Heat Equation Solution using alpaka

alpaka Team

HZDR

October 14, 2024

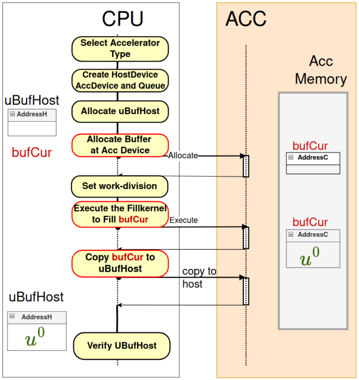
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Overview

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

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

Define domain and halo extents

```cpp
    // alpaka::Vec is a static array similar to std::array.
    // Dim is a compile-time constant, which is 2.
    // Create a static array of size Dim.

     constexpr alpaka::Vec<Dim, Idx> numNodes{64, 64};
     constexpr alpaka::Vec<Dim, Idx> haloSize{2, 2};
     constexpr alpaka::Vec<Dim, Idx> extent = numNodes + haloSize;
```

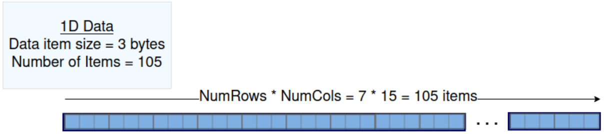
Allocate memories at host and accelerator

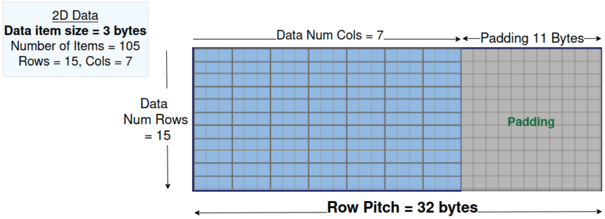
```cpp
     // Allocate memory for host-buffer
     auto uBufHost = alpaka::allocBuf<double, Idx>(devHost, extent);

     // Allocate memory for accelerator buffer
     auto uBufAcc = alpaka::allocBuf<double, Idx>(devAcc, extent);
```

Allocated area at the memory

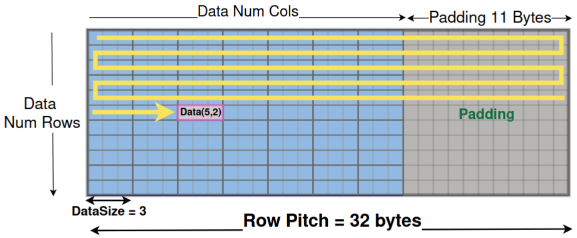
Let's assume that 105 item with 3-byte each will be allocated to pass to the kernel.
The pitch value (actually the row-pitch) depends on the GPU or CPU type.





How to access data given the pointer and the pitch?

How to access data at index (5,2) given the pointer ptr and pitch?
pitch = {32bytes,3 bytes} as {row-pitch, datasize}



```
dataPosition = (char *)ptr+ 5*pitch[0]+2*pich[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**
  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.

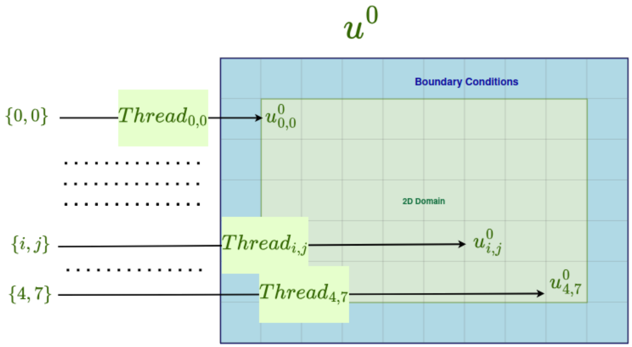
```
    // Signature of function operator of the Kernel
    template<typename TAcc, typename TDim, typename TIdx>
    ALPAKA_FN_ACC auto operator()(
        TAcc const& acc,
        double* const bufData,
        // 2 variables row-pitch and data-type size
        alpaka::Vec<TDim, TIdx> const pitch,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

