COMPUTATIONAL PHYSICS: INCLUDES PARALLEL COMPUTING/PARALLEL PROGRAMMING

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Date: 23 mai 2016.
Key words and phrases. Computational Physics, Parallel Computing, Parallel Programming.

Abstract. Everything about Computational Physics, including Parallel computing/ Parallel programming.

Part 1. Algorithms as needed

1. Divide and Conquer

Divide and Conquer

`https://classes.soe.ucsc.edu/cmps102/Fall01/solutions4.pdf`
`http://www3.cs.stonybrook.edu/~skiena/373/hw/hw.pdf`

Part 2. Parallel Computing

2. Udacity Intro to Parallel Programming : Lesson 1 - The GPU Programming Model

Owens and Luebki pound fists at the end of this video. =)))) Intro to the class.

2.1. Running CUDA locally.

Also, Intro to the class, in Lesson 1 - The GPU Programming Model, has links to documentation for running CUDA locally; in particular, for Linux: `http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html`. That guide told me to go download the NVIDIA CUDA Toolkit, which is the https://developer.nvidia.com/cuda-downloads.

For *Fedora*, I chose Installer Type `runfile (local)`.

Afterwards, installation of CUDA on Fedora 23 workstation had been nontrivial. Go see either my github repository ML-grabbag (which will be updated) or my wordpress blog (which may not be upgraded frequently).

$P = VI = I^2R$ heating.

2.2. (faster) clock speed, instruction level parallelism per clock cycle - Digging Holes, make Computers Run Faster, Chickens or Oxen.

cf. 4. Digging Holes, Around minute 1:41; *Methods for Building a faster processor*

- *Faster clock speed.* Faster clock: let T = time period for single computation $\equiv T(1)$.
  Thus
  $$f = \frac{1}{T} = \frac{1}{T(1)} \qquad \text{(frequency)}$$
  Smaller $T(1)$ increases power consumption.
- **instruction level parallelism per clock cycle** $\sim$ more work per step.
  $$v = \frac{d}{t}$$
  For $t = 1$, how much work d gets done in this "clock cycle?"

In summary (in other words),

Using $vt = d$, consider faster **clock speed**; and so $T(1)$ smaller, and so $v(1) = \frac{d}{T(1)} = \frac{1}{T(1)}$ = clock speed is "bigger" (faster).

This is at the expense of power consumption.

instruction level parallelism per clock cycle (more transistors). Looking at $v = \frac{d}{t}$,

let $t = T(1)$, time period for 1 "cycle",

increase $d = d(1)$, work (instructions) done in 1 cycle. $v = \frac{d}{T(1)}$, d bigger, so v "bigger" (faster) for fixed $T(1)$.

5. Quiz: How to Make Computers Run Faster

threads - "parallel pieces of work on the GPU"

6. Chickens or Oxen?

8. Quiz: How are CPUs Getting Faster? We have more transistors available per computation (i.e. *instruction level parallelism per clock cycle*).

9. Why we Cannot keep increasing clock speed? Heat, power! Can't make processors faster and faster.

10. What kind of Processors are we Building Assuming the major design constraint is power, traditionally for CPUs, CPUs have complex control hardware, allowing for more flexibility and performance, but is expensive in terms of power. GPUs have a simple control hardware, devoting more transistors to computation; its simple units are potentially more power efficient (operations/watt).

2.3. Definitions of Latency and throughput (or bandwidth).

cf. 12. Building a Power Efficient Processor

We can seek to ***minimize*** **latency**.

For a set of instructions := process, or instructions, the time interval between instruction(s) initiation to completion, or amount of time to complete a task T is latency

Definition 1 (latency)**.** ***latency*** $= T$

Definition 2 (throughput)**.** *throughput - tasks completed per unit time,*

$$(1) \qquad = d/T$$

with T fixed.

EY : 20170601: is throughput = bandwidth (???)

CPUs optimize for latency $T(1)$ (minimize latency $T(1)$).

GPUs optimize for throughput $\frac{d}{T_1}$, T_1 fixed (some unit time). (maximize throughput $\frac{d}{T_1}$, T_1 fixed).

bandwidth := bit rate of available or consumed information capacity (bits per second) $v = \frac{d}{t}$.

EY: 20170601 so throughput and bandwidth defined similarly, but *are they the same notion? Same thing???*

13. Quiz, Latency vs Bandwidth

latency [sec]. From the title "Latency vs. bandwidth", I'm thinking that throughput = bandwidth (???). throughput = job/time (of job).

Given total task, velocity v,

total task $/v$ = latency. throughput = latency/(jobs per total task).

Also, in Building a Power Efficient Processor. Owens recommends the article David Patterson, "Latency..."

2.3.1. *Core GPU Design Tenets.*

14. Core GPU Design Tenets

(1) GPUs have lots of simple compute units, more compute power for simpler control complexity (tradeoff)
(2) Explicitly parallel programming model
(3) optimize for throughput, not latency

cf. GPU from the Point of View of the Developer

$n_{\text{core}} \equiv$ number of cores

$n_{\text{vecop}} \equiv$ ($n_{\text{vecop}}-$wide axial vector operations/*core* core)

$n_{\text{thread}} \equiv$ threads/core (hyperthreading)

$$n_{\text{core}} \cdot n_{\text{vecop}} \cdot n_{\text{thread}} \text{ parallelism}$$

There were various websites that I looked up to try to find out the capabilities of my video card, but so far, I've only found these commands (and I'll print out the resulting output):

```
$ lspci -vnn | grep VGA -A 12
03:00.0 VGA compatible controller [0300]: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] [10de:17c8] (rev a1) (prog-if 00 [VG
        Subsystem: eVga.com. Corp. Device [3842:3994]
        Physical Slot: 4
        Flags: bus master, fast devsel, latency 0, IRQ 50
        Memory at fa000000 (32-bit, non-prefetchable) [size=16M]
        Memory at e0000000 (64-bit, prefetchable) [size=256M]
        Memory at f0000000 (64-bit, prefetchable) [size=32M]
        I/O ports at e000 [size=128]
        [virtual] Expansion ROM at fb000000 [disabled] [size=512K]
        Capabilities: <access denied>
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia

$ lspci | grep VGA -E
03:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] (rev a1)

$ grep driver /var/log/Xorg.0.log
[    18.074] Kernel command line: BOOT_IMAGE=/vmlinuz-4.2.3-300.fc23.x86_64 root=/dev/mapper/fedora-root ro rd.lvm.lv=fedora
[    18.087] (WW) Hotplugging is on, devices using drivers 'kbd', 'mouse' or 'vmmouse' will be disabled.
[    18.087]    X.Org XInput driver : 22.1
[    18.192] (II) Loading /usr/lib64/xorg/modules/drivers/nvidia_drv.so
[    19.088] (II) NVIDIA(GPU-0): Found DRM driver nvidia-drm (20150116)
[    19.102] (II) NVIDIA(0):     ACPI event daemon is available, the NVIDIA X driver will
[    19.174] (II) NVIDIA(0): [DRI2]    VDPAU driver: nvidia
[    19.284]    ABI class: X.Org XInput driver, version 22.1
...
```

```
$ lspci -k | grep -A 8 VGA
03:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX 980 Ti] (rev a1)
        Subsystem: eVga.com. Corp. Device 3994
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia
03:00.1 Audio device: NVIDIA Corporation GM200 High Definition Audio (rev a1)
        Subsystem: eVga.com. Corp. Device 3994
        Kernel driver in use: snd_hda_intel
        Kernel modules: snd_hda_intel
05:00.0 USB controller: VIA Technologies, Inc. VL805 USB 3.0 Host Controller (rev 01)
```

CUDA Program Diagram

$$\begin{tikzcd}
& \text{CUDA program in C with extensions} \arrow[dl] \arrow[dr] & \\
\text{CPU ``Host''} \arrow[rr, "\text{coprocessor}"] \arrow[d] & & \text{GPU ``Device''} \arrow[loop right, "4"] \\
\text{Memory} \arrow[rr, "1"] & & \text{Memory} \arrow[loop right, "3"] \arrow[ll, bend left, "2"]
\end{tikzcd}$$

CPU "host" is the boss (and issues commands) -Owen.

$\text{Coprocessor}: \text{CPU ``host''} \to \text{GPU ``device''}$

$\text{Coprocessor}: \text{CPU process} \mapsto \text{(co)-process out to GPU}$

With

1 data cpu → gpu
2 data gpu → cpu (initiated by cpu host)

1., 2., uses `cudaMemcpy`
3 allocate GPU memory: `cudaMalloc`
4 launch kernel on GPU

Remember that for 4., this launching of the kernel, while it's acting on GPU "device" onto itself, it's initiated by the boss, the CPU "host".

Hence, cf. Quiz: What Can GPU Do in CUDA, GPUs can respond to CPU request to receive and send Data CPU → GPU and Data GPU → CPU, respectively (1,2, respectively), and compute a kernel launched by the CPU (3).

A CUDA Program A typical GPU program

- `cudaMalloc` - CPU allocates storage on GPU
- `cudaMemcpy` - CPU copies input data from CPU → GPU
- *kernel launch* - CPU launches kernel(s) on GPU to process the data
- `cudaMemcpy` - CPU copies results back to CPU from GPU

Owens advises minimizing "communication" as much as possible (e.g. the `cudaMemcpy` between CPU and GPU), and do a lot of computation in the CPU and GPU, each separately.

Defining the GPU Computation

Owens circled this

BIG IDEA $\boxed{\text{This is Important}}$

Kernels look like serial programs

Write your program as if it will run on **one** thread

The GPU will run that program on **many** threads

Squaring A Number on the CPU

Note

(1) Only 1 thread of execution: ("thread" := one independent path of execution through the code) e.g. the `for` loop
(2) no explicit parallelism; it's serial code e.g. the `for` loop through 64 elements in an array

GPU Code A High Level View

CPU:

- Allocate Memory
- Copy Data to/from GPU
- Launch Kernel - species degree of parallelism

GPU:

- Express Out = In · In - says *nothing* about the degree of parallelism

Owens reiterates that in the GPU, everything looks serial, but it's only in the CPU that anything parallel is specified.

pseudocode: CPU code: square kernel <<< 64 >>> (outArray,inArray)

Squaring Numbers Using CUDA Part 3

From the example

```
// launch the kernel
square<<<1, ARRAY_SIZE>>>(d_out, d_in)
```

we're introduced to the "CUDA launch operator", initiating a kernel of 1 block of 64 elements (`ARRAY_SIZE` is 64) on the GPU. Remember that `d_` prefix (this is naming comvention) tells us it's on the device, the GPU, solely.

With CUDA launch operator ≡<<<>>>, then also looking at this explanation on `stackexchange` (so surely others are confused as well, of those who are learning this (cf. CUDA kernel launch parameters explained right?). From Eric's answer,

threads are grouped into blocks. all the threads will execute the invoked kernel function.

Certainly,

$$<<<>>>: (n_{\text{block}}, n_{\text{threads}}) \times \text{kernelfunctions} \mapsto \text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>> \in \text{End} : \text{Dat}_{\text{GPU}}$$

$$<<<>>>: \mathbb{N}^+ \times \mathbb{N}^+ \times \text{Mor}_{\text{GPU}} \to \text{EndDat}_{\text{GPU}}$$

where I propose that GPU can be modeled as a category containing objects Dat_{GPU}, the collection of all possible data inputs and outputs into the GPU, and Mor_{GPU}, the collection of all kernel functions that run (exclusively, and this *must* be the class, as reiterated by Prof. Owen) on the GPU.

Next,

$$\text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>>: \text{din} \mapsto \text{dout} \qquad \text{(as given in the ``square'' example, and so I propose)}$$

$$\text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>>: (\mathbb{N}^+)^{n_{\text{threads}}} \to (\mathbb{N}^+)^{n_{\text{threads}}}$$

But keep in mind that dout, din are pointers in the C program, pointers to the place in the memory.

`cudaMemcpy` is a functor category, s.t. e.g. $\text{Obj}_{\text{CudaMemcopy}} \ni \text{cudaMemcpyDevicetoHost}$ where

$$\text{cudaMemcopy}(-, -, n_{\text{thread}}, \text{cudaMemcpyDeviceToHost}) : \text{Memory}_{\text{GPU}} \to \text{Memory}_{\text{CPU}} \in \text{Hom}(\text{Memory}_{\text{GPU}}, \text{Memory}_{\text{CPU}})$$

Squaring Numbers Using CUDA 4

Note the C language construct *declaration specifier* - denotes that this is a kernel (for the GPU) and not CPU code. Pointers need to be allocated on the GPU (otherwise your program will crash spectacularly -Prof. Owen).

2.3.2. *What are C pointers?* Is ⟨ type ⟩∗, a pointer, then a mapping from the category, namely the objects of types, to a mapping from the specified value type to a memory address?

e.g.

$$\langle\,\rangle * : \text{float} \mapsto \text{float} *$$
$$\text{float } * : \text{din} \mapsto \text{ some memory address}$$

and then we pass in mappings, not values, and so we're actually declaring a square *functor*.

What is `threadIdx`? What is it mathematically? Consider that ∃ 3 "modules":

$$\begin{gathered} \text{threadIdx}.x \\ \text{threadIdx}.y \\ \text{threadIdx}.z \end{gathered}$$

And then the line

```
int idx = threadIdx.x;
```

says that idx is an integer, "declares" it to be so, and then assigns idx to threadIdx.x which surely has to also have the same type, integer. So (perhaps)

$$idx \equiv \text{threadIdx}.x \in \mathbb{Z}$$

is the same thing.

Then suppose threadIdx ⊂ FinSet, a subcategory of the category of all (possible) finite sets, s.t. threadIdx has 3 particular morphisms, $x, y, z \in \text{Mor}threadIdx$,

$$\begin{aligned} x &: \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}} \\ y &: \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}} \\ z &: \text{threadIdx} \mapsto \text{threadIdx}.x \in \text{Obj}_{\text{FinSet}} \end{aligned}$$

Configuring the Kernel Launch Parameters Part 1

$n_{\text{blocks}}, n_{\text{threads}}$ with $n_{\text{threads}} \geq 1024$ (this maximum constant is GPU dependent). You should pick the $(n_{\text{blocks}}, n_{\text{threads}})$ that makes sense for your problem, says Prof. Owen.

2.3.3. *More thoughts on Squaring Numbers Using CPU, and then using CUDA.* Note that this squaring of numbers is really element-wise multiplication of a vector.

I sought an isomorphism between abstract algebra and computer code.

