C.extent(0) && j < C.extent(1))
8          {
9              C(i, j) = A(i, j) + B(i, j);
10         }
11     } };

```

```

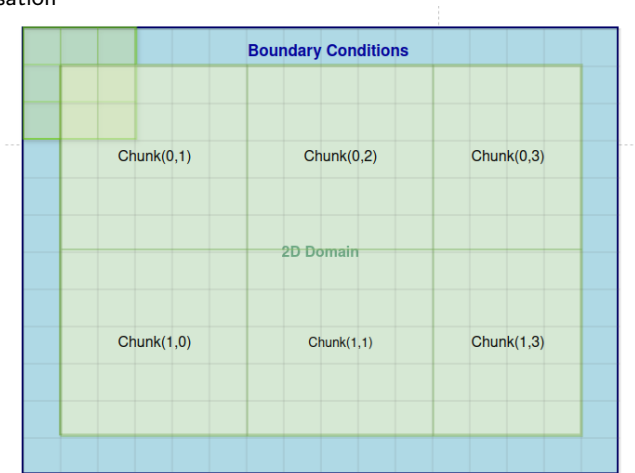
1  // Update the kernel: Use mdspan instead of pointer+pitch
2  struct StencilKernel
3  {
4      template<typename TAcc, typename TDim, typename TIdx>
5      ALPAKA_FN_ACC auto operator()(
6          TAcc const& acc,
7          double const* const uCurrBuf, double* const uNextBuf,
8          alpaka::Vec<TDim, TIdx> const& pitchCurr,
9          alpaka::Vec<TDim, TIdx> const& pitchNext,
10         alpaka::Vec<TDim, TIdx> const& haloSize,
11         double const dx, double const dy, double const dt) const -> void
12     { .....
13     };

```

# Hands-on Session5: Use mdspan to pass data to kernel

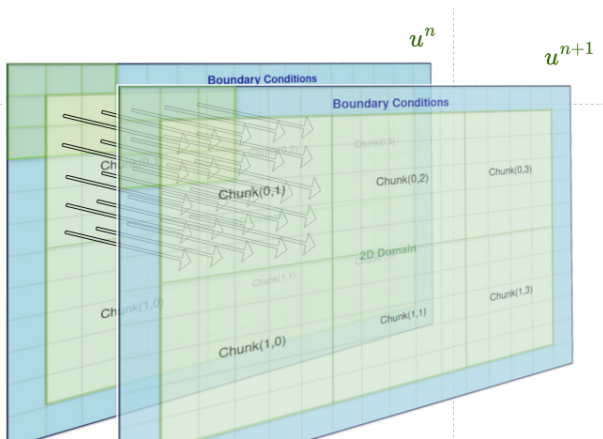
## Chunk Definition

**Chunk:** Subdomains needed for latency management of block level parallelisation



## Calculation by a block of grids of stencil kernel

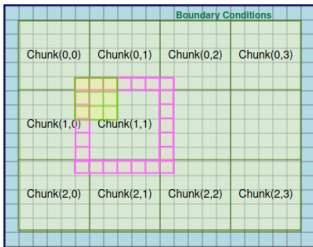
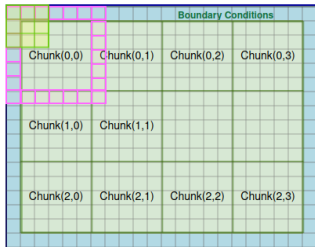
- Chunking is a Domain Decomposition method
- A block of threads update a chunk of heat data
- A grid of threads updates the whole domain



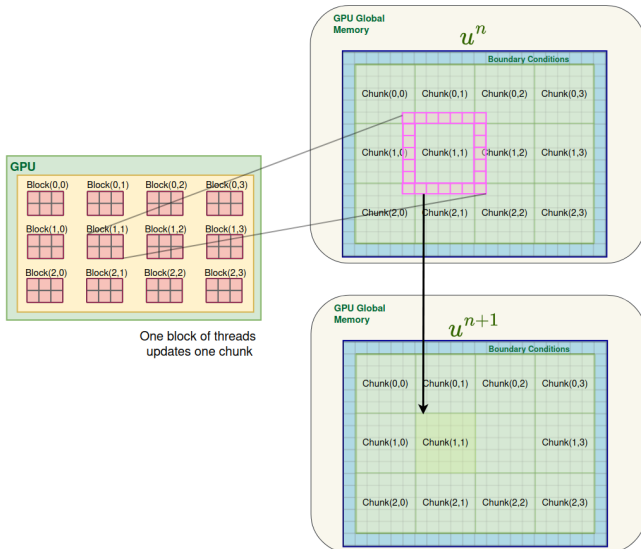


## Chunks in Parallel Grid Computations

- **Halo Region around chunk:** A layer of grid cells surrounding the subdomains.
- **Halo Size:** Typically 1 for a 5-point stencil.
- Chunks-size could be larger than the block size.



## A block is responsible from a chunk



## Determine WorkDiv for Chunked Solution

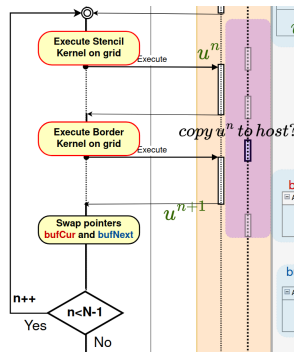
### Set work division fields directly:

