Heat Equation Simulation using Alpaka

Alpaka Team

HZDR

September 5, 2024

Overview

The Heat Equation

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

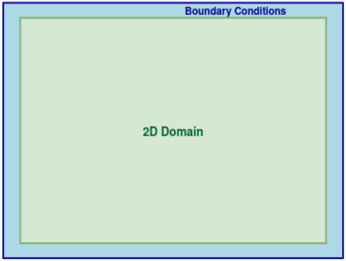
$$\frac{\partial u(x,y,t)}{\partial t} = \alpha \left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \right)$$

Difference approximations for Time and Spatial Derivatives:

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

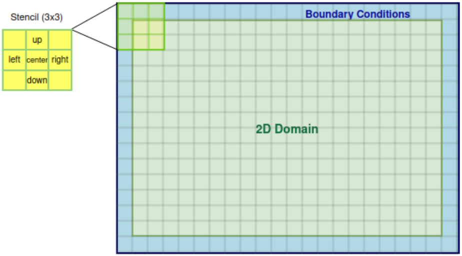
- **Resulting difference equation:**

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



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
- Calculation of $u_{0,0}^{n+1}$ by a single thread

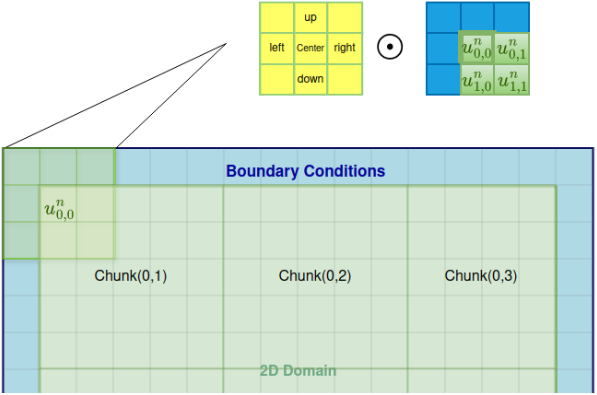


Figure: First thread calculates $u_{0,0}^{n+1}$ using frobenious inner product of 3x3 matrices

- Second thread calculates $u_{0,1}^{n+1}$

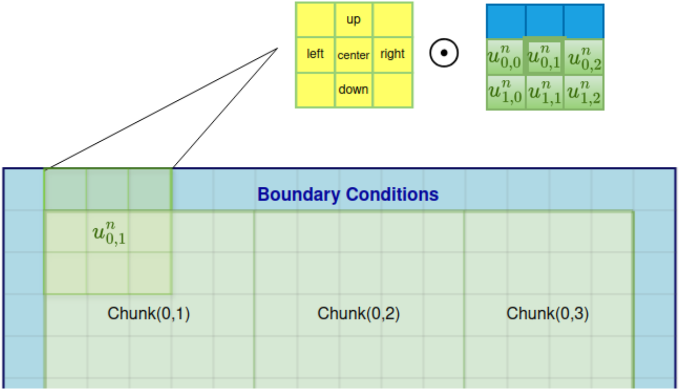
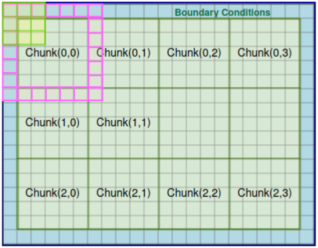
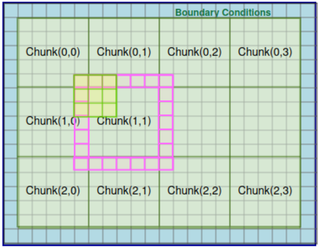


Figure: Second thread calculates $u_{0,1}^{n+1}$ using

Chunks in Parallel Grid Computations

- **Chunk:** Subdomains needed for block level parallelisation
- **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.





Chunk Definition (in detail?)

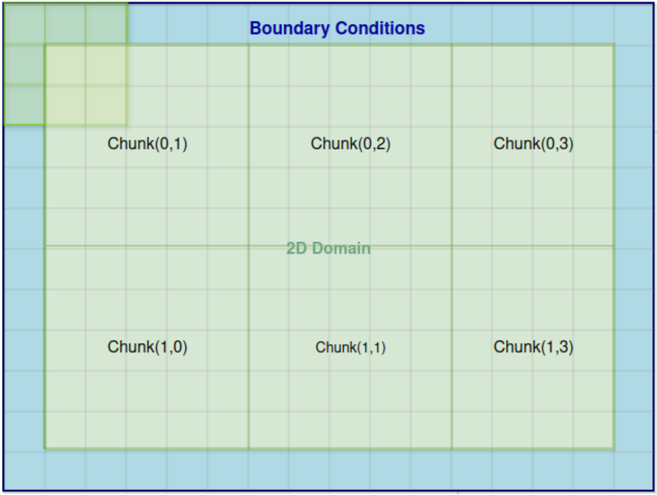


Figure: Heat values correponding to each chunk is updated by a block

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

Select a device from platform of Acc

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

Select a host and hosttype 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;
```

Allocate memory at Host and at Device

```
// Allocate host memory buffers
using BufHost = alpaka::Buf<DevHost, DataType, Dim, Idx>;
BufHost bufHostA(alpaka::allocBuf<DataType, Idx>(devHost, extent));

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

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

Need a queue to copy the data to the device

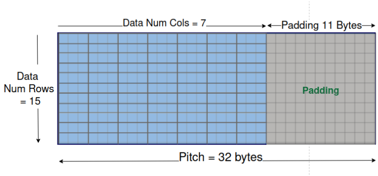
```cpp
using Acc = alpaka::AccCpuSerial<Dim, Idx>;
auto const platformAcc = alpaka::Platform<Acc>{};
auto const devAcc = alpaka::getDevByIdx(platformAcc, 0);
// A queue is needed for all acc related operations
alpaka::Queue<Acc, alpaka::Blocking> queue(devAcc);

// Define the 2D extents (dimensions)
Vec const extentA(static_cast<Idx>(M), static_cast<Idx>(K));

// Allocate host memory, the memory size is determined by extent
auto bufHostA = alpaka::allocBuf<DataType, Idx>(devHost, extentA
    );

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

// Copy data to device, use host buffer and device buffer
// Queue must be an accelerator queue
alpaka::memcpy(queue, bufDevA, bufHostA);
```

Passing multi dimentional data to the kernel

Multi-dimentional 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.