Consider

$$\begin{aligned} \mathbb{R}^N \ni x & \qquad N \in \mathbb{Z}^+ \\ \mathbb{R} \ni x[i] & \qquad i = 1 \dots N \to i = 0, \dots N-1 \end{aligned}$$

Then the element-wise squaring of numbers is

$$(x[i])^2 = x[i] \cdot x[i]$$

In general,

$$(x[i])^p = \underbrace{x[i] \cdot x[i] \dots x[i]}_{p \text{ times}}$$

2.3.4. *Memory layout of blocks and threads.* $\forall (n_{\text{blocks}}, n_{\text{threads}}) \in \mathbb{Z} \times \{1 \dots 1024\}$, $\{1 \dots n_{\text{block}} \times \{1 \dots n_{\text{threads}}\}$ is now an ordered index (with lexicographical ordering). This is just 1-dimensional (so possibly there's a 1-to-1 mapping to a finite subset of $\mathbb{Z}$).

I propose that "adding another dimension" or the 2-dimension, that Prof. Owen mentions is being able to do the Cartesian product, up to 3 Cartesian products, of the block-thread index.

Quiz: Configuring the Kernel Launch Parameters 2

Most general syntax:

Configuring the kernel launhc

```
kernel<<<grid of blocks, block of threads >>>(...)

// for example

square<<<dim3(bx,by,bz), dim3(tx,ty,tz), shmem>>>(...)
```

where `dim3(tx,ty,tz)` is the grid of blocks $bx \cdot by \cdot bz$

`{dim3}(tx,ty,tz)` is the block of threads $tx \cdot ty \cdot tz$

`shmem` is the shared memory per block in bytes

Quiz: Map

I wanted to try to mathematically formulate the idea of `map`.

$$\texttt{MAP(ELEMENTS,FUNCTION)} \Longleftrightarrow \begin{gathered} \text{given } x \in \mathbb{R}^N \\ x[i] \xrightarrow{f} f(x[i]) \\ \text{or} \\ \text{set of elements (finite, so can be indexed)} \qquad \{x_0, \dots, x_{n-1}\}_{\mathcal{A}} \in \text{ObjFin} \\ x_i \xrightarrow{f} f(x_i), \qquad \forall\, i \in \mathcal{A} \end{gathered}$$

Problem Set 1 "Also, the image is represented as an 1D array in the kernel, not a 2D array like I mentioned in the video."

Here's part of that code for squaring numbers:

```
__global__ void square(float *d_out, float *d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f*f;
  }
```

2.3.5. *Problem Set 1, Udacity CS344.* Let

$$\begin{aligned} L_x &\equiv \text{ total number of pixels in } x\text{-direction of image } \in \mathbb{Z}^+ \\ L_y &\equiv \text{ total number of pixels in } y\text{-direction of image } \in \mathbb{Z}^+ \end{aligned}$$

and so $L_x L_y$ = total number of pixels in image.

The formula for ensuring that all threads will be computed, given an arbitrary choice of the number of threads in a (single) block, is the following:

$$\begin{aligned} \frac{L_x + (M_x - 1)}{M_x} &= N_x \in \mathbb{N} & N_x &= \text{ number of (thread) blocks in } x\text{-direction} \\ \frac{L_y + (M_y - 1)}{M_y} &= N_y \in \mathbb{N} & N_y &= \text{ number of (thread) blocks in } y\text{-direction} \end{aligned}$$

Then

$$(M_x, M_y, 1) \in \mathbb{N}^3 \Longleftrightarrow \texttt{dim3}$$

needs to be determined manually, empirically, and in consideration of the actual GPU hardware architecture (look up number of CUDA cores, and allowed maximum threads), where

$$\begin{aligned} M_x &\equiv \text{ number of threads per block in } x\text{-direction} \\ M_y &\equiv \text{ number of threads per block in } y\text{-direction} \end{aligned}$$

Consider that we want to go from the indices on each thread per block, on each block on the grid, in each of the 2 dimensions, to a global 2-dimensional position, and then "flatten" these coordinates to a 1-dimensional array that CUDA C can load onto global memory. In other words, for

$$\begin{aligned} i_x &\in \{0,\dots,M_x-1\} && \Longleftrightarrow \texttt{threadIdx.x} \\ i_y &\in \{0,\dots,M_y-1\} && \Longleftrightarrow \texttt{threadIdx.y} \\ j_x &\in \{0,\dots,N_x-1\} && \Longleftrightarrow \texttt{blockIdx.x} \\ j_y &\in \{0,\dots,N_y-1\} && \Longleftrightarrow \texttt{blockIdx.y} \end{aligned}$$

and so for

$$\begin{gathered} (k_x,k_y) \\ k_X = i_x + j_x M_x \\ k_y = i_y + j_y M_y \end{gathered}$$

then we sought the following operations:

$$\begin{aligned} (j_x,j_y)\times(i_x,i_y) &\in \{0,\dots,N_x-1\}\times\{0\dots N_y-1\}\times\{0\dots M_x-1\}\times\{0\dots M_y-1\}\in \texttt{dim3}\times\texttt{dim3} \\ &\mapsto (k_x,k_y)\in\{0\dots L_x-1\}\times\{0\dots L-y-1\} \\ &\mapsto k = k_x + L_x k_y \in \{0\dots L_xL_y-1\} \end{aligned}$$

2.3.6. *Grid of blocks, block of threads, thread that's indexed; (mathematical) structure of it all.* Let

$$\text{grid} = \prod_{I=1}^N (\text{block})^{n_I^{\text{block}}}$$

where $N = 1,2,3$ (for CUDA) and by naming convention
$$\begin{aligned} I &= 1 \equiv x \\ I &= 2 \equiv y \\ I &= 3 \equiv z \end{aligned}$$

Let's try to make it explicitly (as others had difficulty understanding the grid, block, thread model, cf. colored image to greyscale image using CUDA parallel processing, Cuda gridDim and blockDim) through commutative diagrams and categories (from math):

$$\begin{tikzcd} \prod_{I=1}^N \mathbb{Z}^+ \arrow[d, "\text{dim3}"] & \ni (N_x^{\text{blocks}}, N_y^{\text{blocks}}, N_z^{\text{blocks}}) \arrow[d, mapsto, "\text{dim3}"'] \\ \text{grid} \arrow[u, bend left, "\text{gridDim}"] & \ni \text{gridSize}(N_x^{\text{blocks}}, N_y^{\text{blocks}}, N_z^{\text{blocks}}) \arrow[u, mapsto, bend right, "(\text{gridDim}.x,\ \text{gridDim}.y,\ \text{gridDim}.z)"'] \end{tikzcd}$$

$$\begin{tikzcd} \text{grid} \arrow[d, "\text{blockIdx}"] & \ni \texttt{d\_rgbaImage} \arrow[d, mapsto, "(\text{blockIdx}.x,\text{blockIdx}.y,\text{blockIdx}.z)"] \\ \prod_{I=1}^N \mathbb{Z} \supset \prod_{I=1}^N \{1\dots N_I^{\text{blocks}}\} & \ni (i^{\text{blocks}}, j^{\text{blocks}}, k^{\text{blocks}}) \end{tikzcd}$$

and then similar relations (i.e. arrows, i.e. relations) go for a block of threads:

$$\begin{tikzcd} \prod_{I=1}^N \mathbb{Z}^+ \arrow[d, "\text{dim3}"] & \ni (N_x^{\text{threads}}, N_y^{\text{threads}}, N_z^{\text{threads}}) \arrow[d, mapsto, "\text{dim3}"'] \\ \text{block} \arrow[u, bend left, "\text{blockDim}"] & \ni \text{blockSize}(N_x^{\text{threads}}, N_y^{\text{threads}}, N_z^{\text{threads}}) \arrow[u, mapsto, bend right, "(\text{blockDim}.x,\ \text{blockDim}.y,\ \text{blockDim}.z)"'] \end{tikzcd}$$

$$\begin{tikzcd} \text{block} \arrow[d, "\text{threadIdx}"] & \ni \text{block} \arrow[d, mapsto, "(\text{threadIdx}.x,\text{threadIdx}.y,\text{threadIdx}.z)"] \\ \prod_{I=1}^N \mathbb{Z} \supset \prod_{I=1}^N \{1\dots N_I^{\text{threads}}\} & \ni (i^{\text{threads}}, j^{\text{threads}}, k^{\text{threads}}) \end{tikzcd}$$

gridsize help assignment 1 Pp explains how threads per block is variable, and remember how Owens said Luebki says that a GPU doesn't get up for more than a 1000 threads per block.

2.3.7. *Generalizing the model of an image.* Consider vector space V, e.g. $\dim V = 4$, vector space V over field $\mathbb{K}$, so $V = \mathbb{K}^{\dim V}$.

Each pixel represented by $\forall\, v \in V$.

Consider an image, or space, M. $\dim M = 2$ (image), $\dim M = 3$. Consider a local chart (that happens to be global in our case):

$$\begin{aligned} &\varphi : M \to \mathbb{Z}^{\dim M} \supset \{1\dots N_1\}\times\{1\dots N_2\}\times\dots\times\{1\dots N_{\dim M}\} \\ &\varphi : x \mapsto (x^1(x), x^2(x), \dots, x^{\dim M}(x)) \end{aligned}$$

$$\begin{tikzcd} E \arrow[r, "\varphi"] \arrow[d, "\pi"'] & M \times V \arrow[dl] \\ M & \end{tikzcd} \qquad \begin{tikzcd} E \arrow[r, "\varphi"] \arrow[d, "\pi"'] & \text{grid} \times \text{ block of threads} \arrow[dl] \\ \text{grid} & \end{tikzcd}$$

Consider a "coarsing" of underlying M:

$$\begin{tikzcd} M \times V \arrow[r, "\text{proj}"] \arrow[d, "\pi"'] & \text{proj}(M)\times\text{proj}(V) \arrow[d, "\text{proj}(\pi)"] \\ M = \{1\dots N_1\}\times\{1\dots N_2\}\times\dots\times\{1\dots N_{\dim M}\} \arrow[r, "\text{proj}"] & \text{proj}(M) = \{1\dots \frac{N_1}{N_1^{\text{threads}}}\}\times\{1\dots\frac{N_2}{N_2^{\text{threads}}}\}\times\dots\times\{1\dots\frac{N_{\dim M}}{N_{\dim M}^{\text{threads}}}\} \end{tikzcd}$$

e.g. $N_1^{\text{thread}} = 12$
$N_2^{\text{thread}} = 12$

Just note that in terms of syntax, you have the "block" model, in which you allocate blocks along each dimension. So in

$$\begin{gathered} \textit{const dim3 blockSize}(n_x^b, n_y^b, n_z^b) \\ \textit{const dim3 gridSize}(n_x^{\text{gr}}, n_y^{\text{gr}}, n_z^{\text{gr}}) \end{gathered}$$

Then the condition is $n_x^b/\dim V, n_y^b/\dim V, n_z^b/\dim V \in \mathbb{Z}$ (condition), $\quad (n_x^{\text{gr}}-1)/\dim V, n_y^{\text{gr}}/\dim V, n_z^{\text{gr}}/\dim V \in \mathbb{Z}$

2.4. Unit 2, Lesson 2 GPU Hardware and Parallel Communication Patterns.

Transpose Part 1

Now

$$\mathrm{Mat}_{\mathbb{F}}(n,n) \xrightarrow{T} \mathrm{Mat}_{\mathbb{F}}(n,n)$$
$$A \mapsto A^T \text{ s.t. } (A^T)_{ij} = A_{ji}$$
$$\mathrm{Mat}_{\mathbb{F}} \xrightarrow{T} \mathbb{F}^{n^2}$$
$$A_{ij} \mapsto A_{ij} = A_{in+j}$$

$$\begin{tikzcd}
\mathrm{Mat}_{\mathbb{F}}(n,n) \arrow[r] \arrow[d, "T"'] & \mathbb{F}^{n^2} \arrow[d, "T"] \\
\mathrm{Mat}_{\mathbb{F}}(n,n) \arrow[r] & \mathbb{F}^{n^2}
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{ij} \arrow[r, mapsto] \arrow[d, mapsto, "T"'] & A_{in+j} \arrow[d, mapsto, "T"] \\
(A^T)_{ij} = A_{ji} \arrow[r, mapsto] & A_{jn+i}
\end{tikzcd}$$

Transpose Part 2

Possibly, transpose is a functor.

Consider struct as a category. In this special case, Objstruct = {arrays} (a struct of arrays). Now this struct already has a hash table for indexing upon declaration (i.e. "creation"): so this category struct will need to be equipped with a "diagram" from the category of indices J to struct: $J \to \text{struct}$.

So possibly

$$\begin{aligned}
\text{struct} &\xrightarrow{T} \quad \text{array} \\
\text{ObjStruct} = \{ \text{ arrays } \} &\xrightarrow{T} \text{Objarray} = \{ \text{ struct } \} \\
J \to \text{ struct } &\xrightarrow{T} \quad J \to \text{ array}
\end{aligned}$$

Quiz: What Kind Of Communication Pattern This quiz made a few points that clarified the characteristics of these so-called communication patterns (amongst the memory?)

- map is bijective, and map : Idx → Idx
- gather - not necessarily surjective
- scatter - not necessarily surjective
- stencil - surjective
- transpose (see before)

Parallel Communication Patterns Recap

- map - bijective
- transpose - bijective
- gather - not necessarily surjective, and is many-to-one (by def.)
- scatter - one-to-many (by def.) and is not necessarily surjective
- stencil - several-to-one (not injective, by definition), and is surjective
- reduce - all-to-one
- scan/sort - all-to-all

Programmer View of the GPU

thread blocks: group of threads that cooperate to solve a (sub)problem

Thread Blocks And GPU Hardware

CUDA GPU is a bunch of SMs:

Streaming Multiprocessors (SM)s

SMs have a bunch of simple processors and memory.

Dr. Luebki:

$$\boxed{\begin{array}{c}\text{Let me say that again because it's really important} \\ \text{GPU is responsible for allocating blocks to SMs}\end{array}}$$

Programmer only gives GPU a pile of blocks.

Quiz: What Can The Programmer Specify

I myself thought this was a revelation and was not intuitive at first:

Given a single kernel that's launched on many thread blocks include X, Y, the programmer cannot specify the sequence the blocks, e.g. block X, block Y, run (same time, or run one after the other), and which SM the block will run on (GPU does all this).

Quiz: A Thread Block Programming Example

Open up `hello blockIdx.cu` in Lesson 2 Code Snippets (I got the repository from github, repo name is cs344).

At first, I thought you can do a single file compile and run in Eclipse without creating a new project. No. cf. Eclipse creating projects every time to run a single file?.

I ended up creating a new CUDA C/C++ project from File -¿ New project, and then chose project type Executable, Empty Project, making sure to include Toolchain CUDA Toolkit (my version is 7.5), and chose an arbitrary project name (I chose cs344single). Then, as suggested by Kenny Nguyen, I dragged and dropped files into the folder, from my file directory program.