- **Simple Alternative:** Pass an **alpaka mdspan** object

```
    template<typename TAcc, typename TDim, typename TIdx, typename TMdSpan>
    ALPAKA_FN_ACC auto operator()(
        TAcc const& acc,
        TMdSpan uAccMdSpan
        ...) 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 Adress in Buffer:** Calculate the corresponding memory adress in buffer using thread index. Take into account row-pitch and data-size
- **Set Value at the Adress:** Set the data at the memory position to the calculated initial condition.

```
template<typename TAcc, typename TDim, typename TIdx>
ALPAKA_FN_ACC auto operator()(
    TAcc const& acc, double* const bufData,
    alpaka::Vec<TDim, TIdx> const pitch, double dx, double dy) const -> void {
// Get 2D thread index using alpaka index function
.....
// Calculate analytical solution at point
auto heatAtPointValue = analyticalSolution(acc, gridThreadIdx[1] * dx, gridThreadIdx[0] * dy,
    0.0);
// Calculate data position in buffer, from thread index and pitches
auto ptr = getElementPtr(bufData, gridThreadIdx, pitch);
// Set the value using the adress
*ptr    = heatPointValue;
} // function operator
```

Hands-on Session: Filling an accelerator buffer paralelly

The Heat Equation

- **The heat equation models the Heat Diffusion over time in a given medium.**

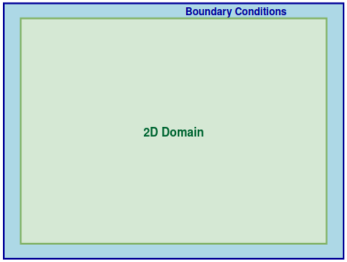
$$\frac{\partial u(x,y,t)}{\partial t} = \alpha \left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \right)$$

Difference approximations for Time and Spatial Derivatives:

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

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

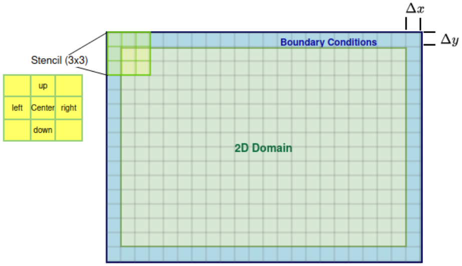
  Then $u_{i,j}^{n+1}$ is:

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

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

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

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

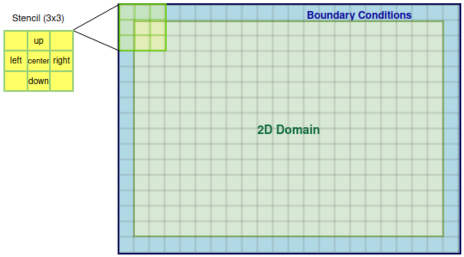


Main Simulation Loop: Leveraging Parallelism

- **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.



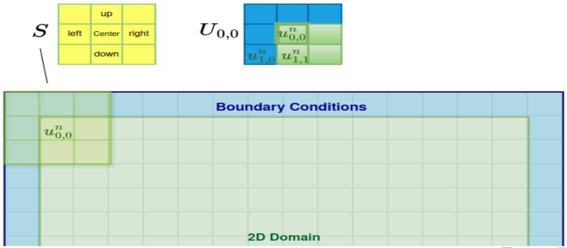
- **Halo Region for BC:** A layer of grid cells surrounding the problem domain for Boundary Conditions.
  - Facilitates stencil operations at the boundaries of subdomains.

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

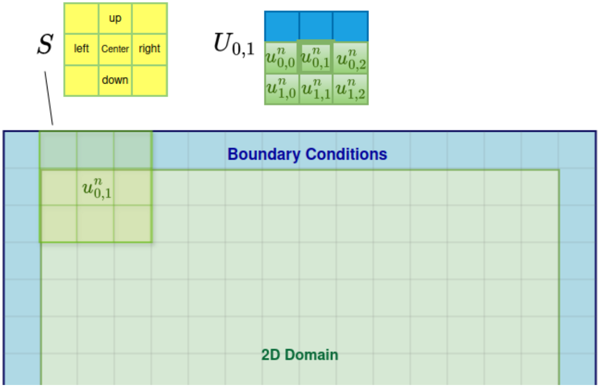
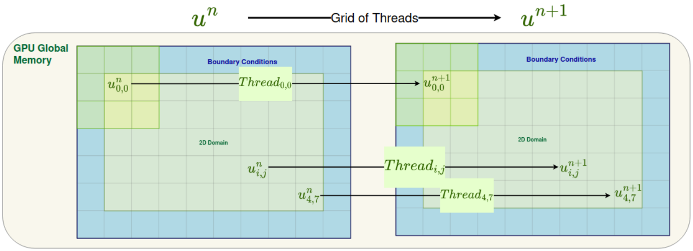


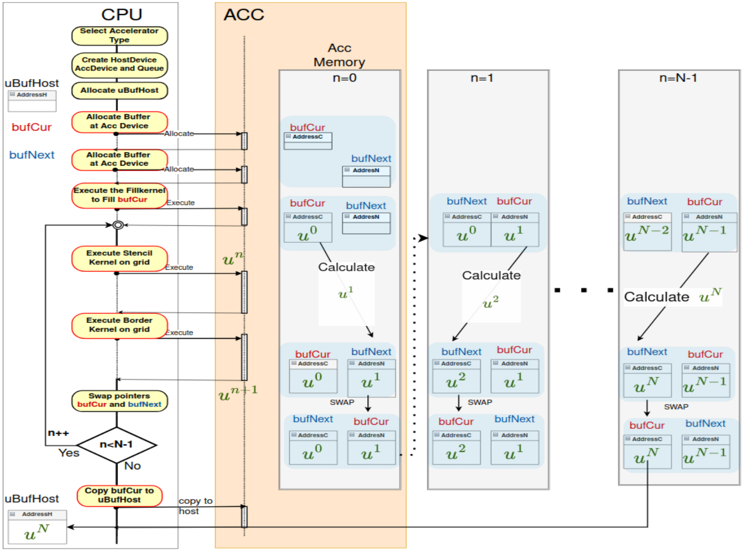
Figure: Second thread calculates $u_{0,1}^{n+1}$ using

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 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 Adress in Buffer:** Calculate the corresponding memory adress 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 Adress:** Set the data at the memory adress.