```
1 // Define a workdiv for the shared memory based heat eqn solution
2 constexpr alpaka::Vec<Dim, Idx> elemPerThread{1, 1};
3 // Get max threads that can be run in a block for this kernel
4 auto const kernelFunctionAttributes = alpaka::getFunctionAttributes<Acc>(
5     devAcc,
6     stencilKernel,
7     uCurrBufAcc.data(), uNextBufAcc.data(),
8     chunkSize,
9     pitchCurrAcc, pitchNextAcc,
10    dx, dy, dt);
11 auto const maxThreadsPerBlock = kernelFunctionAttributes.maxThreadsPerBlock;
12 auto const threadsPerBlock
13     = maxThreadsPerBlock < chunkSize.prod() ? alpaka::Vec<Dim, Idx>{
14         maxThreadsPerBlock, 1} : chunkSize;
15 alpaka::WorkDivMembers<Dim, Idx> workDiv_manual{numChunks, threadsPerBlock,
16     elemPerThread};
```

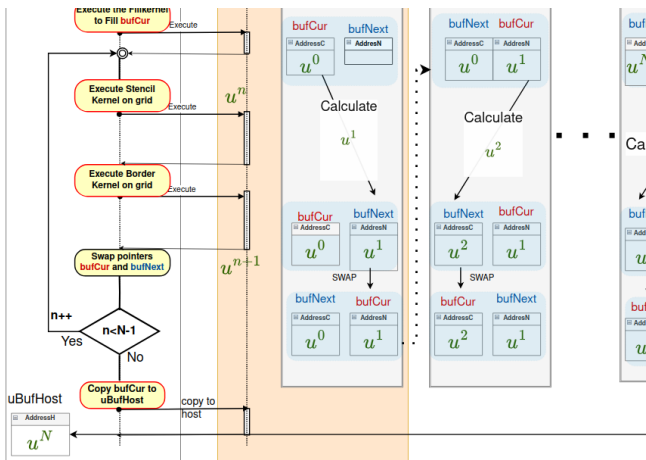
# Hands-on Session6: Optimized Heat Eqn. solution by Domain Decomposition

## Running 2 parallel queues: Additional queue to dump temporary results

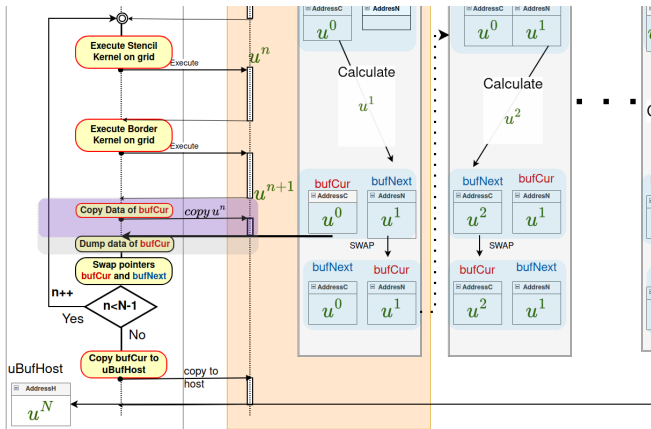
- Create an additional alpaka::queue instance at accelerator to run parallelly
- The temporary heat result  $u^n$  will be copied to host from accelerator at the end of each iteration
- Copying can start while the stencil and boundary kernels are running
- In order to run 2 queues parallelly they should be a NonBlocking queues
- The copied heat data will be used to create an animation of images



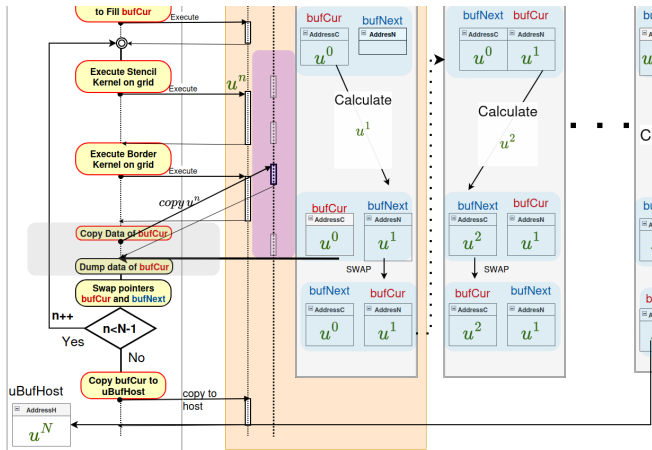