I ran the program with the "Play" triangle button, clicking on the green triangle button, and it ran as expected. I also turned off Build Automatically by deselecting the option (no checkmark).

GPU Memory Model

$$\begin{tikzcd}
\text{thread} & \text{local memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"]
\end{tikzcd}$$

Then consider threadblock ≡ thread block

$$\text{Objthreadblock} \supset \{ \text{ threads } \}$$
$$\text{FinSet} \xrightarrow{\text{threadIdx}} \text{thread } \in \text{Morthreadblock}$$

$$\begin{tikzcd}
\text{threadblock} & \text{shared memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"]
\end{tikzcd}$$

$\forall$ thread,

$$\begin{tikzcd}
\text{thread} & \text{global memory} \arrow[l, "\text{read}"'] \arrow[loop right, "\text{write}"]
\end{tikzcd}$$

Synchronization - Barrier

Danger: what if a thread reads a result before another thread writes it?

Threads need to *synchronize*.

one of the most fundamental problems in parallel computing

Quiz: The Need For Barriers

3 barriers were needed (wasn't obvious to me at first). All threads need to finish the write, or initialization, so it'll need a barrier.

While

```
array[idx] = array[idx+1];
```

is 1 line, it'll actually need 2 barriers; first read. Then write.

So *actually* we'll need to *rewrite* this code:

```
int temp = array[idx+1];
__syncthreads();
array[idx] = temp;
```

```
__syncthreads();
```

Make sure each *read* and *write* operation is completed.
kernels have implicit barrier for each.
Writing Efficient Programs

(1) Maximize *arithmetic intensity* arithmetic intensity := $\frac{\text{math}}{\text{memory}}$

video: Minimize Time Spent On Memory
local memory is fastest; global memory is slower

$$\text{local} > \text{ shared} >> \text{global} >> \text{CPU}$$

kernel we know (in the code) is tagged with `__global__`

2.4.1. *Coalesce global memory accesses.* 31. Coalesce Memory Access, from Unit 2/Lesson 2 - GPU Hardware and Parallel Communication Patterns

Whenever a thread on the GPU reads or writes global memory, it always acceses a large chunk of memory at once.
Even if the thread needs to only access a smaller subset of that large chunk.
If other threads are making similar memory access, the GPU can exploit that and reuse that larger chunk.
We saw such access pattern is coalesced; GPU must efficient when threads read or write contiguous memory locations.
quiz: A Quiz on Coalescing Memory Access
Work it out as Dr. Luebki did to figure out if it's coalesced memory access or not.
Atomic Memory Operations
Atomic Memory Operations
atomicadd atomicmin atomicXOR atomicCAS Compare And Swap

2.4.2. *On Problem Set 2.* There is what I call the "naive global memory" scheme, that solves the objective of blurring a photo with a local stencil of the values, using only global memory on the GPU.

Given image of size $L_x \times L_y$, i.e. $(L_x, L_y) \in (\mathbb{Z}^+)^2$; image is really a designated or particular mapping f,

$$\begin{gathered} f : \{0 \dots L_x - 1\} \times \{0 \dots L_y - 1\} \to \{0 \dots 255\}^4 \\ f(x,y) = (f^{(r)}(x,y), f^{(b)}(x,y), f^{(g)}(x,y), f^{(\alpha)}(x,y)) \end{gathered}$$

Consider "naive global memory scheme" - establishing the following notation:

$$\begin{aligned} i_x \in \{0 \dots M_x - 1\} &\Longleftrightarrow \texttt{threadIdx.x} \\ i_y \in \{0 \dots M_y - 1\} &\Longleftrightarrow \texttt{threadIdx.y} \\ j_x \in \{0 \dots N_x - 1\} &\Longleftrightarrow \texttt{blockIdx.x} \\ j_y \in \{0 \dots N_y - 1\} &\Longleftrightarrow \texttt{blockIdx.y} \\ M_x \in \{1 \dots 1024\} &\Longleftrightarrow \texttt{blockDim.x} \\ M_y \in \{1 \dots 1024\} &\Longleftrightarrow \texttt{blockDim.y} \end{aligned}$$

with

$$\begin{gathered} N_x := (L_x + M_x - 1)/M_x \in \mathbb{Z}^+ \\ N_y := (L_y + M_y - 1)/M_y \in \mathbb{Z}^+ \end{gathered}$$

There should be a functor called "flatten" such that we end up with the image as a 1-dimensional, contiguous array on the global memory of the GPU; so for

$$k = k_x + k_y L_x \in \{0 \dots L_x L_y - 1\}$$

then

$$\begin{gathered} (k_x, k_y) \Longleftrightarrow (x,y) \in \{0 \dots L_x - 1\} \times \{0 \dots L_y - 1\} \xmapsto{\text{flatten}} k \in \{0 \dots L_x L_y - 1\} \\ f : \{0 \dots L_x - 1\} \times \{0 \dots L_y - 1\} \xrightarrow{\text{flatten}} f : \{0 \dots L_x L_y - 1\} \to \{0 \dots 255\}^4 \\ f(x,y) = f(k_x, k_y) \xmapsto{\text{flatten}} f(k) \end{gathered}$$

Then there should be a functor called "separateChannels" to represent the `__global__` kernel `separateChannels`.

$$\begin{gathered} f : \{0 \dots L_x L_y - 1\} \to \{0 \dots 255\}^4 \xrightarrow{\text{separateChannels}} f^{(c)} : \{0 \dots L_x L_y - 1\} \to \{0 \dots 255\}, \, c = \{r, g, b\} \\ f(k) \xmapsto{\text{separateChannels}} f^{(c)}(k) \end{gathered}$$

Then consider a "stencil" of size `filterWidth` × `filterWidth` $\Longleftrightarrow W \times W \in (\mathbb{Z}^+)^2$.
Let $(\nu_x, \nu_y) \in \{0 \dots W - 1\}^2$ and so

$$\left(\nu_x - \frac{W}{2}, \nu_y - \frac{W}{2} \right) \in \{ \frac{-W}{2}, \dots \frac{W}{2} - 1 \} \subset \mathbb{Z}$$

Now let

$$\begin{aligned} k_x^{\text{st}} &= k_x + \nu_x - \frac{W}{2} \Longleftrightarrow \texttt{stencilindex\_x} \\ k_y^{\text{st}} &= k_y + \nu_y - \frac{W}{2} \Longleftrightarrow \texttt{stencilindex\_y} \end{aligned}$$

with $k_x^{\text{st}} \in \{0 \dots L_x - 1\}$
$k_y^{\text{st}} \in \{0 \dots L_y - 1\}$

We also have to apply the flatten functor on the stencil:

$$(\nu_x, \nu_y) \in \{0 \dots W - 1\}^2 \xmapsto{\text{flatten}} \nu = \nu_x + W \nu_y \in \{0 \dots W^2 - 1\}$$

And so the gist of the blurring operation is in this equation:

$$g^{(c)}(k) = \sum_{\nu_x = 0}^{W-1} \sum_{\nu_y = 0}^{W-1} c_{\nu = \nu_x + W\nu_y} f^{(c)}(k_x^{\text{st}} + L_x \cdot k_y^{\text{st}}) \qquad \forall \, c = \{r, g, b\} \tag{2}$$

with $k_x^{\text{st}} = k_x^{\text{st}}(\nu_x) := k_x + \nu_x - \frac{W}{2}$
$k_y^{\text{st}} = k_y^{\text{st}}(\nu_y) := k_y + \nu_y - \frac{W}{2}$

2.4.3. *Problem Set 2, shared memory "tiling" scheme.* I think the `__shared__` memory "tiling" scheme is non-trivial due to accounting for the values "at the edges" of the thread block, including the "corners" the so-called "halo" cells. Storing the value of the "cells" or threads within a thread block into shared memory is *relatively* straightforward - it is a 1-to-1 mapping. But taking care of the corner cases, due to the desired "stencil" for blurring, is nontrivial, I think.

Consider my scheme for "tiling" using shared memory:

Let

$$\begin{gathered} k_x = i_x + j_x M_x \in \{0 \dots L_x - 1\} \\ k_y = i_y + j_y M_y \in \{0 \dots L_y - 1\} \\ k_x < L_x \text{ and } k_y < L_y \\ 0 \leq k_x < L_x \text{ and } 0 \leq k_y < L_y \\ k := k_x + L_x k_y \end{gathered}$$

and let

$$\begin{gathered} S_x := M_x + 2r \\ S_y := M_y + 2r \\ s_x := i_x + r \\ s_y := i_y + r \\ 0 \leq s_x < S_x \text{ and } 0 \leq s_y < S_y \\ s_k := s_x + S_x s_y \end{gathered}$$

where r is the "radius" or essentially the stencil size, out in 1-direction.

Loading the regular cells,

$$s_{\text{in}}[s_k] = f^{(c)}(k)$$

Loading the halo cells,

if $(i_x < r)$,
then requiring

$$\begin{gathered} 0 \leq s_x - r < S_x \qquad 0 \leq k_x - r < L_x \\ 0 \leq s_y < S_y \qquad 0 \leq k_y < L_y \\ s_{\text{in}}[s_x - r + S_x s_y] = f^{(c)}[k_x - r + L_x k_y] \\ 0 \leq s_x + M_x < S_x \qquad 0 \leq k_x + M_x < L_x \\ 0 \leq s_y < S_y \qquad 0 \leq k_y < L_y \\ s_{\text{in}}[s_x + M_x + S_x s_y] = f^{(c)}[k_x + M_x + L_x k_y] \end{gathered}$$

If $(i_y < r)$,
then requiring

$$\begin{gathered} 0 \leq s_x < S_x \qquad 0 \leq k_x < L_x \\ 0 \leq s_y - r < S_y \qquad 0 \leq k_y - r < L_y \\ s_{\text{in}}[s_x + S_x(s_y - r)] = f^{(c)}[k_x + L_x(k_y - r)] \\ 0 \leq s_x < S_x \qquad 0 \leq k_x < L_x \\ 0 \leq s_y + M_y < S_y \qquad 0 \leq k_y + M_y < L_y \\ s_{\text{in}}[s_x + S_x(s_y + M_y)] = f^{(c)}[k_x + L_x(k_y + M_y)] \end{gathered}$$

And now the actual stencil calculation:

```
$\forall \nu_y \in \{\nu_y = 0, 1 \dots W - 1 | 0 \leq \nu_y < W\}$,
    $k_y^{\text{st}} := s_y + \nu_y - r$
    $\forall \nu_x \in \{\nu_x = 0, 1 \dots W - 1 | 0 \leq \nu_x < W\}$,
        $k_x^{\text{st}} := s_x + \nu_x - r$

        $\texttt{inputvalue} = s_{\text{in}}[k_x^{\text{st}} + S_x k_y^{\text{st}}]$ with $0 \leq k_x^{\text{st}} < S_x$
                                               $0 \leq k_y^{\text{st}} < S_y$
        $\texttt{filtervalue} = c(\nu_x + W\nu_y)$
        $\texttt{value} += \texttt{filtervalue} \cdot \texttt{inputvalue}$,
```

i.e.

$$g^{(c)}(k) = \sum_{\nu_y=0}^{W-1} \sum_{\nu_x=0}^{W-1} c_{\nu=\nu_x+W\nu_y} s_{\text{in}}[k_x^{\text{st}} + k_y^{\text{st}} S_x]$$

Unfortunately, the (literal) corner cases aren't accounted for correctly, (when $i_x < r$ *and* $i_y < r$), as can be seen by the difference image and output image when it's run.

Samuel Lin or Samuel271828 had both an elegant and *correct* implementation. It's also in Samuel Lin or samuellin3310's github repositories, in student fuction improved share.cu (sic).

Here it is, mathematically:

Let

$$\begin{gathered} s_{\text{in}} \in \mathbb{R}^{(M_x+2r)(M_y+2r)} \\ k_x = i_x + j_x M_x \in \mathbb{Z} \\ k_y = i_y + j_y M_y \in \mathbb{Z} \\ k_x < L_x \text{ and } k_y < L_y \\ 0 \leq k_x < L_x \text{ and } 0 \leq k_y < L_y \\ k := k_x + L_x k_y \end{gathered}$$

Then,

```
$\forall i \in \{i = i_x - r, i_x - r + M_x, i_x - r + 2M_x, \dots | i_x - r \leq i < M_x + r\}$,
    $\forall j \in \{j = i_y - r, i_y - r + M_y, i_y - r + 2M_y \dots | i_y - r \leq j < M_y + r\}$,
```

$$\begin{aligned} l_x &:= i + M_x j_x \in \mathbb{Z} \text{ with (enforcing) } 0 \leq l_x < L_x \\ l_y &:= j + M_y j_y \in \mathbb{Z} \text{ with (enforcing) } 0 \leq l_y < L_y \\ s_{\text{in}}&[i + r + (j + r)(M_x + 2r)] = f^{(c)}(l_x + l_y L_x) \end{aligned}$$

Enforce $k_x < L_x$ and $k_y < L_y$, otherwise nothing happens.

And now the actual stencil calculation:

```
$\forall \nu_y \in \{\nu_y = 0, 1 \dots W - 1 | 0 \leq \nu_y < W\}$,
    $k_y^{\text{st}} := s_y + \nu_y - r$
    $\forall \nu_x \in \{\nu_x = 0, 1 \dots W - 1 | 0 \leq \nu_x < W\}$,
        $k_x^{\text{st}} := s_x + \nu_x - r$

        $\texttt{inputvalue} = s_{\text{in}}[k_x^{\text{st}} + S_x k_y^{\text{st}}]$ with $0 \leq k_x^{\text{st}} < S_x$
                                               $0 \leq k_y^{\text{st}} < S_y$
        $\texttt{filtervalue} = c(\nu_x + W\nu_y)$
        $\texttt{value} += \texttt{filtervalue} \cdot \texttt{inputvalue}$,
```

i.e.

$$g^{(c)}(k) = \sum_{\nu_y=0}^{W-1} \sum_{\nu_x=0}^{W-1} c_{\nu=\nu_x+W\nu_y} s_{\text{in}}[k_x^{\text{st}} + k_y^{\text{st}} S_x]$$

It may be non-intuitive, as was in my case, from my personal experience, to have to move the requirement that $k_x < L_x$, and $k_y < L_y$, i.e. the line

```
if ( k_x >= numCols || k_y >= numRows ) {
  return; }
```

after loading all the values from global memory into shared memory, and not in the beginning of the kernel. This can be proven, in general. And again, shout-outs (i.e. credit should go) to Samuel Lin or Samuel271828 for the clarifying discussion here.

For simplicity, consider the 1-dimensional case, e.g. a 1-dimensional pixelated image, represented by a 1-dimensional array. The discussion below can easily be generalized to n-dimensions.

