

# Heat Equation Solution using alpaka

alpaka Team

HZDR

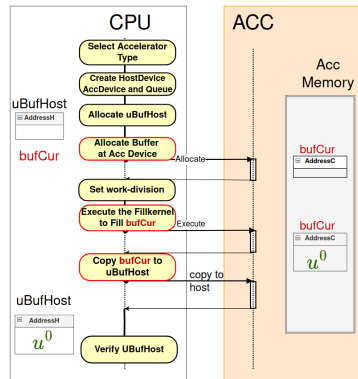
October 1, 2024

# Overview

- 1 Steps filling a buffer in parallel
- 2 Memory Allocation and Passing to Kernel
- 3 Define InitializeBuffer Kernel and Execute
- 4 Introduction for Heat Equation
- 5 Main Simulation Loop of Heat Eqn.
- 6 Heat Eqn. Domain and Stencil
- 7 Setting up the stage to run kernels
- 8 Optimization of Heat Eqn. Solution
- 9 Recap

## 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
- 4 Decide how to parallelize: set work-division
- 5 Decide where will the parallel and non-parallel parts of the code run
- 6 Create the kernel instance and execute 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 domain and halo extents

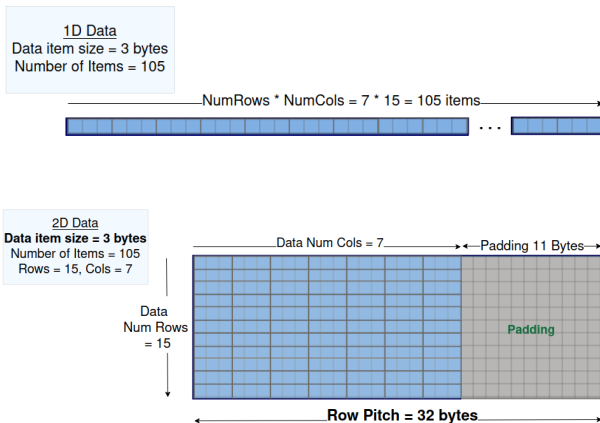
```
1 // alpaka::Vec is a static array similar to std::array.
2 // Dim is a compile-time constant, which is 2.
3 // Create a static array of size Dim.
4
5 constexpr alpaka::Vec<Dim, Idx> numNodes{64, 64};
6 constexpr alpaka::Vec<Dim, Idx> haloSize{2, 2};
7 constexpr alpaka::Vec<Dim, Idx> extent = numNodes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

## Allocate memories at host and accelerator

```
1 // Allocate memory for host-buffer
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);
```

## Allocated area at the memory

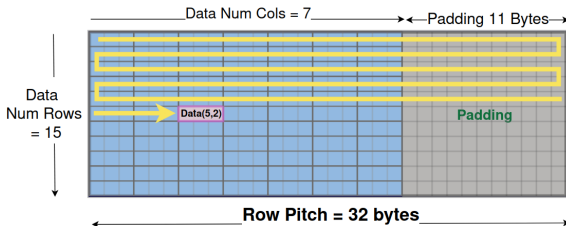
Let's assume that 105 item with 3-byte each will be allocated to pass to the kernel.  
The pitch value (actually the row-pitch) depends on the GPU or CPU type.



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

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

**pitch** = {32bytes,3 bytes}



$$\text{dataPosition} = \text{ptr} + (5 \cdot 32 + 2 \cdot 3) / 3$$

$$\text{dataPosition} = \text{ptr} + (5 \cdot \text{pitch}[0] + 2 \cdot \text{pitch}[1]) / \text{pitch}[1]$$

$$\text{data} = *(\text{dataPosition})$$

## Passing multi dimensional buffer to the kernel

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

Multi-dimensional memory allocated in memory uses aligned rows. Hence, 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

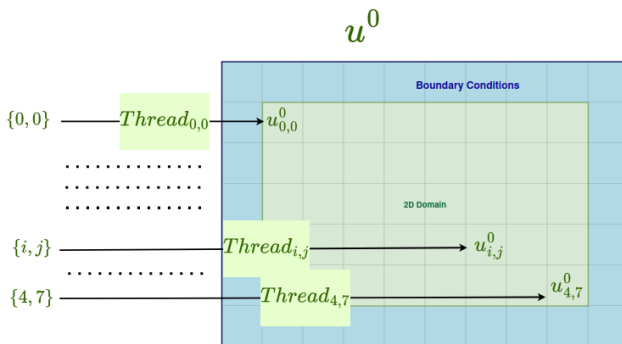
```

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.