# Current Loop



# Copy $u^n$ back at each iteration sequentially



# Hands on 7: Copy $u^n$ back at each iteration while kernel is running



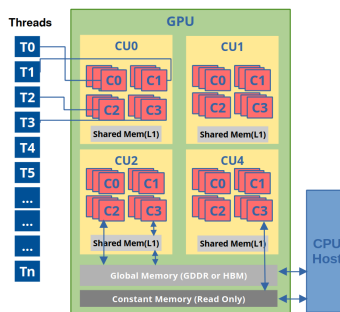


# Hands-On Session7: Running 2 parallel queues to dump heat at each step

# Efficient Stencil Application with Shared Memory

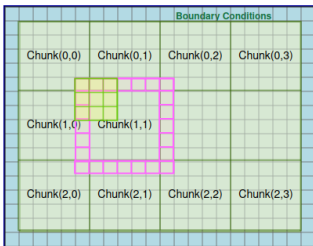
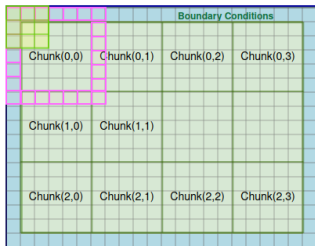
## Shared Memory at GPUs

- A fast, limited-size memory accessible by all threads within a block.
- Faster than global memory. Stores data at CU (or SM).
- Shared Memory allocation can be static or dynamic.
- Filling shared memory is done by the same kernel calculating the stencil
- Threads in a block must synchronize to ensure all data is loaded into shared memory before computation begins.

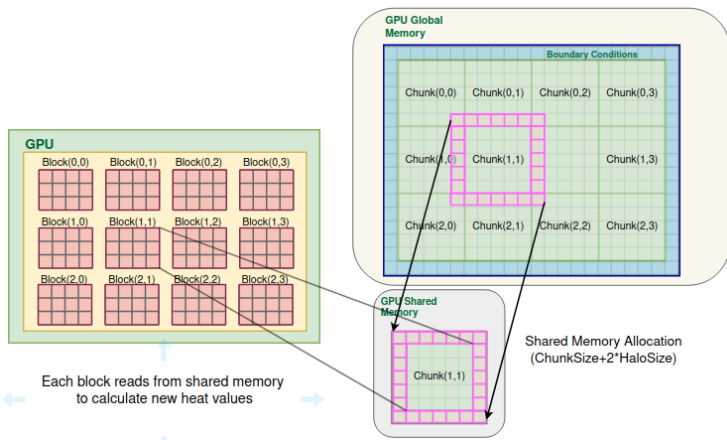


## Shared memory data: chunk with halo

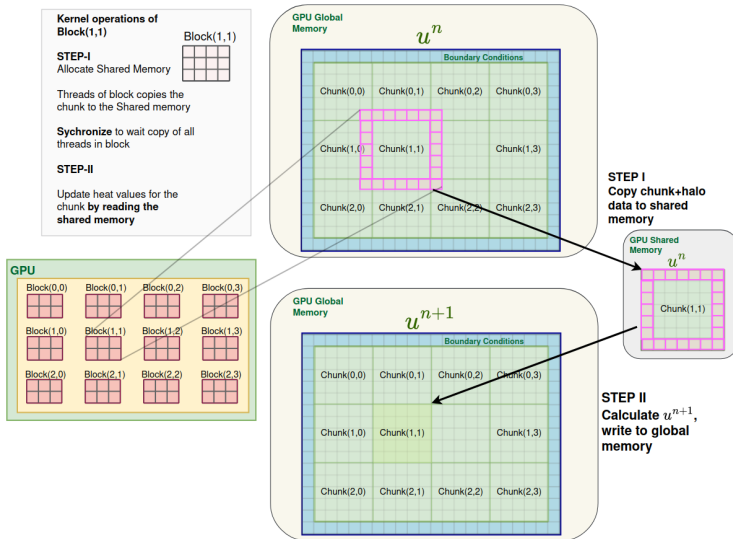
- **Halo Region around chunk:** A layer of grid cells surrounding the subdomains. In order to use the heat value beside the current chunk
- **Halo Size:** Typically 1 for a 5-point stencil.
- Chunks might include more than one blocks depending on the blocksize
- Kernel will install the data to shared memory then use the data from shared memory



# Finding $u^{n+1}$ by using $u^n$ from chunks at shared memory



# Kernel Operations of a Block to Find $u^{n+1}$ for block data



## Hands on 8: Stencil Kernel using shared memory

### ■ Allocate shared memory inside kernel