Recall that

$$k_x := i_x + M_x j_x$$

for

$$\begin{gathered} i_x \in \{0 \dots M_x - 1\} \\ j_x \in \{0 \dots N_x - 1\} \end{gathered}$$

for

$$
\begin{aligned}
i_x &\Longleftrightarrow \texttt{threadIdx.x} \\
j_x &\Longleftrightarrow \texttt{blockIdx.x} \\
M_x &\Longleftrightarrow \texttt{blockDim.x}
\end{aligned}
$$

with N_x determined by a formula immediately below.

Then

$$k_x \in \{0 \dots N_x M_x - 1\}$$

with N_x being determined by

$$N_x := \frac{L_x + M_x - 1}{M_x} \in \mathbb{Z}$$

By integer division, this formula for N_x guarantees that the number of blocks in the grid is the lowest number that would guarantee that *all* the needed grid points are computed, i.e. N_x is the lowest number such that there are enough (i.e. minimal number of) threads that'll compute all the needed grid points, or pixels, or computations, etc., by how integer division works. Note that

$$L_x \Longleftrightarrow \texttt{numCols} \text{ and so } N_x \Longleftrightarrow \texttt{gridDim.x}$$

Now

$$N_x M_x \geq L_x$$

meaning, that we could have the case where the very last (thread) block would have more threads than is needed to compute all the pixels.

e.g.

$$
\begin{aligned}
L_x &= 127 \\
M_x &= 128 \,(\text{ so } N_x = 1) \\
k_x &= i_x + 128 \cdot 0 = i_x \\
k_x &= 127
\end{aligned}
$$

and so $k_x \geq L_x$, namely $127 \geq 127$. If we had, in the beginning, the line that returns nothing for the condition $k_x \geq L_x$, we won't be including this case.

Now recall the shared memory tiling scheme, but in 1-dimension (for the purposes of this present discussion):

$$\forall \{ i \in i_x - r, i_x - r + M_x, i_x - r + 2M_x, \dots | i_x - r \leq i < M_x + r \}$$

so $-r \leq i < M_x + r$.

Then

$$l_x := i + M_x j_x$$

and

$$s_{\text{in}}[i + r] = f^{(c)}(l_x)$$

e.g.

$$
\begin{aligned}
r &= \texttt{filterWidth}/2 = 9/2 = 4 \\
i_x &= 127 \\
i &= 127 - 4 = 123 = l_x \\
s_{\text{in}}[127] &= f^{(c)}(123)
\end{aligned}
$$

$k_x = 127 = L_x$ so $k_x \geq L_x$

So in this very insightful example, we've seen that for $k_x = 127$, it loads the regular value at index $l_x = 123$ into the appropriate slot in the shared memory, namely $s_{\text{in}}[127]$ correctly, and yet if we placed the code line in question that tests k_x against L_x in the wrong order, this step would've been excluded!

How can we see this in general?

Consider now that $l_x = i + M_x j_x = i_x + M_x j_x - r = k_x - r$.

Also

$$i + r = i_x - r + r = i_x$$

Now then

$$s_{\text{in}}[i_x] = f^{(c)}(l_x) = f^{(c)}(k_x - r)$$

while $N_x M_x - r \geq L_x$ or $N_x M_x - r < L_x$. It's this ambiguity that forces the check of $k_x \geq L_x$ to be moved to after loading all values into shared memory. The $l_x \leq L_x$ check guarantees both that we aren't going "outside" the array indices in global memory *and* that we're "clamping" down on the absolute boundary value if we reach the absolute boundary or "end" of the array.

My big takeaway is that doing the shared memory tiling scheme is much more nuanced and deserves more inspection that the relatively straightforward "naive" global memory scheme.

2.5. Unit 3, Lesson 3 Fundamental GPU Algorithms (Reduce, Scan, Histogram; Udacity cs344).

More on Udacity forums, in particular, forum for cs344: Please elaborate the micro-optimization techniques discussed by the instructor

cf. Reduce Part 2

Reduce: Inputs.

(1) Set of elements.

Assume they are in an array.

(2) reduction operator

(a) binary

(b) associative

e.g. of binary operators, cf. Binary and Associative Operators

- multiply $*$
- min
- logical or $(a \parallel b)$
- bitwise and $(a \& b)$

Indeed, let f represent the input:

$$
\begin{aligned}
&f \in K^N; \qquad N \in \mathbb{Z}^+ \\
&f(i) \in K; \qquad i \in \{0, 1 \dots, N-1\}
\end{aligned}
$$

Serial Implementation of Reduce

$$
\begin{aligned}
&S = 0, \\
\forall\, i \in \{0, 1, \dots N-1\} &\Longleftrightarrow S = \sum_{i=0}^{N-1} f(i) \\
&S += f(i)
\end{aligned}
$$

Step Complexity of Parallel Reduce

N	steps
2	1
4	2
8	3
2^n	n

The answer is $\log_2 N = n$. Let's prove this with induction.

Consider $N = 2^{n+1}$, number of things to compute.

$\frac{N}{2} = 2^n$. After 1 step (of binary operations), there are 2^n things left to compute.

By induction step, n steps are required.

$$\log_2 N = \log_2 2^{n+1} = n + 1$$

Done.

Reduction Using Global and Shared Memory

For a good explanation of *bitwise left shift* and *bitwise right shift* operators (and also for assignment), see What does a bitwise shift (left or right) do and what is it used for?. Also, a good implementation that converts from integers to bitwise representation is given in Converting integer to a bit representation

Keep in mind these are called *bitwise shift* operations (so we could look them up by this name).

Left shift.

$$\texttt{x << y} \iff x \cdot 2^y$$

Right shift.

$$\texttt{x >> y} \iff x/2^y$$

cf. reduce.cu of Lesson 3 Code Snippet for Udacity cs344

global memory reduce. $M_x \in \mathbb{Z}^+$, e.g. $M_x = 1024$
$k_x := i_x + M_x j_x \in \mathbb{Z}^+$
$t_{id} := i_x \in \{0 \dots M_x - 1\} \subset \mathbb{Z}^+$
$\forall\, s \in \{ \frac{M_x}{2}, \frac{M_x}{2^2}, \frac{M_x}{2^3}, \dots \}$,
If $t_{id} < s$,

$$d_{in}[k_x] + = d_{in}[k_x + s] \iff d_{\text{out}}[j_x] = \sum_{\substack{t_{\text{id}} < s \\ s \in \{ \frac{M_x}{2}, \frac{M_x}{2^2}, \frac{M_x}{2^3}, \dots \}}} d_{\text{in}}[k_x + s] + d_{\text{in}}[k_x]$$

Finally,

$$d_{\text{out}}[j_x] = d_{\text{in}}[k_x]$$

shared memory reduce. $k_x := i_x + M_x j_x \in \mathbb{Z}^+$
$t_{id} := i_x \in \{0 \dots M_x - 1\} \subset \mathbb{Z}^+$
Load values into shared memory:

$$s_{\text{data}}[i_x] = d_{\text{in}}[k_x] \text{ where } s_{\text{data}} \in \mathbb{R}^{M_x}$$

$\forall\, s \in \{ \frac{M_x}{2}, \frac{M_x}{2^2}, \frac{M_x}{2^3}, \dots \}$,
If $t_{id} < s$,

$$s_{\text{data}}[t_{\text{id}}] + = s_{\text{data}}[t_{\text{id}} + s] \iff d_{\text{out}}[j_x] = \sum_{\substack{t_{\text{id}} < s \\ s \in \{ \frac{M_x}{2}, \frac{M_x}{2^2}, \frac{M_x}{2^3}, \dots \}}} s_{\text{data}}[t_{\text{id}} + s] + s_{\text{data}}[t_{\text{id}}]$$

Finally,

$$d_{\text{out}}[j_x] = s_{\text{data}}[0]$$

Scan
Inputs to Scan

Inputs to scan. Like reduce,

- input array, $\begin{matrix} f \in K^N & N \in \mathbb{Z}^+ \\ f(i) \in K & i \in \{0 \dots, N-1\} \end{matrix}$
- binary associative operator

new feature (not in reduce),

- identity element $[I \,\text{op}\, a = a\,]$

cf. What Scan Actually Does

$$f \in K^N \xmapsto{\text{scan}} g \in K^N$$

inclusive scan

$$g(i) = \bigoplus_{j=0}^{i} f(j)$$

exclusive scan

$$g(i) = \begin{cases} \bigoplus_{j=0}^{i-1} f(j) & \text{if } i \geq 1 \\ 1 & \text{if } i = 0 \end{cases}$$

These definitions are worth repeating:

Definition 3 (Scan). *Given*

- input array $f \in K^N$, *i.e.*

$$\begin{matrix} f \in K^N & N \in \mathbb{Z}^+ \\ f[i] \in K & i \in \{0, \dots, N-1\} \end{matrix}$$

with K equipped with a

- binary associative operator
- identity element *[I op a = a]*

Then for

$$f \xmapsto{\textit{scan}} g$$

(3)
$$K^N \xrightarrow{\textit{scan}} K^N$$

*with **inclusive scan** defined as*

(4)
$$\boxed{g(i) = \bigoplus_{j=0}^{i} f(j)}$$

*and **exclusive scan** defined as*

(5)
$$\boxed{g(i) = \begin{cases} \bigoplus_{j=0}^{i-1} f(j) & \textit{if } i \geq 1 \\ 1 & \textit{if } i = 0 \end{cases}}$$

cf. Hillis Steele Scan

Hillis/Steele inclusive scan. Let $\begin{matrix} f \in K^N & N \in \mathbb{Z}^+ \\ f(j) \in K & j \in \{0 \dots N-1\} \end{matrix}$
Consider step $i = 0$.
For $g_i \in K^N$, $\forall\, i \in \{0, \dots, \log_2 N - 1\}$,
Doing the first 3 steps,

$$g_0(j) = \begin{cases} f(j) + f(j - 2^0) & \text{if } j \geq 1 \\ f(j) & \text{if } j < 1 \end{cases}$$

$$g_1(j) = \begin{cases} g_0(j) + g_0(j - 2^1) & \text{if } j \geq 2^1 \\ g_0(j) & \text{if } j < 2^1 \end{cases}$$

$$g_2(j) = \begin{cases} g_1(j) + g_1(j - 2^2) & \text{if } j \geq 2^2 \\ g_1(j) & \text{if } j < 2^2 \end{cases}$$

Thus

$$g_i(j) = \begin{cases} g_{i-1}(j) + g_{i-1}(j - 2^i) & \text{if } j \geq 2^i \\ g_{i-1}(j) & \text{if } j < 2^i \end{cases}$$

Now do the following induction cases:

$$\begin{aligned} g_{\log_2 N}(0) &= f(0) \\ g_{\log_2 N}(1) &= g_0(1) = f(1) + f(0) \\ g_{\log_2 N}(2) &= g_1(2) = g_0(2) + g_0(0) = f(2) + f(1) + f(0) \\ g_{\log_2 N}(3) &= g_1(3) = g_0(3) + g_0(1) = f(3) + f(2) + f(1) + f(0) \end{aligned}$$

So generalize to the induction case:

$$g_{\log_2 N}(j) = \bigoplus_{i=0}^{j} f(i)$$

Then one would check the induction step for a number of cases, whether j was greater or smaller or equal to $2^{\log_2 N}$.

To summarize for the Hillis/Steele inclusive scan,

$$\boxed{\begin{gathered} \forall\, i \in \{0, 1, \dots, \log_2 N | 2^i < N\} \\ g_{i-1}(j) = f(j) \qquad \forall\, j \in \{0 \dots N-1\} \\ g_i(j) = \begin{cases} g_{i-1}(j) + g_{i-1}(j - 2^i) & \text{if } j \geq 2^i \\ g_{i-1}(j) & \text{if } j < 2^i \end{cases} \end{gathered}} \tag{6}$$

i.e.

$$\boxed{\begin{gathered} \forall\, i \in \{0, 1, \dots, \lfloor \log_2 N \rfloor | 2^i < N\} \\ g_{-1}(j) = f(j) \qquad \forall\, j \in \{0 \dots N-1\} \\ g_i(j) = \begin{cases} g_{i-1}(j) + g_{i-1}(j - 2^i) & \text{if } j \geq 2^i \\ g_{i-1}(j) & \text{if } j < 2^i \end{cases} \end{gathered}} \tag{7}$$

Implementation is CUDA C/C++: consider $k_x := i_x + M_x j_x \in \{0, 1, \dots, L_x\}$, $L_x \in \mathbb{Z}^+$, with the formula

$$N_x := \frac{L_x + M_x - 1}{M_x}$$

that'll give us `blockDim.x`.

$\forall\, i \in \{0, 1 \dots \log_2 k_x\}$ (or $k_x < 2^i$),

$$g_i(k_x) = g_{i-1}(k_x) + g_{i-1}(k_x - 2^i)$$

For *(inclusive) scan*,
step : $O(\log n)$
work : $O(n^2)$

For *Hillis/Steele* ,
step : $O(\log n)$
work : $O(n \log n)$

(more step efficient)
For B
cf. Inclusive Scan Revisited, Hillis Steele vs Blelloch Scan,Hillis Steele Scan,

2.6. **Blelloch scan.** cf. Blelloch Scan

2.6.1. *Blelloch scan, reduce, 1st part, up sweep.* My development: consider the most basic cases. So for $f \in K^N$,

$$\begin{array}{cc} i \in \{0, 1, \dots N-1\} & N = 8 \qquad i \in \{0, 1, \dots, 8-1 = 7\} \\ 2j, \qquad j \in \{1, 2, \dots, \lfloor N/2 \rfloor\} & 2j \in \{2, 4, 6, 8\},\ j \in \{1, 2, 3, 4\} \\ i = 2j - 1 & i = 2j - 1 \in \{1, 3, 5, 7\} \\ 2^2 j - 1, \qquad j \in \{1, 2, \dots, \lfloor N/2^2 \rfloor\} & j \in \{1, 2\}, \qquad i \in \{3, 7\} \\ 2^3 j - 1, \qquad j \in \{1, 2, \dots, \lfloor N/2^3 \rfloor\} & j \in \{1\}, \qquad i \in \{7\} \end{array}$$

Then $\forall\, k = \{1, 2, \dots, \lfloor \log_2 N \rfloor\}$,

$$k = 1, \qquad f_{\text{out}}[i] = \begin{cases} f[i] + f[i-1] & \text{if } i = 2j-1,\ j \in \{1, 2, \dots \lfloor N/2 \rfloor\} \\ f[i] & \text{otherwise} \end{cases}$$

$$k = 2, \qquad f_{\text{out}}[i] = \begin{cases} f[i] + f[i - 2^{(k-1)}] & \text{if } i = 2^2 j - 1,\ j \in \{1, 2, \dots \lfloor N/2^2 \rfloor\} \\ f[i] & \text{otherwise} \end{cases}$$

$$k \quad , \qquad f_{\text{out}}[i] = \begin{cases} f[i] + f[i - 2^{(k-1)}] & \text{if } i = 2^k j - 1,\ j \in \{1, 2, \dots \lfloor N/2^k \rfloor\} \\ f[i] & \text{otherwise} \end{cases}$$

Thus

Definition 4 (Blelloch scan: 1st part that's reduce, i.e. up sweep)**.**

$$\forall\, k \in \{1, 2, \dots \lfloor \log_2 N \rfloor\}$$

$$f_{out}[i] = \begin{cases} f[i] + f[i - 2^{(k-1)}] & \textit{if } i = 2^k j - 1,\ j \in \{1, 2, \dots \lfloor N/2^k \rfloor\} \\ f[i] & \textit{otherwise} \end{cases} \tag{8}$$

$$\begin{tikzcd} f \arrow[r, mapsto] & f_{out} \\ K^N \arrow[r] & K^N \end{tikzcd}$$

for

2.6.2. *Blelloch scan, down sweep part.* Now consider the down sweep. First, importantly, set the "last" element in the array, entry number $\lfloor \log_2 N \rfloor$, N being the length of the array, to the identity element of K.