```cpp
struct StencilKernel
{
    template<typename TAcc, typename TDim, typename TIdx>
    ALPAKA_FN_ACC auto operator()(
        TAcc const& acc,
        double const* const uCurrBuf, double* const uNextBuf,
        alpaka::Vec<TDim, TIdx> const chunkSize,
        alpaka::Vec<TDim, TIdx> const pitchCurr, alpaka::Vec<TDim, TIdx> const pitchNext,
        double const dx,double const dy, double const dt) const -> void
    {
        ...
    }
};
```

Hands-on Session:Stencil Kernel

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 accelarator
5. WorkDiv
6. Define Queue

Accelerator, Device and Host

Define number of dim and index type

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

Define the accelerator

```
// AccGpuCudaRt, AccGpuHipRt, AccCpuThreads, AccCpuSerial,
// AccCpuOmp2Threads, AccCpuOmp2Blocks, AccCpuTbbBlocks
using Acc = alpaka::AccGpuHipRt<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);
```

Select a host and hostype to allocate memory for data

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

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

```
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

```
// 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& sdata = alpaka::declareSharedVar<double[T_SharedMemSize1D], __COUNTER__>
    acc);
// get pointer to the block shared dynamic memory
auto* const sharedN = alpaka::getDynSharedMem<float>(acc);
```

  - Enables synchronization on the block level

```
// synchronize all threads within the block
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 [][].