## 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 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, gridThreadIdx[1] * dx, gridThreadIdx[0] * dy,
9          0.0);
10     // Calculate data position in buffer, from thread index and pitches
11     auto ptr = getElementPtr(bufData, gridThreadIdx, pitch);
12     // Set the value using the adress
13     *ptr = heatPointValue;
14 } // function operator

```

## Hands-On Session

# Hands-on Session: Filling an accelerator buffer paralelly



# 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: $\alpha = 1$ , $r_X = \frac{\Delta t}{\Delta x^2}$ , $r_Y = \frac{\Delta t}{\Delta y^2}$

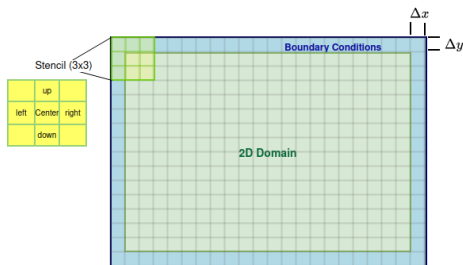
Then  $u_{i,j}^{n+1}$  is:

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

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

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

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



# Main Simulation Loop: Leveraging Parallelism

## ■ 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:

- **Step 1:** Execute `StencilKernel` to compute next values.
- **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.

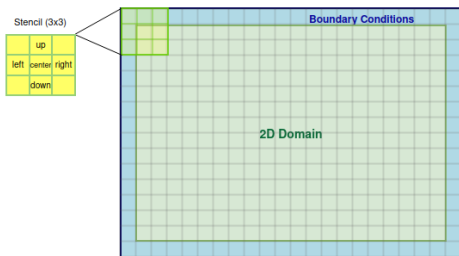
## ■ Parallel Efficiency:

- Subdomains are processed in parallel, with halos ensuring data consistency and correct boundary conditions.
- Optimization: Shared memory optimizes memory access within each block using chunks of data.

## ■ Validation

# Parallel Heat Equation Solution

- **Data Parallelism:** Each point on the grid can be updated independently based on its neighbors, enabling parallel computation.
- **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. A 5-point stencil is needed.



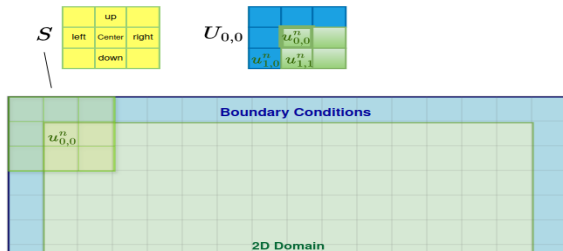
- **Halo Region for BC:** A layer of grid cells surrounding the problem domain for Boundary Conditions.
  - Facilitates stencil operations at the boundaries of subdomains.

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

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

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

- $S$  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  $U_{0,1}^n$

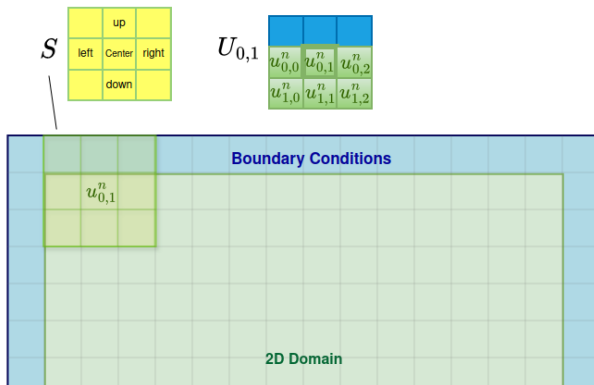
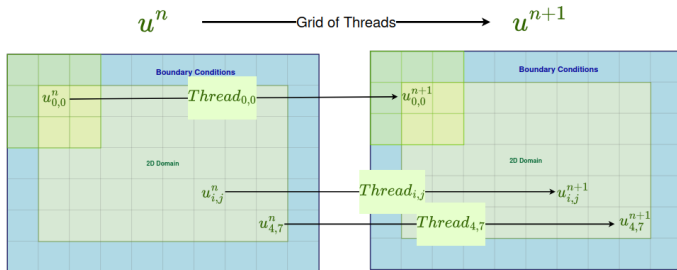


Figure: Second thread calculates  $u_{0,1}^{n+1}$  using

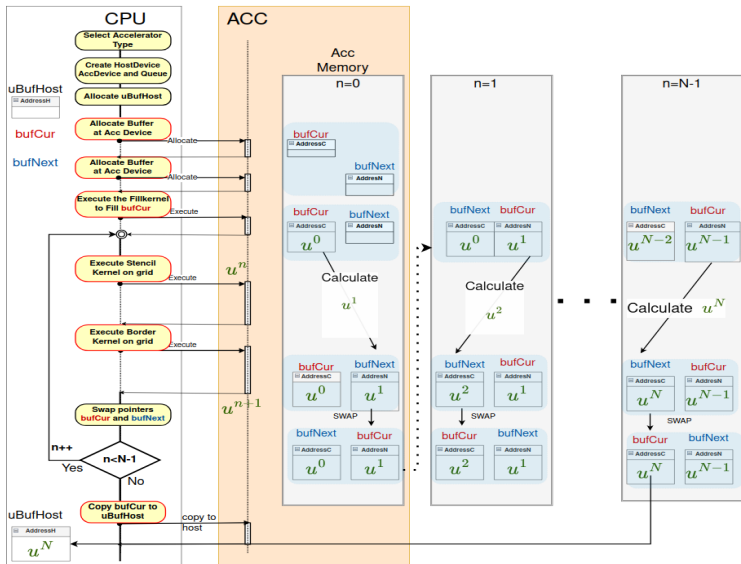


# Stencil Kernel Execution by a grid of threads



- Stencil kernel will update only core nodes not the border
- The workdiv for stencil kernel can be calculated by setting gridthread extent to nodes domain
- The workdiv for the borders kernel would need extended extents, because halo is going to be updated as well

# Complete Heat Equation solution



# 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
- **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* 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 Session

# Hands-on Session: Stencil Kernel

## alpaka Basics

# 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);
3 // get block index on the grid
4 auto gridBlockIdx = alpaka::getIdx<Grid, Blocks>(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);
3 // get pointer to the block shared dynamic memory
4 auto* const sharedN = alpaka::getDynSharedMem<float>(acc);
```