I'll present the first few induction steps explicitly, and the general form can be guessed from there.

$$\forall\, k \in \{\lfloor \log_2 N \rfloor, \lfloor \log_2 N \rfloor - 1, \dots, 2, 1\}$$

If $i = 2^k j - 1$, $\ j \in \{1, 2, \dots, \lfloor N/2^k \rfloor\}$, for e.g. $k = \lfloor \log_2 N \rfloor$,

$$\begin{aligned} f_{\text{out}}[i - 2^{(k-1)}] &= f[i] \\ f_{\text{out}}[i] &= f[i] + f[i - 2^{(k-1)}] \end{aligned}$$

For $k = \lfloor \log_2 N \rfloor - 1$, e.g. $k = 3 - 1 = 2$, then for this exmaple,
e.g. $i = 2^k j - 1 = 4j - 1$; $j \in \{1, 2\}$, $i \in \{3, 7\}$,

$$\begin{aligned} f_{\text{out}}[i] &= f[i] + f[i-2] \\ f_{\text{out}}[i-2] &= f[i] \end{aligned}$$

Thus, in general,

Definition 5 (Blelloch scan: down sweep step)**.**

$$\forall\, k \in \{\lfloor \log_2 N \rfloor, \lfloor \log_2 N \rfloor - 1, \dots, 2, 1\}$$

$$\begin{cases} f_{out}[i] := f[i] + f[i-2^{(k-1)}] & \textit{if } i = 2^k j - 1 \textit{ and } j \in \{1, 2, \dots, \lfloor N/2^k \rfloor\} \\ f_{out}[i-2^{(k-1)}] := f[i] & \\ f_{out}[i] := f[i] & \textit{otherwise} \end{cases} \tag{9}$$

cf. Problem Set 3

Histogram Equalization.

(1) Map
(2) Reduce
(3) Scatter
(4) Scan

2.7. **Reduce, Parallel Reduction.** I will expound upon the excellent article from Mark Harris, "Optimizing Parallel Reduction in CUDA". cf. *Optimizing Parallel Reduction in CUDA*, Mark Harris : `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`

2.8. Unit 4.

2.9. Unit 5: Lesson 5 - Optimizing GPU Programs.

2.9.1. *1. Quiz: Optimizing GPU Programs.* Quiz: principals of efficient GPU programming.

We want to

- decrease time spent on memory operations (we want to do more math; $\frac{\text{math}}{\text{memory}}$
- coalesce global memory accesses
- avoid thread divergence

We don't want to necessarily

- decrease arithmetic intensity
- do fewer memory operations per thread
- move all data to shared memory

Reasons; we want to *maximize* arithmetic intensity!

2.9.2. *22. Quiz: Tiling.* Tiling

For the kernel `transpose_parallel_per_element_tiled(float in[], float out[])`, (cf. transpose cu, we desire

$$B = A^T \text{ or } B_{ij} = A_{ji} \quad \forall\, i = 0, 1, \dots L_x - 1, \forall\, j = 0, 1 \dots L_y - 1$$

Consider $M_x = M_y = M$, with

$$\begin{aligned} M_x &\equiv \texttt{blockDim.x} \quad & i_x &\equiv \texttt{threadIdx.x} \quad & j_x &\equiv \texttt{blockIdx.x} \\ M_y &\equiv \texttt{blockDim.y} \quad & i_y &\equiv \texttt{threadIdx.y} \quad & j_y &\equiv \texttt{blockIdx.y} \end{aligned}$$

and so

$$i := i_x + j_x M_x \in \{0, 1, \dots N_x M_x - 1\} \quad j := i_y + j_y M_y \in \{0, 1, \dots N_y M_y - 1\}$$

For the *shared* indices,

$$\begin{aligned} i_{\text{sh}} &= i_x \in \{0, 1 \dots M_x - 1\} \\ j_{\text{sh}} &= i_y \in \{0, 1 \dots M_y - 1\} \end{aligned}$$

$$A_{\text{sh}} \in \mathbb{R}^{K^2} = \mathbb{R}^{K \times K} \in \texttt{\_\_shared\_\_}$$

Note that $\mathbb{R}^{K^2} = \mathbb{R}^{K \times K} \in$ `__shared__` can be a 1-dim. (or 2-dim., multidimensional!) array in shared memory (!!!).

If $A \in \text{Mat}_{\mathbb{R}}(L_x, L_y)$, $B = A^T \in \text{Mat}_{\mathbb{R}}(L_y, L_x)$.

Globally,

$$B(j, i) = B(i_y + j_y M_y, i_x + j_x M_x) = A(i, j) = A(i_x + j_x M_x, i_y + j_y M_y)$$

for

$$\begin{aligned} A_{\text{sh}} &\in \mathbb{R}^{M_x M_y} \\ A_{\text{sh}}(j_{\text{sh}}, i_{\text{sh}}) &= A_{ij} \\ A_{\text{sh}}(i_{\text{sh}}, j_{\text{sh}}) &= A_(j_{\text{sh}} + j_x M_x, i_{\text{sh}} + j_y M_y) \\ \Longrightarrow B(i_x + j_y M_y, i_y + j_x M_x) := A_{\text{sh}}(i_{\text{sh}}, j_{\text{sh}}) &= A(j_{\text{sh}} + j_x M_x, i_{\text{sh}} + j_y M_y) \end{aligned}$$

2.9.3. *Occupancy.* cf. Occupancy Part 1

Each SM (streaming multi-processor) has a limited number of

- thread blocks (so there's a maximum number of thread blocks, e.g. 8)
- threads (so there's a maximum number of threads, e.g. 1536-2048)
- registers for all threads (every thread takes a certain number of registers, and there's a maximum number of registers for all the threads, e.g. 65536)
- bytes of shared memory

cf. Quiz: Occupancy Part 2

Compile and run `deviceQuery_simplified.cpp` (can be found in Lesson 5 Code Snippets).

Look at

- Total amount of shared memory per block
- Maximum number of threads per multiprocessor
- Maximum number of threads per block

For Luebke's laptop, it's 49152 bytes, 2048, 1024 respectively, and 65536 total registers available per block.

2.9.4. *Thread Divergence.* cf. 38. Quiz: Switch Statements and Thread Divergence, of Lesson 5 - Optimizing GPU Programs

Threads in warp is 32 (check hardware, usually this is the case for modern GPUs). Only 32 threads in a warp, i.e. 2^5.

CUDA assigns thread IDs to warps:

- x varies fastest
- y varies slower
- z varies slowest

warp assignment, consider $(M_x, M_y, M_z) \equiv$ number of threads in a single thread block, in each dimensional direction.

Consider

$$\begin{aligned} i_x &\in \{0, \dots, M_x - 1\} \\ i_y &\in \{0, \dots, M_y - 1\} \end{aligned}$$

If $M_x/2^5 \in \mathbb{Z}^+$, i.e. $M_x/2^5 \geq 1$, then threads in x-direction are in warps that branch (go to same case in if-else, or switch, case statement in kernel) to same case.

2.9.5. *41. Quiz: thread Divergence in the Real World Part 1.* cf. Thread Divergence in the Real World Part 1

Example: Operating on a 1024 ×1024 image, with special handling of pixels on the boundary (boundary condition),

The maximum branch divergence of any warp (32 threads) is 2-way.

That's because, if you consider a 1 pixel deep boundary condition at the (absolute) boundary of the grid, then for most 32-thread warps in the middle, there's no divergence. At most a warp will include 1 boundary pixel, horizontally. Warp that lies completely on a vertical boundary will call the boundary condition, all of those pixels. So there's no divergence there either.

2.9.6. *43. Thread Divergence in the Real World Part 3.* cf. Thread Divergence in the Real World Part 3

- Be aware of branch divergence
- "But don't freak out about it." -Luebke

Reducing branch divergence (general principles)

- Avoid branchy code
  - consider if adjacent threads will likely take different paths
- Beware of large imbalance in thread workloads

2.9.7. *45. Host-GPU Interaction.* cf. Host-GPU Interaction of Lesson 5- Optimizing GPU Programs

-
-

2.10. **Streams.** Stream is a sequence of operations that'll execute in order.
Type is `cudaStream_t`

2.10.1. *Problem Set 5.* 2. One Basic Strategy
A. Sort the Data, into bins, sort the input data, into coarse bins
B. use threads, compute local histogram, use each thread block, to compute local histogram
c. concatenate
e.g.
Consider
$J =$ total number of coarse bins, $J \in \mathbb{Z}^+$.
$j \in \{0 \dots J-1\}$, e.g. $J = 10$.
$K =$ total number of bins,
$k \in \{0 \dots K-1\}$, e.g. $K = 100$
The condition is that $J < K$
For each thread block,

2.10.2. *Parallel radix sort.* cf. look at, in the directory `http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/` for the parallel radix sort implementation called `radix_sort_cuda.cc`.
Let $M_x \in \mathbb{Z}^+$ represent

$$M_x \leftrightarrow \texttt{blockDim.x} \tag{10}$$

$$j_x \in \{0, 1, \dots M_x - 1\} \leftrightarrow \texttt{blockIdx.x} \tag{11}$$

$\forall\, j_x \in \{0, 1 \dots, M_x - 1\}$, j_x denote which thread block we are in.
j_x also corresponds to the so-called coarse bin id.

2.11. Lesson 6.1 - Parallel Computing Patterns Part A.

2.11.1. *parallel All Pairs N-body.* cf. Quiz: All Pairs N-Body
From Arnold, Kozlov, Neishtadt, and Khukhro (2006) [17], given N-bodies $\{1, 2 \dots N\} \subset \mathbb{Z}^+$.

$$(\mathbf{r}_1, M_1), (\mathbf{r}_2, M_2), \dots (\mathbf{r}_N, M_N) \qquad \mathbf{r}_i \in \mathbb{R}^d$$

With the force acting upon the ith body ("destination") due to the jth body ("source"), $\mathbf{F}_{ij}$,

$$\mathbf{F}_{ij} = -\frac{\gamma M_i M_j}{|\mathbf{r}_{ij}|^3}\mathbf{r}_{ij} \qquad \text{with } \mathbf{r}_{ij} := \mathbf{r}_j - \mathbf{r}_i \tag{12}$$

$N(N-1)$ are the number of (unordered) pairs, for

$$\mathbf{F}_i = \sum_{i \neq j} \mathbf{F}_{ij} \tag{13}$$

with $\mathbf{F}_i$ denoting the force acting upon ith body, $N-1$ computations

$$\mathbf{F}_i = \sum_{i\neq j} \frac{-\gamma M_i M_j}{|\mathbf{r}_{ij}|^3}\mathbf{r}_{ij} = -\gamma M_i \sum_{i \neq j} \frac{M_j}{|\mathbf{r}_{ij}|^3}\mathbf{r}_{ij} \tag{14}$$

Consider another pairwise computation:

$$W(\lambda) = \sum_{i=1}^m \lambda_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)} y^{(j)} \lambda_i \lambda_j K(X^{(i)}, X^{(j)}) \tag{15}$$

Define, for (writing) convenience, $f_1(\lambda) = -\frac{1}{2}\sum_{i,j=1}^m y^{(i)} y^{(j)} \lambda_i \lambda_j K(X^{(i)}, X^{(j)})$.
cf. Quiz: How To Implement Dense N-Body as Simply As Possible
For a single body, say Ith body $(\mathbf{r}_I, M_I)$, consider $\mathbf{F}_i = \sum_{i \neq j} \mathbf{F}_{ij} \qquad \forall\, i = 1, \dots N$.
$j = I$ once, $\forall\, i = 1, \dots N$ and $i \neq I$ (Ith body as a "source") $N-1$ times.
$i = I$ once, in $\mathbf{F}_I = \sum_{i \neq j} \mathbf{F}_{Ij}$, we'll need $(\mathbf{r}_I, M_I)$, to calculate $\mathbf{F}_{Ij}$, $N-1$ times.
If all N^2 force computations, go to global memory, the number of times we'd fetch each element, as a function of N is $\boxed{2(N-1)}$.
cf. Quiz: Dividing N by N Matrix Into Tiles
Consider $N \times N$ matrix A, $A \in \text{Mat}_{\mathbb{K}}(N_x, N_y)$.
Consider dividing it by tiles, calculate each tile, a tile of $P_x \times P_y$.
So there are $\frac{N_x N_y}{P_x P_y}$ tiles overall.
For this particular problem, and in general, a problem involving pairwise *combination* out of a given set, say N bodies, consider N^2 possible pairs total. Then this all N-body algorithm idea is to transform all these combinations N^2, into a matrix, and so $N_x = N_y = N$, $P_x = P_y = P$.

$$\mathbf{F}_i = \sum_{i \neq j}^N \mathbf{F}_{ij} = \sum_{t=0}^{\frac{N}{P}-1} \sum_{\substack{j=0 \\ Pt+j \neq i}}^{P-1} \mathbf{F}_{i(Pt+j)} \equiv \sum_{t=0}^{\frac{N}{P}-1} \mathbf{F}_i^{(t)} \tag{16}$$

$2P$ fetches per tile, instead of $2P^2$, if $\forall$ tile, store the P elements into shared memory.
cf. Using on P threads
Consider this computation:

$$\mathbf{F}_i^{(t)} = \sum_{\substack{j=0 \\ pt+j \neq i}}^{P-1} \mathbf{F}_{i(Pt+j)} \qquad i = 0 \dots N-1, t = 0, 1, \dots \frac{N}{P} - 1 \tag{17}$$

For

$$i \Longleftrightarrow i_x + M_x j_x, \qquad \begin{aligned} & i_x \equiv \texttt{threadIdx.x} \in \{0, 1 \dots M_x - 1\} \\ & j_x \equiv \texttt{blockIdx.x} \in \{0, 1, \dots N/M_x - 1\} \\ & M_x = P \end{aligned}$$

So $\forall\, (i_x, j_x)$, (i_x, j_x) representing a single thread, this single thread will be responsible for computing

$$\mathbf{F}_i^{(t)} = \sum_{j=0}^{P-1} \mathbf{F}_{i(Pt+j)}$$

, as opposed to only computing $\mathbf{F}_{i(Pt+j)}$ only.
The pseudocode that Owens gives in Using On P Threads, Lesson 6.1 is the following:

```
__device__ float3
tile_calculation(Params myParams, float3 force) {
  int i;
  extern __shared__ Params[] sourceParams;
  for (i=0; i<blockDim.x; i++) {
    force += bodyBodyInteraction(myParams, sourceParams[i]);
  }
  return force;
}
```

2.12. Lesson 7.1 Additional Parallel Computing.

2.12.1. *4. Quiz: data Layout Transformation.* 1. Data layout transformation

Quiz: Global memory coalescing is important because:
(modern) DRAM systesm transfer large chunks of data per transaction.