```

1      // Allocate shared memory inside kernel, this will be done only once per
      // block although it is in the kernel
2      // Size is determined in compile time and is passed to kernel as a type
3      auto& sdata = alpaka::declareSharedVar<double[T_SharedMemSize1D],
      __COUNTER__>(acc);
  
```

### ■ Calculate thread index

### ■ Fill the shared memory by block of threads

### ■ Wait for shared memory to be filled by all block threads

```

1      alpaka::syncBlockThreads(acc);
  
```

### ■ Calculate new heat value using the data from the shared memory

### ■ Set the new heat value

# Hands-on Session8: Optimized Heat Eqn. solution by using shared memory

## Conclusion: Parallel Techniques For Solving Heat Equation

### ■ Kernel Definition

- Kernel to fill a buffer in parallel
- Stencil Kernel for calculating the next set of heat values
- Boundary Kernel

### ■ Work division

- Getting a valid work division according to accelerator
- Setting work-division manually

### ■ Allocating and Setting Memory at Host and Accelerator

- Using `alpaka::buffer`
- Using `alpaka::memcpy`

### ■ Alpaka Structures

- Accelerator, Device, Queue, Task

### ■ Optimizations and Usability

- Using `alpaka Mdspan`
- Domain Decomposition
- Using Multiple Async Queues
- Using GPU's Shared Memory



End of Alpaka Hackathon.  
Any Questions?