- Enables synchronization on the block level

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

# What is alpaka Buffer, Vector and View

- **alpaka::Buf** is multi-dimensional dynamic (runtime sized) container.
  - Contains memory adress, extent, datatype and the device that memory belongs to!
  - Since buffer already knows the it's device and extent; device to device copy is easy in alpaka
  - 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 — 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



# 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 there is a second queue, queue feature “blocking” and “non-blocking” becomes important
- Different queues can run in parallel for many devices
- Within a single queue accelerator back-ends can be mixed (used in interleaves)

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

# Queue Operations

## ■ Queues are used for synchronization

```
1 // block caller until all operations have completed
2 alpaka::wait(myQueue);
3 // block myQueue until myEvent has been reached
4 alpaka::wait(myQueue, myEvent);
5 // block myQueue until otherQueue has completed
6 alpaka::wait(myQueue, otherQueue);
```

## ■ Queues can be checked for completion of all tasks

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

# 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)

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

- Order of tasks in different queues is unspecified

```
1 alpaka::enqueue(queueA, task1);
2 alpaka::enqueue(queueB, task2);
3 // task2 starts before, after or in parallel to task1 if queueA is non-
   blocking
```

- 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
```

## Execute the Kernel and copy data back to host

### ■ First create the queue

```
1 // Create queue,  
2 // queue is needed for kernel execution and copies to/from accelerator  
3 alpaka::Queue<Acc, alpaka::NonBlocking> queue{devAcc};
```

### ■ 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);
```

## Determine WorkDiv

Let alpaka calculate work division for you:

```
1 // All kernel inputs are needed because work-division depends on the kernel
2 // Create kernel instance
3 InitializeBufferKernel initBufferKernel;
4 // Elements per thread needed to determine work-div
5 constexpr alpaka::Vec<Dim, Idx> elemPerThread{1, 1};
6 // Bundle the extent and elements per thread
7 alpaka::KernelCfg<Acc> const kernelCfg = {extent, elemPerThread};
8 // Kernel input row-pitch and data-type-size
9 auto const pitchCurrAcc{alpaka::getPitchesInBytes(uBufAcc)};
10 // Determine the work-div
11 auto workDiv = alpaka::getValidWorkDiv(kernelCfg, devAcc, initBufferKernel,
    uBufAcc.data(), pitchCurrAcc, dx, dy);
```