```
// Allocate buffers
auto bufCpu = alpaka::allocBuf<float, Idx>(devCpu, extent);
auto bufGpu = alpaka::allocBuf<float, Idx>(devGpu, extent);
....
// Copy buffer from CPU to GPU — destination comes first
alpaka::memcopy(gpuQueue, bufGpu, bufCpu);
// cuda way: cudaMemcpy(b_d, b_host, sizeof(float)*N, cudaMemcpyHostToDevice)
```

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

```
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 sycnhronization 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)

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

Queue Operations

- Queues are used for synchronization

```cpp
// block caller until all operations have completed
alpaka::wait(myQueue);
// block myQueue until myEvent has been reached
alpaka::wait(myQueue, myEvent);
// block myQueue until otherQueue has completed
alpaka::wait(myQueue, otherQueue);
```

- Queues can be checked for completion of all tasks

```cpp
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)

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

- 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
```

Execute the Kernel and copy data back to host

- First create the queue

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

- Execute the kernel using the queue, the workdiv and kernel arguments:

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

- Copy the filled buffer back to the host

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

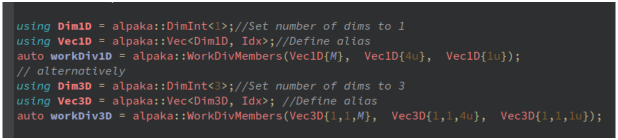
Determine WorkDiv

Let alpaka calculate work division for you:

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

Determine the work-division manually

```cpp
// Set Dim and Index type
using Dim1D = alpaka::DimInt<1u>; // 1 as a type
using Idx = uint32_t;
// M blocks each has 4 threads, each level is 1D
alpaka::WorkDivMembers<Dim1D, Idx> workdiv1D{M, 4, 1};
// Set Dim2D and use same index type
using Dim2D = alpaka::DimInt<2u>; // 2 as a type
alpaka::Vec<Dim2D, Idx> gridBlockExtent{M,N}; // 2D grid
alpaka::Vec<Dim2D, Idx> blockThreadExtent{32,32}; // 2D block
alpaka::Vec<Dim2D, Idx> elementExtentPerThread{1,1};
// MxN blocks each has 32x32 threads, each level is 2D
alpaka::WorkDivMembers<Dim2D, Idx> workdiv2D{gridBlockExtent,
    blockThreadExtent, elementExtentPerThread};
```


```cpp
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});
```

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 asynch 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

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

Create an mdspan view of a buffer, then pass to the kernel

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

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Kernel using mdspan instead of data buffer pointer and pitch

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

```
 struct MatrixAddKernel
 {   template<typename TAcc, typename TMdSpan>
     ALPAKA_FN_ACC void operator()(TAcc const& acc, TMdSpan A, TMdSpan B, TMdSpan
           C) const
     {
         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);
         }
     } };
```

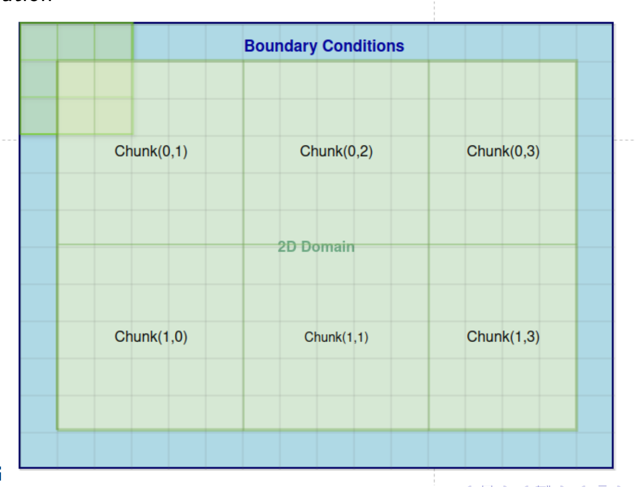
```

struct StencilKernel
{
    template<typename TAcc, typename TDim, typename TIdx, typename TMdSpan>
    ALPAKA_FN_ACC auto operator()(
        TAcc const& acc,
        TMdSpan uCurrBuf,
        TMdSpan uNextBuf,
        alpaka::Vec<TDim, TIdx> const chunkSize,
        double const dx,
        double const dy,
        double const dt) const -> void
    { ....
    } };
```