2.12.2. *Additional Data Transformation Methods.*

2.12.3. *6. Quiz: Burst Utilization.* Burst Utilization

As a reminder of what the difference between **Array of Structures** and **Structure of Arrays** are illustrated in these examples:

Array of Structures:

```
struct foo {
        float a;
        float b;
        float c;
        float d;
} A[8];
```

Structure of Arrays

```
struct foo {
        float a[8];
        float b[8];
        float c[8];
        float d[8];
} A;
```

Quiz: which layout will perform better on these codes?

```
int i = threadIdx.x;
A[i].a++;
A[i].b += A[i].c * A[i].d;
```

and

```
int i = threadIdx.x;


A.a[i]++;
A.b[i] += A.c[i] + A.d[i];
```

Array of Structures (AoS) Example:

$$\begin{gathered}\{0,1,\dots M_x-1\}\to\mathbb{R}^4\\ i_x\mapsto(a_{i_x},b_{i_x},c_{i_x},d_{i_x})\in\mathbb{R}^4\equiv(\texttt{float})^4\end{gathered}$$

In general,

$$\begin{gathered}\{0,1,\dots L-1\}\to\mathbb{K}^d\\ i\in\mathbb{Z}\mapsto A(i)\in\mathbb{K}^d\end{gathered}\qquad(AoS)\tag{18}$$

with field $\mathbb{K}=\mathbb{R},\mathbb{C},\mathbb{Z}^+\dots$ i.e. $\mathbb{K}\in$ **Type** so that $\mathbb{K}=$ `float, int` .

So that

$$\begin{gathered}(\{0,1,\dots L-1\}\to\mathbb{K}^d)\to\mathbb{Z}\times\mathbb{K}^d\xrightarrow{\text{flatten}}\prod_{i=0}^{L-1}\mathbb{K}^d\\ (i\mapsto A(i))\mapsto(i,(A(i))^\mu)\mapsto((A(i))^1,(A(i))^2,\dots(A(i))^d)\end{gathered}$$

$$(i,\mu)\mapsto\mu+id$$

So for $t\equiv$ thread index $=i$,
$\mu+td$ address is "strided" by d.

Structure of Arrays (SoA)

Example:

$$\{0,1,\dots M_x-1\}\times\{0,1,\dots M_x-1\}\times\{0,1,\dots M_x-1\}\times\{0,1,\dots M_x-1\}=\{0,1,\dots M_x-1\}^4$$

In general

$$\prod_{\alpha=\{a,b,\dots\}}\{0,1,\dots L^{(\alpha)}-1\}\qquad(SoA)\tag{19}$$

So that

$$\begin{gathered}\prod_{\alpha=\{a,b,\dots\}}\{0,1,\dots L^{(\alpha)}-1\}\overset{\text{flatten}}{\to}\{0,1\cdots\prod_\alpha L^{(\alpha)}\}\\ i^{(\alpha)}\xrightarrow{\text{flatten}}\left(\sum_{\alpha'<\alpha}L^{(\alpha')}\right)+i^{(\alpha)}\end{gathered}\tag{20}$$

So for $\forall\, i_x\equiv$ `threadIdx.x` , or in general $\forall\, i\in\{-,1,\dots M_xN_x-1\}=i_x+j_xM_x=$ `threadIDx.x+blockDim.x * blockIdx.x`,
Want to consider,

$$i^{(\alpha)}+\sum_{\alpha'<\alpha}L^{(\alpha')}\overset{f}{\to}i$$

$$f(i^{(\alpha)}+\sum_{\alpha'<\alpha}L^{(\alpha')}=i^{(\alpha)}+\sum_{\alpha'<\alpha}L^{(\alpha')}=i$$

2.13. Final for Udacity cs344.

2.13.1. *Quiz: Final - Question 7.* cf. https://classroom.udacity.com/courses/cs344/lessons/2133758814/concepts/1421890260923#

Second question:

What is the ratio (expressed as a percentage) between kernel that uses more memory bandwidth and the kernel that uses less?

For both parts of this question, here's how to approach it.

Consider a stencil of radius RAD $\equiv R=2$. So the filter-width in 1-dimension for this 2-dimensional stencil is

$$\text{filterwidth}\equiv W=2\cdot\text{RAD}+1=5$$

W^2 multiples, and W^2-1 adds.

$\forall\, k\in K$, a (single) pixel of greyscale image, $k\in\{0,1,\dots N-1\}$,

Consider the threads per block configuration (i.e. a single thread block).

Let $M\equiv M_xM_y=1024$ in both, either, case, i.e. totla number of threads per block is 1024.

Consider $(M_x,M_y)=(1024,1)$ vs. ,
$(M_x,M_y)=(32,32)$ vs.

Correspoding to

$$\texttt{dim3}M_i(M_x,M_y)$$

for

$$\text{smooth}\ \texttt{<<<}\ \ \texttt{, M\_i >>>}\ (\texttt{v\_new},v);$$

The big idea is that we have a tile of input data we want to load into 2-dimensional shared memory. It is of size

$$(M_x+2R)(M_y+2R)\tag{21}$$

Compare this quantity to when $(M_x, M_y) = (1024, 1)$ vs. when it's $(32, 32)$.

$$(1024+4)(1+4)/(32+4)*(32+4) = \boxed{3.966}$$

This is the ratio between the kernel with more memory bandwidth and the kernel with less memory bandwidth.

Note, in the Udacity forum, Final: Updated Questions 7 & 8, jlmayfield gave the hint to consider those stencil halo cells or what I call the radius R of the stencil, as this stencil is $5x5$.

2.13.2. *Quiz: Final - Question 8.* cf. Quiz:Final - Question 8

32 × 32 block, but consider

- 4x storage per SM
- 4x registers
- 4x shared memory

now 64 x 64 since, in each dimension, 2x the global memory available ($2 \times 2 = 4$).

Express, in decimals, what the speedup is on this new GPU.

Here's how I approached this problem:

so in a single thread block, we can have a much larger tile with inputted data to reside in shared memory and so we can process a much larger tile of inputted data in a single thread block:

$$(M_x + 2R)(M_y + 2R) = (64 + 2*2)*(64+2*2) = 4624 \tag{22}$$

instead of when $(M_x, M_y) = (32, 32)$.

Not only that 4x of shared memory, 4x storage per SM, but also *4x registers*. More registers for the threads, more a thread can be doing something and not have to wait for another thread (or even to hop on to another thread if the thread itself doesn't have enough registers). More throughput, by 4x.

$$(64+2R)(64+2R)*4/(32+2R)(32+2R) = \boxed{14.27 \text{ or } 14.0} \tag{23}$$

There was confusion about Final - Question 8 in the Udacity Forums: Final - Question 8

2.13.3. *Quiz: Final - Question 9.* cf. `https://classroom.udacity.com/courses/cs344/lessons/2133758814/concepts/1389566540923`

If I want to run the maximum number of threads possible that are all resident on an SM (streaming multiprocessor) at the same time, each thread must use no more than, *how many*, registers?

$$\boxed{32 \text{ registers}} \tag{24}$$

This is obtained by looking at `http://en.wikipedia.org/wiki/CUDA` (this was given) and look for

- "Maximum number of resident threads per multiprocessor": 2048 for Compute ability (version) 3.5.
-

Final Exam Question 9 confusion
`https://discussions.udacity.com/t/final-exam-question-9-confusion/88637`

2.13.4. *Quiz: Final - Question 11.* Maximum number of blocks resident on the same SM, what is the maximum number of threads / block that we can have?

This is obtained by looking at `http://en.wikipedia.org/wiki/CUDA` (this was given) and look for

- "Maximum number of resident blocks per multiprocessor": 16 for Compute ability (version) 3.5.
- "Maximum number of resident threads per multiprocessor": 2048 for Compute ability (version) 3.5.

The answer is

$$\boxed{2048/16 = 128} \tag{25}$$

Since we take max. number of resident threads per SM, 2048, but the condition is using the max. number of resident blocks per SM, so, divide by 16.

2.13.5. *Quiz: Final - Question 12; Fast "compact" primitive.* Consider the *Fast " compact" primitive.*

(1) sum up the no. of "T" flags

e.g. warp of size 4 (only to make it easy to understand), (4 warps of 4 flags each), e.g.

$$TFTF/FTFT/TTTT/tfff$$

(1)

$$2/2/4/1$$

Let warp size $W = 2^n$; $n \in \mathbb{Z}^+$, e.g. $n = 5$ for $W = 32$.

Given shared memory $s \in (\mathbb{Z}^+)^W$,

$$\forall\, i \in \{W/2, W/4, \dots | i > 0\}$$

For given `ARRAY_SIZE = 32 =` M_x so $M_x = W$ (thread block size is equal to warp size in this case).

Keep in mind, even in a `__device__` function, we've got **access** to

$$i_x \equiv \texttt{threadIdx.x} \in \{0, 1 \dots, M_x - 1\}$$

(and so on).

$i_x < i$, i.e. $\forall\, i_x < i$ and $i_x \in \{0, 1 \dots M_x - 1\}$

$$s[i_x] = s[i_x] + s[i_x + i]$$

e.g. $i = W/2$

$$i_x \in \{0, 1 \dots, \frac{W}{2} - 1\}$$

$i = W/4$

$$i_x \in \{0, 1 \dots \frac{W}{4} - 1\}$$

$$s[i_x] = s[i_x] + s[i_x + 1] \Longrightarrow \begin{aligned} & s[0] = s[0] + s[\frac{W}{2}] + s[0 + \frac{W}{4}] \\ & \vdots \\ & s[\frac{W}{4} - 1] = s[\frac{W}{4} - 1] + s[\frac{3W}{4} - 1] + s[\frac{W}{2} - 1] \end{aligned}$$

So by induction, for $i = 1$, $i_x = 0$.

$$s[0] = \sum_{j=0}^{W-1} s[j]$$

which is the desired result (!!!).

Code answer uploaded here: Q12warpreduce shared.cu or here warpreduce.cu

3. Pointers in C; Pointers in C categorified (interpreted in Category Theory)

Suppose $v \in \text{ObjData}$, category of data **Data**,
e.g. $v \in \text{Int} \in \text{ObjType}$, category of types Type.

$$\begin{gathered} \text{Data} \xrightarrow{\&} \text{Memory} \\ v \overset{\&}{\mapsto} \&v \end{gathered}$$

with address $\&v \in \text{Memory}$.

With

assignment $pv = \&v$,

$$\begin{aligned} & pv \in \text{Objpointer}, && \text{category of pointers, pointer} \\ & pv \in \text{Memory} && (\text{i.e. not } pv \in \text{Dat, i.e. } pv \notin \text{Dat}) \end{aligned}$$

$$\text{pointer} \ \ni pv \overset{*}{\mapsto} *pv \in \text{Dat}$$

$$\begin{tikzcd} v \arrow[r, mapsto, "\&"] & \&v \arrow[d, mapsto, "="] \\ *pv \arrow[u, mapsto, "=="] & pv \arrow[l, mapsto, "*"] \end{tikzcd} \qquad \begin{tikzcd} \text{Data} \arrow[r, "\&"] & \text{Memory} \arrow[d, "="] \\ \text{Data} \arrow[u, "=="] & \text{pointer} \arrow[l, "*"] \end{tikzcd}$$

Examples. Consider `passfunction.c` in Fitzpatrick [6].

Consider the type `double`, `double` $\in$ ObjTypes.

fun1, fun2 $\in$ MorTypes namely
fun1, fun2 $\in \text{Hom}(\text{double}, \text{double}) \equiv \text{Hom}_{\text{Types}}(\text{double}, \text{double})$

Recall that

$$\begin{gathered} \text{pointer} \xrightarrow{*} \text{Dat} \\ \text{pointer} \xrightarrow{\&} \text{Memory} \end{gathered}$$

$*, \&$ are functors with domain on the category pointer.

Pointers to functions is the "extension" of functor $*$ to the codomain of MorTypes:

$$\begin{aligned} & \text{pointer} && \xrightarrow{*} \text{MorTypes} \\ & \text{fun1} && \overset{*}{\mapsto} *\text{fun1} \in \text{Hom}_{\text{Types}}(\text{double}, \text{double}) \end{aligned}$$

$$\begin{tikzcd} \text{double} \arrow[r, "\&"] & \text{Memory} \arrow[d, "\cong"] \\ \text{double} \arrow[d, "\text{cube}"'] & \text{pointer} \arrow[l, "*"'] \arrow[dl, "*"] \\ \text{double} & \end{tikzcd} \qquad \begin{tikzcd} \text{res1} \arrow[r, mapsto, "\&"] & \&\text{res1} \arrow[d, mapsto, "\cong"] \\ *\text{res1} \arrow[d, mapsto, "\text{cube}"'] & \text{res1} \arrow[l, mapsto, "*"'] \arrow[dl, mapsto, "*"] \\ *\text{res1} = y^3 & \end{tikzcd}$$

It's unclear to me how `void cube` can be represented in terms of category theory, as surely it cannot be represented as a mapping (it acts upon a functor, namely the $*$ functor for pointers). It doesn't return a value, and so one cannot be confident to say there's explicitly a domain and codomain, or range for that matter.