# Determine the work-division manually

```

1 // Set Dim and Index type
2 using Dim1D = alpaka::DimInt<1u>; // 1 as a type
3 using Idx = uint32_t;
4 // M blocks each has 4 threads, each level is 1D
5 alpaka::WorkDivMembers<Dim1D, Idx> workdiv1D{M, 4, 1};
6 // Set Dim2D and use same index type
7 using Dim2D = alpaka::DimInt<2u>; // 2 as a type
8 alpaka::Vec<Dim2D, Idx> gridBlockExtent{M,N}; // 2D grid
9 alpaka::Vec<Dim2D, Idx> blockThreadExtent{32,32}; // 2D block
10 alpaka::Vec<Dim2D, Idx> elementExtentPerThread{1,1};
11 // MxN blocks each has 32x32 threads, each level is 2D
12 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});

```

## Optimization and usability

# alpaka Usability and Optimization Features

- 1 Use alpaka **mdspan** to set,get, pass buffers easily
- 2 Use **Domain Decomposition**: Divide the domain in **chunks**
- 3 Use **2 async queues** for performance increase
- 4 Use **shared memory** for performance increase

# alpaka::experimental::mdspan

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

- Part of C++23 standard. Can be used with C++17.
- Consists Data Pointer, Data Pitch and Data item 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 ..
```

## Create an mdspan view of a buffer, then pass to the 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 buffer pointer and pitch

## Example usage to access/set multi-dim data at host or in kernel

```

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
2  struct StencilKernel
3  {
4      template<typename TAcc, typename TDim, typename TIdx, typename TMdSpan>
5      ALPAKA_FN_ACC auto operator()(
6          TAcc const& acc,
7          TMdSpan uCurrBuf,
8          TMdSpan uNextBuf,
9          alpaka::Vec<TDim, TIdx> const chunkSize,
10         double const dx,
11         double const dy,
12         double const dt) const -> void
13     { ....
14     } };

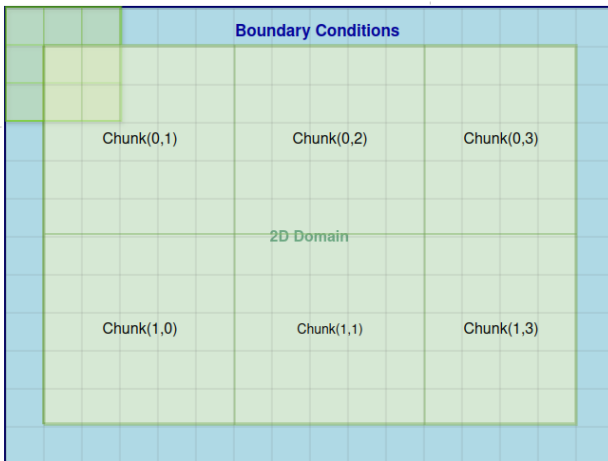
```

## Hands-On Session

# Hands-on Session: Use mdspan for the kernel using shared memory

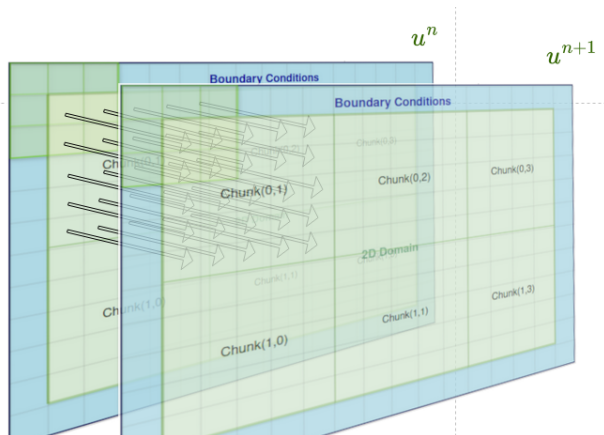
# Chunk Definition

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



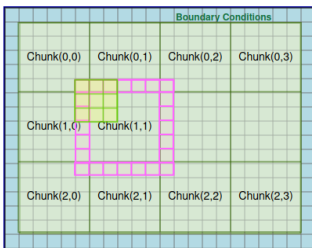
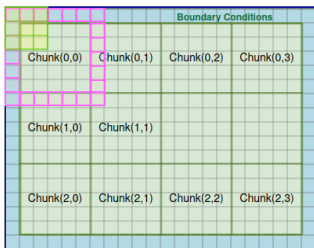
## Calculation by a block of grids of stencil kernel

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

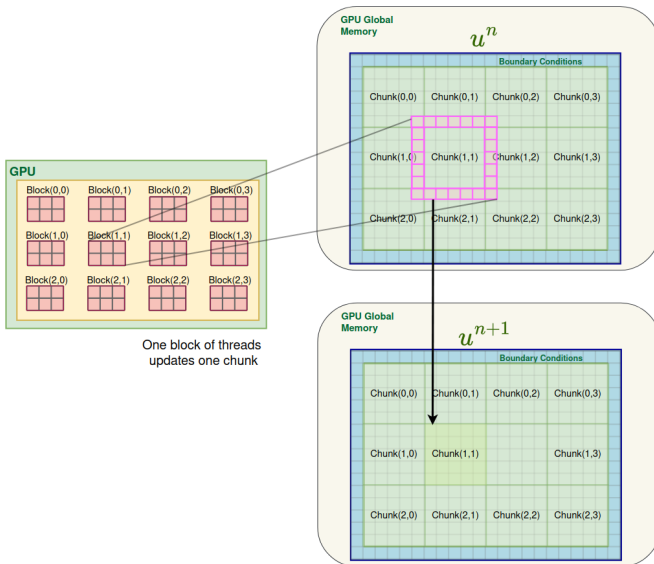


# Chunks in Parallel Grid Computations - I

- **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



## Chunks in Parallel Grid Computations - II



# 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};