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

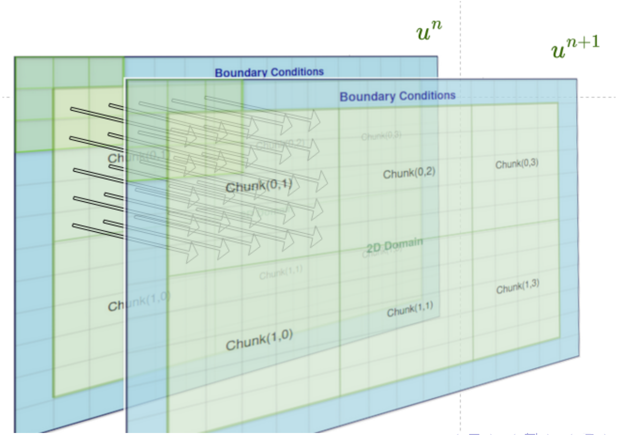
Chunk Definition

Chunk: Subdomains needed for latency management of block level parallelisation



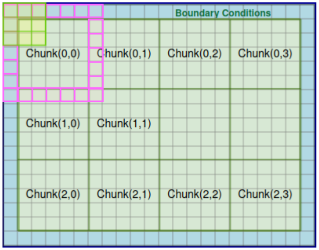
Calculation by a block of grids of stencil kernel

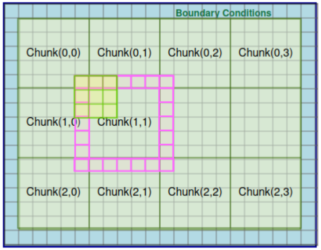
- Chunking is a domain decomposition method
- A block of threads update a chunk of heat data
- A grid of threads updates the whole domain



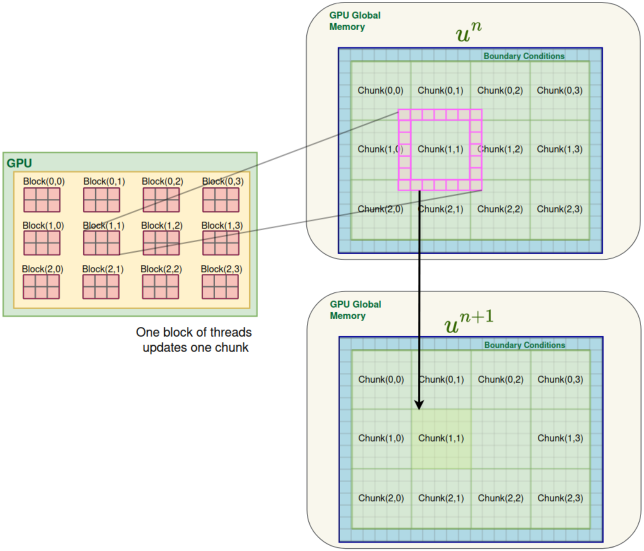
Chunks in Parallel Grid Computations - I

- **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



One block of threads updates one chunk

Determine WorkDiv for Chunked Solution

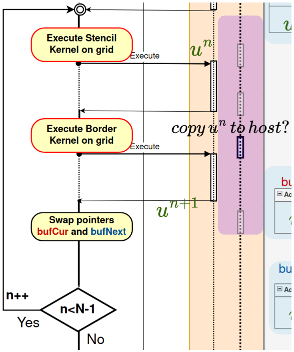
Set work division fields directly:

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

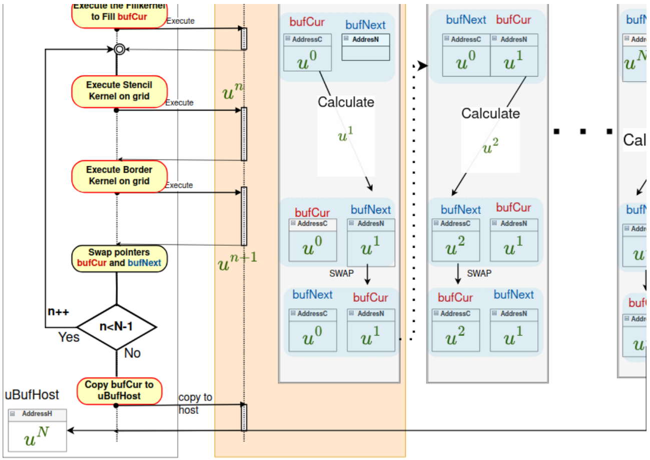
Hands-on Session: Optimized Heat Eqn. solution by Domain Decomposition

Running 2 parallel queues: Additional queue to dump temporary results

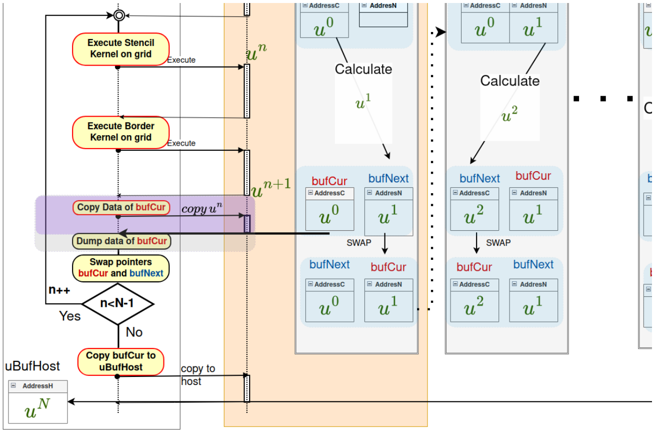
- Create an additional alpaka::queue instance at accelerator to run parallely
- The temporary heat result u^n will be copied to host from accelerator at the end of each iteration
- Copying can start while the stencil and boundary kernels are running
- In order to run 2 queues paralelly they should be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


Execute Stencil Kernel on grid
Execute
u^n
copy u^n to host?
Execute Border Kernel on grid
Execute
u^{n+1}
Swap pointers bufCur and bufNext
n++
n<N-1
Yes
No

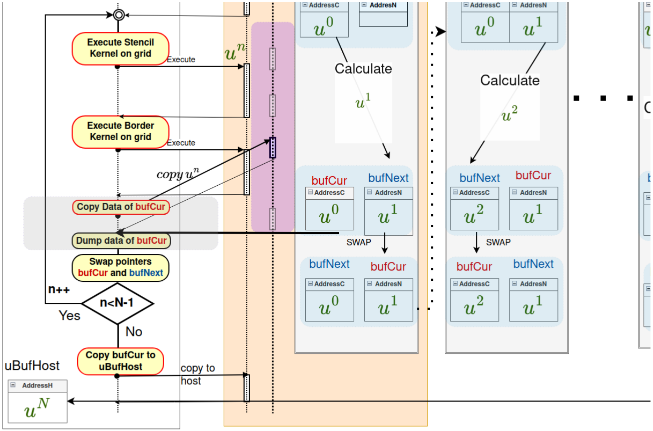
Current Loop



Copy u^n back at each iteration sequentially



Copy u^n back at each iteration in parallel



Running 2 parallel queues

Efficient Stencil Application with Shared Memory

Shared Memory at GPUs

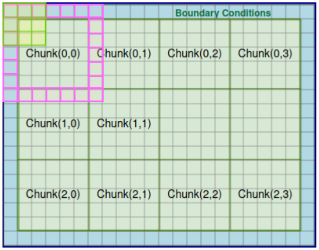
- A fast, limited-size memory accessible by all threads within a block.
- Used to store data locally in Compute Unit(or SM), reducing the need to access slower global memory.
- Shared Memory allocation can be static or dynamic
  - Static (compile time determined extent)
  - Dynamic (runtime determined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