- **Pass 3 variables to kernel: pointer, pitch, and datasize**

```
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 pitchCurr,
    alpaka::Vec<TDim, TIdx> const pitchNext,
    ...) const -> void
```



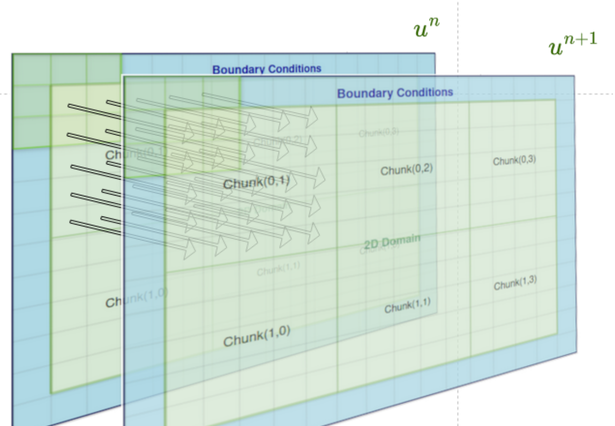
2D Buffer with size 15x7 in memory

- **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 uCurrBuf,
    TMdSpan uNextBuf
    ...) const -> void
```

WorkDiv-I: Setting workdiv fields directly

```
auto blocksPerGrid = alpaka::Vec<Dim, Idx>{M/8, N/128};
auto threadsPerBlock = alpaka::Vec<Dim, Idx>{8, 128};
auto elementsPerThread = alpaka::Vec<Dim, Idx>{1u, 1u};
using WorkDiv = alpaka::WorkDivMembers<Dim, Idx>;
auto workDiv = WorkDiv{blocksPerGrid, threadsPerBlock,
    elementsPerThread};
```



WorkDiv-II: Let Alpaka calculate for you

```
using Acc = alpaka::AccCpuSerial<Dim, Idx>;
auto const devAcc = alpaka::getDevByIdx(platformAcc, 0);

// Define the 2D extents as {Width, Height} of matrix
alpaka::Vec<Dim, Idx> const extentAsThreadsPerGrid(M, N);
auto elementsPerThread = alpaka::Vec<Dim, Idx>{1u, 1u};
MatrixMulKernel kernel;

// Let alpaka calculate good block and grid sizes given our full
    problem extent
alpaka::KernelCfg<Acc> const kernelCfg = {extentAsThreadsPerGrid,
    elementsPerThread};
auto const workDiv = alpaka::getValidWorkDiv<Acc>(kernelCfg, devAcc,
     kernel, // kernel params here
);
```

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 (chunk size, block size, etc.).
- **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.
- **Validation**

Executing the Kernel

- **Execution Flow:**
  - Each kernel is executed on the selected accelerator (e.g., CPU, GPU).
  - Halo regions and shared memory are leveraged for optimal parallel performance.
- **Kernel Execution:**

```
alpaka::exec<Acc>(
queue1,
workDiv_manual,
stencilKernel,
uCurrBufAcc.data(),
uNextBufAcc.data(),
chunkSize,
dx, dy, dt);
```

- **Run Example:** Execute for all enabled accelerators (e.g., CUDA, HIP, OpenMP).

Applying Boundary Conditions in Parallel

- **Boundary Kernel:** Ensures correct values at the boundaries of the grid.
- **Challenges in Parallel Computing:**
  - Boundary points might need special handling, particularly when subdomains are processed independently.
  - Halo regions play a crucial role here.
- **Code Snippet:**

```
if (gridBlockIdx[0] == 0) {
applyBoundary(globalIdx, chunkSize[1], true);
}
```

Efficient Stencil Computation with Shared Memory

- **Shared Memory:**
  - A fast, limited-size memory accessible by all threads within a block.
  - Used to store grid points locally, reducing the need to access slower global memory.
- **Benefits:**
  - Reduces memory latency by storing the working set of data (halo + core) in shared memory.
  - Enables efficient data reuse across threads in the same block.
- **Example:**

```
auto& sdata =
alpaka::declareSharedVar<double[T_SharedMemSize1D],
__COUNTER__>(acc);
```

- **Synchronization ...I/O:**
  - Threads in a block must synchronize to ensure all data is loaded into shared memory before computation begins.

Using multiple queues

Results and Performance in Parallel Execution

- **Validation:**
  - Accuracy of results compared to the analytical solution.
  - Performance considerations: Speedup achieved by parallelizing the computation.
- **Output:**
  - Print whether the results are correct.
  - Report on the maximum error.
  - Discuss any performance metrics (e.g., execution time).
- **Visual Output (Optional):**
  - Periodic snapshots of the temperature distribution.

Conclusion: Parallel Techniques for Efficient Simulation

- **Key Takeaways:**
  - Efficient use of shared memory significantly boosts performance in parallel computations.
  - Halo regions are crucial for managing data dependencies in stencil operations.
  - The combination of Alpaka's abstraction and careful memory management enables scalable and portable parallel solutions.

Questions?