```

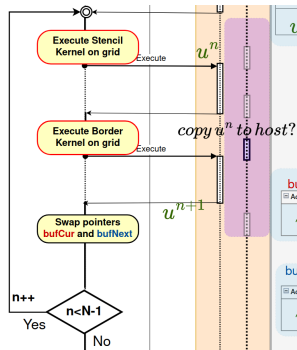
## Hands-On Session

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

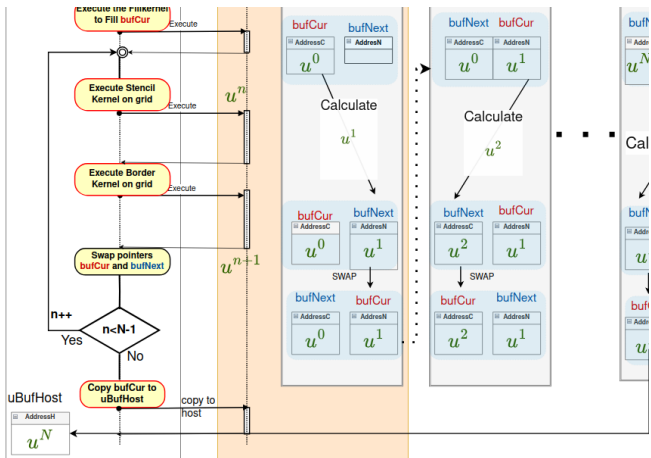


## Running 2 parallel queues: Additional queue to dump temporary results

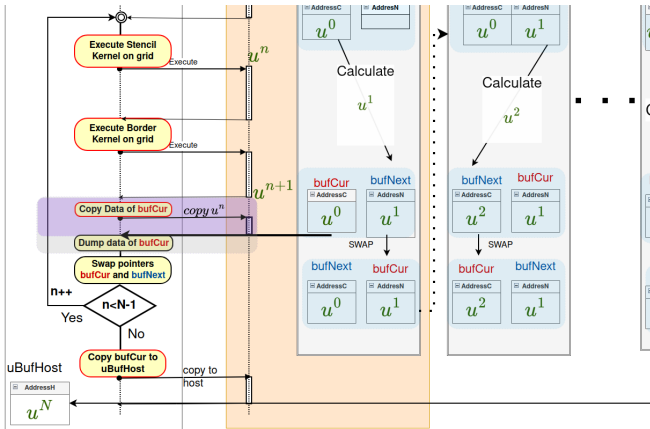
- Create an additional alpaka::queue instance at accelerator to run parallelly
- The temporary heat result  $u^n$  will be copied to host from accelerator at the end of each iteration
- Copying can start while the stencil and boundary kernels are running
- In order to run 2 queues parallelly they should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