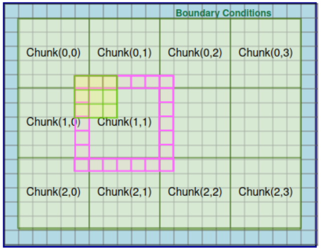
Benefits:

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

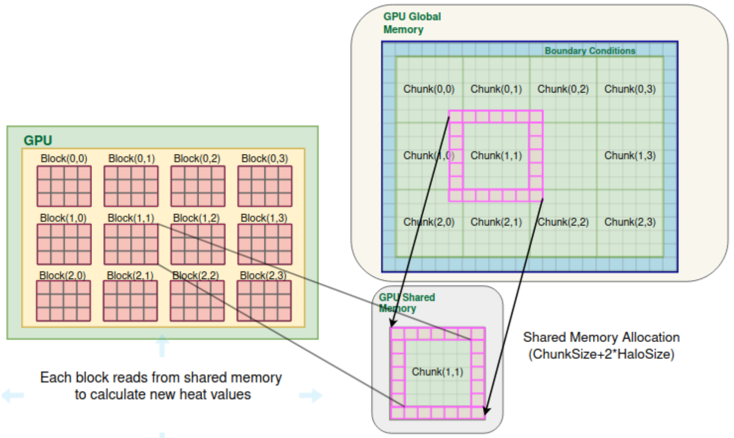
Shared memory data: chunk with halo

- **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

Kernel operations of Block(1,1)

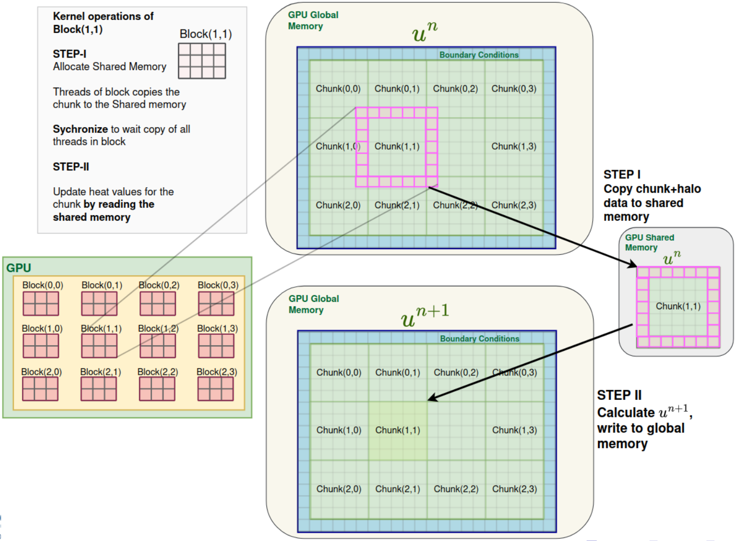
STEP-I
Allocate Shared Memory

Threads of block copies the chunk to the Shared memory

Sychronize to wait copy of all threads in block

STEP-II

Update heat values for the chunk **by reading the shared memory**



STEP I
Copy chunk+halo data to shared memory

STEP II
Calculate u^{n+1}, write to global memory

CENTER FO
SYSTEMS UNDERSTANDING

Steps of Stencil Kernel using shared memory

- **Allocate shared memory inside kernel**

```
    // Allocate shared memory inside kernel, this will be done only once per
        block although it is in the kernel
    // Size is determined in compile time and is passed to kernel as a type
     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**

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

- **Calculate new heat value using the data from the shared memory**
- **Set the new heat value**

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

Questions?