But what is going on is that

$$\begin{gathered} \text{pointer}\ ,\ \text{double}\ ,\ \text{pointer} \xrightarrow{\text{cube}} \text{pointer}\ ,\ \text{pointer} \\ \text{fun1}, x, \text{res1} \overset{\text{cube}}{\mapsto} \text{fun1}, \text{res1} \end{gathered}$$

s.t. $*\text{res1} = y^3 = (*\text{fun1}(x))^3$

So I'll speculate that in this case, `cube` is a functor, and in particular, is acting on $*$, the so-called deferencing operator:

$$\begin{array}{c} \text{pointer} \xrightarrow{*} \text{float} \in \text{Data} \\ \text{res1} \overset{*}{\mapsto} *\text{res1} \end{array} \xrightarrow{\text{cube}} \begin{array}{c} \text{pointer} \xrightarrow{\text{cube}(*)} \text{float} \in \text{Data} \\ \text{res1} \overset{\text{cube}(*)}{\mapsto} \text{cube}(*\text{res1}) = y^3 \end{array}$$

cf. Arrays, from Fitzpatrick [6]

$$\text{Types} \xrightarrow{\text{declaration}} \text{arrays}$$

If $x \in \text{Objarrays}$,

$$\&x[0] \in \text{Memory} \xrightarrow{==} x \in \ \text{pointer (to 1st element of array)}$$

cf. Section 2.13 Character Strings from Fitzpatrick [6]

```
char word[20] = ``four''
char *word = ``four''
```

cf. C++ extensions for C according to Fitzpatrick [6]

- simplified syntax to pass by reference pointers into functions
- inline functions
- variable size arrays

```
int n;
double x[n];
```

- complex number class

4. Summary of Udacity cs344 concepts

4.1. **Histogram.** Consider given input values (of observations).
For n observations,
For $i \in \{1, 2, \dots n\} \subset \mathbb{Z}^+, \equiv i\{0, 1, \dots n-1\} \subset \mathbb{Z}^+$.

Consider $x[i] \in B \subset \mathbb{K}$, so $x \in B^n \subset \mathbb{K}^n$.

e.g. $\mathbb{K} = \mathbb{R}$, B is a subset that we can make K bins out of, i.e.

$$B = \prod_{j=1}^{K} B_j \equiv \prod_{j=0}^{K-1} B_j$$

e.g. B is bounded interval of $\mathbb{R}$, i.e. $\begin{aligned} &\max B < \infty \\ &\min B > -\infty \end{aligned}$

Then the *histogram* H is mapping from bins to *number* of observations, i.e.

$$\begin{gathered} H : \{1, 2, \dots K\} \to \mathbb{N}^K \\ H(j) \in \mathbb{N} = \{0, 1, \dots\} \end{gathered} \tag{26}$$

s.t.

$$\boxed{n = \sum_{j=1}^K H(j)} \tag{27}$$

with n being the total number of observations.

In the implementation of $x : \{1,2,\dots n\} \to B \subset \mathbb{K}$,

this was simplified to

$$x : \{1,2,\dots n\} \to \{1,2,\dots K\} \subset \mathbb{Z}$$

meaning each observation value is *itself* which bin the observation belongs to.

Otherwise, a separate "binning" operation is needed:

$$\begin{aligned} x : \{1,2,\dots n\} \to \quad & B \subset \mathbb{K} \to \{1,2,\dots K\} \\ i \mapsto \quad & \quad x[i] \mapsto \text{if } B_{j-1} \leq x[i] < B_j, \text{ then return } j \end{aligned} \tag{28}$$

4.1.1. *Histogram implementations; histogram references.* ernestyalumni/cs344/Problem Sets/Problem Set 5/histogram/ `https://github.com/ernestyalumni/cs344/tree/master/Problem%20Sets/Problem%20Set%205/histogram`

Part 3. Notes on Professional CUDA C Programming, Cheng, Grossman, McKercher

cf. Chen, Grossman, and McKercher (2014) [4]

5. Streams and Concurrency

cf. Ch. 6 Streams and Concurrency of Chen, Grossman, and McKercher (2014) [4]

5.0.2. *Introducing Streams and Events.*

- Functions in the CUDA API with *synchronous behavior* block the host thread until they complete.
- Functions in the CUDA API with *asynchronous behavior* return control to host immediately after being called.

5.0.3. *CUDA Streams.*

- *NULL stream*, default stream the kernel launches, implicitly declared, and data transfers use if you don't explicitly specify a stream.
- *non-null streams* explicitly created and managed; if you want to overlap different CUDA operations, you must use non-null streams.

Consider

```
cudaMemcpy(..., cudaMemcpyHostToDevice);|
kernel<<<grid, block>>>(...);
cudaMemcpy(..., cudaMemcpyDeviceToHost);|
```

From device perspective, all 3 operations are issued to default stream, and executed in order they were issued. Device has no awareness any other host operations performed.

From host perspective, each data transfer is synchronous and forces idle host time while waiting for them to complete. The kernel launch is *asynchronous*, so host application almost immediately resumes execution afterwards.

Part 4. More Parallel Computing

6. Matrix Multiplication, tiled, with shared memory

Consider matrix multiplication:

$$\begin{aligned} A &\in \text{Mat}_{\mathbb{K}}(N_i^A, N_j^A) \\ B &\in \text{Mat}_{\mathbb{K}}(N_i^B, N_j^B) \\ C &\in \text{Mat}_{\mathbb{K}}(N_i^C, N_j^C) \end{aligned}$$

$AB = C$,

$$\begin{aligned} N_i^A &= N_i^C \equiv N^A \\ N_j^A &= N_i^B \equiv N^B \\ N_j^C &= N_j^B \equiv N^C \end{aligned}$$

$$C_{ij} = A_{ik}B_{kj} = \sum_{k=1}^N A_{ik}B_{kj} \qquad \begin{aligned} \forall\, i &= 1,2,\dots N^A \\ j &= 1,2,\dots N^C \end{aligned}$$

Consider, given *block size* $M \in \mathbb{Z}^+$,

$$\begin{aligned} j_y &= 0,1,\dots \frac{N^A}{M} - 1 \equiv \texttt{blockIdx.y} =: j_I \\ j_x &= 0,1,\dots \frac{N^C}{M} - 1 \equiv \texttt{blockIdx.x} =: j_J \end{aligned}$$

Given $A \in \text{Mat}(N_i^A, N_j^A)$, given $(i,j) \in (\mathbb{Z}^+)^2$,

As C++ works also with pointers as C, "point" the first element of our $A_{\text{sub}} \in \text{Mat}_{\mathbb{K}}(M,M)$ submatrix that we want, to where we "start from" in the "source" matrix A, which is at $A_{iM,jM}$:

$$A_{\text{sub}}(0,0) := A(iM, jM)$$

and so this means that i.e. we had supposed $\begin{aligned} i &= 0,1,\dots \frac{N_i^A}{M} - 1. \\ j &= 0,1\dots \frac{N_j^A}{M} - 1 \end{aligned}$

So consider

$$\begin{aligned} i &= i_y + j_I M = 0,1,\dots N^A - 1 \\ j &= i_x + j_J M = 0,1,\dots N^C - 1 \end{aligned}$$

and so the crucial step is seen as

$$C_{ij} = \sum_{k=0}^{N^B-1} A_{ik}B_{kj} = \sum_{j_K=0}^{\frac{N^B}{M}-1} \sum_{k=0}^{M-1} A_{i(k+j_K M)} B_{(k+j_K M),j}$$

I originally implemented matrix multiplication based on the CUDA C Programming Guide here at github : ernestyalumni/CompPhys/moreCUDA/matmultShare.cu

Indeed, the idea from the CUDA C Programming Guide is essentially the above - I will reiterate again, since it was important in Pöppl's presentation for TUM in HPC - Algorithms and Applications (cf. Introduction to CUDA): given

$$\begin{aligned} A &\in \text{Mat}_{\mathbb{R}}(N_i^A, N_j^A) \\ B &\in \text{Mat}_{\mathbb{R}}(N_i^B, N_j^B) \\ C &\in \text{Mat}_{\mathbb{R}}(N_i^C, N_j^C) \end{aligned}$$

Necessarily, for

$$AB = C$$

(29)
$$A_{ik}B_{kj} = C_{ij} \qquad \begin{aligned} &\forall\, i = 1,2\dots N_i^A \\ &\forall\, j = 1,2,\dots N_j^B \end{aligned}$$

we have to have

$$\begin{aligned} N_i^C &= N_i^A \\ N_j^C &= N_j^B \\ N_j^A &= N_i^B \end{aligned}$$

We sought thread warp coalescing, in conjunction with the so-called ”row-major ordering” (also known as ”row-major”, ”order”) in the x-direction of thread block. Keep that in mind for the grid, block thread(s) assignment strategy.

6.0.4. *Grid, block thread(s) assignment strategy for Matrix multiplication.* Consider $N_j^B \times N_i^A$ calculations, i.e. $(j,i) \in \{0,1\dots N_j^B-1\}\times\{0,1\dots N_i^A-1\} \subset (\mathbb{Z}^+)^2$.

For (thread block) size $M \times M \equiv$ `dim3 dimBlock(M,M)`.

Then, the number of thread blocks along each dim. of the grid would be $\frac{N_j^B+M-1}{M} = \lfloor \frac{N_j^B}{M} \rfloor$, $\frac{N_i^A+M-1}{M} = \lfloor \frac{N_i^A}{M} \rfloor$

Nevertheless, the whole concept of tiling, tiling pattern with shared memory, is encapsulated in this equation:

(30)
$$C_{ij} = \sum_{k=0}^{N^B-1} A_{ik}B_{kj} = \sum_{j_K=0}^{\frac{N^B}{M}-1}\sum_{k=0}^{M-1} A_{i(k+j_KM)}B_{(k+j_KM),j}$$

7. Dense Linear Algebra

cf. Dense Linear Algebra, HPC - Algorithms and Applications, Alexander Pöppl, TUM
Bader, Pöppl, and Khakhutskyy [8]

Part 5. C++ and Computational Physics

cf. 2.1.1 Scientific hello world from Hjorth-Jensen (2015) [7]
in C,

```
int main (int argc, char* argv[])
```

`argc` stands for number of command-line arguments
`argv` is vector of strings containing the command-line arguments with
`argv[0]` containing name of program
`argv[1] , argv[2], ...` are command-line args, i.e. the number of lines of input to the program
“To obtain an executable file for a C++ program” (i.e. compile (???)),

```
gcc -c -Wall myprogram.c
gcc -o myprogram myprogram.o
```

`-Wall` means warning is issued in case of non-standard language
`-c` means compilation only
`-o` links produced object file `myprogram.o` and produces executable `myprogram`

```
# General makefile for c - choose PROG =  name of given program

# Here we define compiler option, libraries and the target
CC= c++ -Wall
PROG= myprogram

# Here we make the executable file
${PROG} :           ${PROG}.o
                    ${CC} ${PROG}.o -o ${PROG}

# whereas here we create the object file

#{PROG}.o :          ${PROG}.cpp
                    ${CC} -c ${PROG}.cpp
```

Here’s what worked for me:

```
CC= g++ -Wall
PROG= program1

# Here we make the executable file
${PROG} :           ${PROG}.o
        ${CC} ${PROG}.o -o ${PROG}

# whereas here we create the object file

${PROG}.o :         ${PROG}.cpp
        ${CC} -c ${PROG}.cpp

# EY : 20160602notice the different suffixes, and we see the pattern for the syntax

# (note: the <tab> in the command line is necessary formake towork)
# target: dependency1 dependency2 ...
#         <tab> command
```

cf. 2.3.2 Machine numbers of Hjorth-Jensen (2015) [7]
cf. 2.5.2 Pointers and arrays in C++ of Hjorth-Jensen (2015) [7]
Initialization (diagram):

$$\&\text{var} = \texttt{0x7ffc97efbd8c} \xmapsto{=} \text{pointer} = \&\text{var} = \texttt{0x7ffc97efbd8c}$$

$$\text{Memory} \xrightarrow{\qquad\qquad =\qquad\qquad} \text{pointer}$$

$$\text{(memory) addresses} \xrightarrow{\qquad\quad =\qquad\quad} \text{Obj(pointer)}$$

Referencing and deferencing operations on pointers to variables

$$\begin{tikzcd}
\text{var} \arrow[r, mapsto, "="] & \&\text{var} = \texttt{0x7ffc97egbd8c} & & \\
\text{var}, 421 \arrow[u, mapsto] \arrow[r, mapsto, "=="] \arrow[d, mapsto] & \text{True} & \text{pointer} = \texttt{0x7ffc97egbd8c} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"] & 421 \arrow[r, mapsto, "\text{typedef}"] & \text{int} \\
421 & & \&\text{pointer} = \texttt{0x7ffc97egbd8c} & &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "="] & \mathbf{Memory} & & & \\
\mathbf{Types} \otimes \mathbf{Dat} \arrow[u] \arrow[r, "=="] \arrow[d] & \text{Boolean} = \{\text{True}, \text{False}\} & \mathbf{pointer} \arrow[r, "*"] \arrow[d, "\&"] & \mathbf{Dat} \arrow[r, "\text{typedef}"] & \mathbf{Types} \\
\mathbf{Dat} & & \mathbf{Memory} & &
\end{tikzcd}$$

7.1. **Numerical differentiation and interpolation (in C++).** cf. Chapter 3 "Numerical differentiation and interpolation" of Hjorth-Jensen (2015) [7].

This is how I understand it.