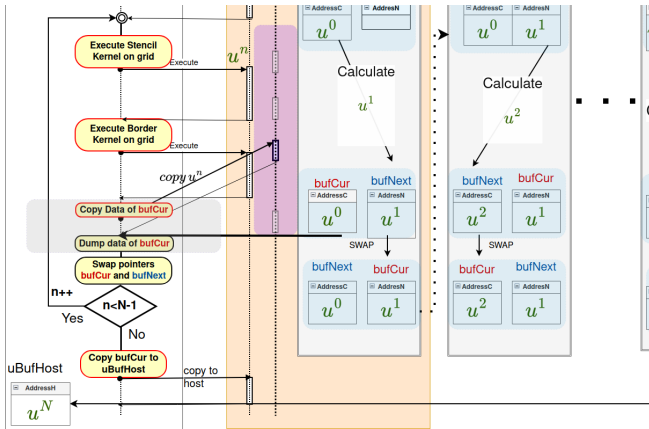
## Current Loop



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



# Copy $u^n$ back at each iteration in parallel



## Hands-On Session

# Running 2 parallel queues

# Efficient Stencil Application with Shared Memory

## Shared Memory at GPUs

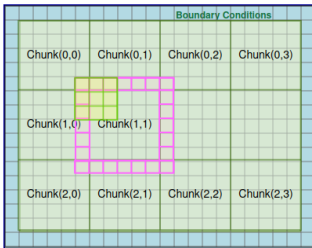
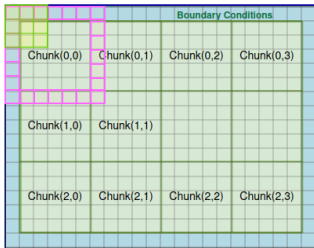
- A fast, limited-size memory accessible by all threads within a block.
- Used to store data locally in Compute Unit(or SM), reducing the need to access slower global memory.
- Shared Memory allocation can be static or dynamic
  - Static (compile time determined extent)
  - Dynamic (runtime determined extent)
- 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.

## Benefits:

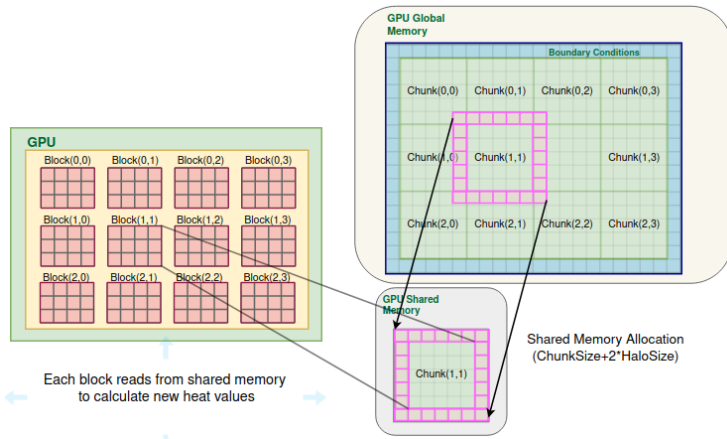
- Reduces memory latency by storing the working set of data (halo + core) in shared memory.

## 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

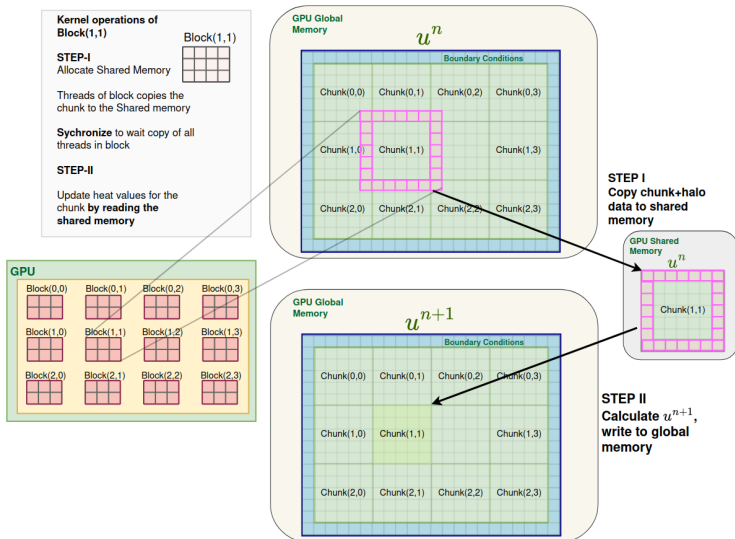


## Updating chunks using shared memory





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



# Steps of 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 Session

# Hands-on Session: 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 Mdspace`
- Domain Decomposition
- Using Multiple Async Queues
- Using GPU's Shared Memory

# Questions?