Consider the Taylor expansion for $f(x) \in C^\infty(\mathbb{R})$:

$$f(x) = f(x_0) + \sum_{j=1}^\infty \frac{f^{(j)}(x_0)}{j!} h^j$$

For $x = x_0 \pm h$,

$$f(x) = f(x_0 \pm h) = f(x_0) + \sum_{j=1}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} h^{2j} \pm \sum_{j=1}^\infty \frac{f^{(2j-1)}(x_0)}{(2j-1)!} h^{2j-1}$$

Then

$$\begin{aligned}
f(x_0 + 2^k h) - f(x_0 - 2^k h) &= 2 \sum_{j=1}^\infty \frac{f^{(2j-1)}}{(2j-1)!}(x_0) 2^{k(2j-1)} h^{2j-1} = \\
&= 2 \left[f^{(1)}(x_0) 2^k h + \sum_{j=2}^\infty \frac{f^{(2j-1)}(x_0)}{(2j-1)!} 2^{k(2j-1)} h^{2j-1} \right] = \\
&= 2 \left[f^{(1)}(x_0) 2^k h + \frac{f^{(3)}(x_0)}{3!} 2^{k(3)} h^3 + \sum_{j=3}^\infty \frac{f^{(2j-1)}(x_0)}{(2j-1)!} 2^{k(2j-1)} h^{2j-1} \right]
\end{aligned}$$

So for $k=1$,

$$f(x_0 + h) - f(x_0 - h) = 2 \left[f^{(1)}(x_0) h + \sum_{j=1}^\infty \frac{f^{(2j+1)}(x_0)}{(2j+1)!} h^{2j+1} \right]$$

Now

$$\begin{aligned}
f(x_0 + 2^k h) + f(x_0 - 2^k h) - 2f(x_0) &= \\
&= 2 \sum_{j=1}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} 2^{2jk} h^{2j} = \\
&= 2 \left[\frac{f^{(2)}(x_0)}{2} 2^{2k} h^2 + \sum_{j=2}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} 2^{2jk} h^{2j} \right] = \\
&= 2 \left[\frac{f^{(2)}(x_0)}{2} 2^{2k} h^2 + \frac{f^{(4)}(x_0)}{4!} 2^{4k} h^4 + \sum_{j=3}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} 2^{2jk} h^{2j} \right]
\end{aligned}$$

Thus for the case of $k=1$,

$$f(x_0 + h) + f(x_0 - h) - 2f(x_0) = f^{(2)}(x_0) h^2 + 2 \sum_{j=2}^\infty \frac{f^{(2j)}(x_0)}{(2j)!} h^{2j}$$

$$\begin{aligned}
\frac{f(x_0+h) - f(x_0-h)}{2h} &= f^{(1)}(x_0) + \sum_{j=1}^\infty \frac{f^{(2j+1)}(x_0)}{(2j+1)!} h^{2j} \\
\frac{f(x_0+h) + f(x_0-h) - 2f(x_0)}{h^2} &= f^{(2)}(x_0) + 2 \sum_{j=2}^\infty \frac{f^{(2(j+1))}(x_0)}{(2(j+1))!} h^{2j}
\end{aligned}$$

A pattern now emerges on how to include more calculations at points $x_0, x_0 \pm 2^k h$ so to obtain better accuracy $O(h^l)$. For instance,

Given 5 pts. $\{x_0, x_0 \pm h, x_0 \pm 2h\}$,

$$\begin{aligned}
f(x_0 + 2h) - f(x_0 - 2h) &= 2[f^{(1)}(x_0) 2^1 h + \frac{f^{(3)}(x_0)}{3!} 2^3 h^3 + O(h^5)] \\
f(x_0 + h) - f(x_0 - h) &= 2[f^{(1)}(x_0) h + \frac{f^{(3)}(x_0)}{3!} h^3 + O(h^5)] \\
\implies f'(x_0) &= \frac{f(x_0 - 2h) - 8f(x_0 - h) + 8f(x_0 + h) - f(x_0 + 2h)}{12h} + O(h^4)
\end{aligned}$$

Hjorth-Jensen (2015) [7] argues, on pp. 46-47, that the additional evaluations are time consuming, to obtain further accuracy, so it's a balance.

To summarize, for $O(h^2)$ accuracy,

$$\begin{aligned}
\frac{f(x_0+h) - f(x_0-h)}{2h} &= f^{(1)}(x_0) + \sum_{j=1}^\infty \frac{f^{(2j+1)}(x_0)}{(2j+1)!} h^{2j} && O(h^2) \\
\frac{f(x_0+h) + f(x_0-h) - 2f(x_0)}{h^2} &= f^{(2)}(x_0) + 2 \sum_{j=1}^\infty \frac{f^{(2j+2)}(x_0)}{(2j+2)!} h^{2j} && O(h^2)
\end{aligned}$$

8. Interpolation

cf. 3.2 Numerical Interpolation and Extrapolation of Hjorth-Jensen (2015) [7]

Given $N+1$ pts. $\begin{aligned} y_0 &= f(x_0) \\ y_1 &= f(x_1) \\ &\vdots \\ y_N &= f(x_N) \end{aligned}$, x_i's distinct (none of x_i values equal)

We want a polynomial of degree n s.t. $p(x) \in \mathbb{R}[x]$

$$p(x_i) = f(x_i) = y_i \qquad i = 0, 1 \dots N$$

$$p(x) = a_0 + a_1(x - x_0) + \dots + a_i \prod_{j=0}^{i-1} (x - x_j) + \dots + a_N(x - x_0)\dots(x - x_{N-1}) = a_0 + \sum_{i=1}^N a_i \prod_{j=0}^{i-1} (x - x_j)$$

$$\begin{aligned} & a_0 = f(x_0) \\ & a_0 + a_1(x_1 - x_0) = f(x_1) \\ & \vdots \\ & a_0 + \sum_{i=1}^k a_i \prod_{j=0}^{i-1} (x_k - x_j) = f(x_k) \end{aligned}$$

Hjorth-Jensen (2015) [7] mentions this Lagrange interpolation formula (I haven't found a good proof for it).

$$\boxed{p_N(x) = \sum_{i=0}^N \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i} \tag{31}$$

9. Classes (C++)

cf. C++ Operator Overloading in expression

Take a look at this link: C++ Operator Overloading in expression. This point isn't emphasized enough, as in Hjorth-Jensen (2015) [7]. This makes doing something like

$$d = a * c + d/b$$

work the way we expect. Kudos to user fredoverflow for his answer:

"The expression `(e_x*u_c)` is an rvalue, and references to non-const won't bind to rvalues.

Also, member functions should be marked `const` as well."

9.1. What are lvalues and rvalues in C and C++? C++ Rvalue References Explained

Original definition of *lvalues* and *rvalues* from *C*:

lvalue - expression e that may appear on the left or on the right hand side of an assignment

rvalue - expression that can only appear on right hand side of assignment =.

Examples:

```
int a = 42;
int b = 43;

// a and b are both l-values
a = b; // ok
b = a; // ok
a = a * b; // ok
// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42; // error, rvalue on left hand side of assignment
```

In *C++*, this is still useful as a first, intuitive approach, but

lvalue - expression that refers to a memory location and allows us to take the address of that memory location via the & operator.

rvalue - expression that's not a lvalue

So & reference *functor* can't act on rvalue's.

9.2. Functors (C++); C++ Functors; C++ class templates.

For categories $\mathbf{A}, \mathbf{B}$, consider trying to understand, wrap your mind around C++, especiall C++11/14 style functors. The key *insight* is *composability*: use the mathematical property of **composition**.

$$\begin{tikzcd} \mathbf{A} \arrow[r, "F"] \arrow[d, "\langle \text{Type} \rangle"] & \mathbf{B} \arrow[d, "\langle \text{Type} \rangle"] \\ \text{Type} \circ \mathbf{A} \arrow[r, "\langle \text{Type} \rangle \circ F"] & \text{Type} \circ \mathbf{B} \end{tikzcd} \tag{32}$$

This webpage from K Hong helped with understanding C++11/14 style functors: C++ Tutorial - Functors(Function Objects) - 2017, cf. `http://www.bogotobogo.com/cplusplus/functors.php`

I implemented all of that in the webpage here: github functors.cpp , `ernestyalumni/CompPhys/Cpp/Cpp14/functors.cpp`

I will try to write a dictionary between math, i.e. mathematical formulation, and the class templates, structs.

I looked at pp. 213 of Conlon, pp. 513 of Rotman, and looked up keywords "functional."

Consider the bilinear functional that results in a function, i.e. $\mathcal{C}^\infty(\mathbb{R})$.

$$\mathbb{R} \times \mathbb{R} \to C^\infty(\mathbb{R})$$

i.e.

$$\begin{aligned} \mathbb{R} \times \mathbb{R} & \to \text{Hom}_{\mathbb{R}}(\mathbb{R}, \mathbb{R}) \\ (a, b) & \mapsto f(x) = ax + b \end{aligned} \tag{33}$$

with

$$\text{Hom}_{\mathbb{R}}(\mathbb{R}, \mathbb{R}) \ni \{ \mathbb{R} \xrightarrow{f} \mathbb{R} \}$$

Compare this directly to `class Line` in `functor.cpp`. Note that this is *class object working as a functor*:

```
class Line {
        double a;                       // slope
        double b;                       // y-intercept

        public:
                Line(double slope = 1, double yintercept = 1) :
                        a(slope), b(yintercept) { }
                double operator()(double x){
                        return a*x + b;
                }};
```

Now consider the use of C++ function object, but with non-type template, C++ templates:

$$\begin{aligned} \mathbb{R} & \to \text{Hom}_{\mathbb{R}}(\mathbb{R},\mathbb{R}) \\ x & \mapsto f(y) = y + x \end{aligned} \tag{34}$$

But suppose $y \in \mathbb{R}^d$, e.g. $y_i \in \mathbb{R}$, $i = 0, 1, \dots d-1$.

$$\begin{aligned} \mathbb{R} & \to \text{Hom}_{\mathbb{R}}(\mathbb{R}^d,\mathbb{R}^d) \\ x & \mapsto f(y) = y + x \text{ or } (f(y))_i = y_i + x \qquad \forall\, i = 0, 1, \dots d-1 \end{aligned} \tag{35}$$

So for the *class template*, to generalize $\mathbb{R}$ to some choice of field $\mathbb{K}$, generalize $\mathbb{R}^d$ to R-module R.

$$\begin{aligned} \mathbb{K} & \to \text{Hom}_{\mathbb{K}}(\mathbb{K},\mathbb{K}) \\ x & \mapsto (f(y))_i = y_i + x \qquad \forall\, i = 0, 1, \dots d-1 \end{aligned} \tag{36}$$

And so the strategy is to generalize type by the class template (declaration), define the Hom from M to M by defining the Hom from $\mathbb{K}$ to $\mathbb{K}$ for each element of M.

Compare this directly to the code for class Add in `functor.cpp`:

```
  template <typename T>
class  Add
{
        T  x;

        public:
                Add(T  xx)  :  x(xx)  {  }
                void  operator()(T&  e)  const  {  e  +=  x;  }
};
...


std::for_each(v2.begin(),  v2.end(),  Add<int>(10));

        std::for_each(v2.begin(),  v2.end(),  Add<int>(*v2.begin())  );
```

Notice how the construction of the Hom needs an input.

10. Numerical Integration

10.0.1. *Trapezoid rule (or trapezoidal rule).* See Integrate.ipynb.

From there, consider integration on $[a,b]$, considering $h := \frac{b-a}{N}$, and $N+1$ (grid) points, $\{a, a+h, a+2h, \dots, a+jh, \dots, a+Nh = b\}_{j=0\dots N}$.

Then $\frac{N}{2}$ pts. are our "x_0"; x_0's $= \{a+h, a+3h, \dots, a+(2j-1)h, \dots, a+\left(\frac{2N}{2}-1\right)h\}_{j=1\dots\frac{N}{2}}$.

Notice how we really need to care about if N is even or not. If N is not even, then we'd have to deal with the integration at the integration limits and choosing what to do.

Then

$$\int_a^b f(x)dx = \sum_{j=1}^{N/2} \int_{a+(2j-1)h-h}^{a+(2j-1)h+h} f(x)dx = \sum_{j=1}^{N/2} \frac{h}{2}(2f(a+(2j-1)h) + f(a+2(j-1)h) + f(a+2jh)) =$$

$$= h(f(a)/2 + f(a+h) + \dots + f(b-h) + \frac{f(b)}{2}) = h\left(\frac{f(a)}{2} + \sum_{j=1}^{N-1} f(a+jh) + \frac{f(b)}{2} \right)$$

10.0.2. *Midpoint method or rectangle method.* .

Let $h := \frac{b-a}{N}$ be the step size. The grid is as follows:

$$\{a, a+h, \dots, a+jh, \dots, a+Nh = b\}_{j=0\dots N}$$

The desired midpoint values are at the following N points:

$$\{a + \frac{h}{2}, a + \frac{3}{2}h, \dots, a + \frac{(2j-1)h}{2}, \dots, a + \left(N - \frac{1}{2}\right)h\}_{j=1\dots N}$$

and so

$$\int_a^b f(x)dx \approx \sum_{j=1}^N f(x_j)h = \sum_{j=1}^N f\left(a + \frac{(2j-1)h}{2}\right)h \tag{37}$$

10.0.3. *Simpson rule.* The idea is to take the next "order" in the Lagrange interpolation formula, the second-order polynomial, and then we can rederive Simpson's rule. The algebra is worked out in Integrate.ipynb.

From there, then we can obtain Simpson's rule,

$$\int_a^b f(x)dx = \sum_{j=1}^{N/2} \int_{a+2(j-1)h}^{a+2jh} f(x)dx = \sum_{j=1}^{N/2} \frac{h}{3}(4f(a+(2j-1)h) + f(a+2(j-1)h) + f(a+2jh)) =$$

$$= \frac{h}{3}\left[f(a) + f(b) + \sum_{j=1}^{N/2} 4f(a+(2j-1)h) + 2\sum_{j=1}^{N/2-1} f(a+2jh) \right]$$

10.1. **Gaussian Quadrature.** cf. Hjorth-Jensen (2015) [7], Section 5.3 Gaussian Quadrature, Chapter 5 Numerical Integration

11. Runge-Kutta methods (RK)

cf. Hjorth-Jensen (2015) [7], Section 8.4 *More on finite difference methods, Runge-Kutta methods* and *wikipedia*, "Runge-Kutta methods," `https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods`

While Runge-Kutta methods are useful initially for ordinary differential equations, remember that under certain (very general, in fact, for smooth manifolds even) conditions, vector fields admit integral lines and there you simply solve an system of linear ODEs.

12. Partial Differential Equations

12.0.1. *Explicit Scheme.* cf. Hjorth-Jensen (2015) [7], Section 10.2.1 Explicit Scheme

Consider

$$\begin{gathered} u = u(t,x) \in C^\infty(M) = C^\infty(\mathbb{R} \times N) \\ \Delta u = \frac{\partial u}{\partial t}(t,x) \end{gathered} \tag{38}$$

with initial conditions

$$u(0,x) = g(x) \qquad \forall\, 0 < x < L_x \text{ or } x \in \Omega \subset N \text{ (in general)}$$

e.g. $L_x = 1$

and boundary conditions

$$\begin{aligned} u(t,0) &= a(t) \quad t \geq 0 \\ u(t,L) &= b(t) \quad t \geq 0 \end{aligned}$$

Consider the act of discretization as a transformation or a functor:

$$\begin{gathered} \frac{\partial u}{\partial t} = \frac{u(t+\Delta t, x) - u(t,x)}{\Delta t} + O(\Delta t) \xrightarrow{\text{discretize}} \frac{u(t_j + \Delta t, x_i) - u(t_j, x_i)}{\Delta t} \\ \frac{\partial^2 u}{\partial (x^i)^2} \approx \frac{u(t, x+\Delta x) - 2u(t,x) + u(t, x-\Delta x)}{(\Delta x)^2} \xrightarrow{\text{discretize}} \frac{u(t_j, x_i + \Delta x) - 2u(t_j, x_i) + u(t_j, x_i - \Delta x)}{(\Delta x)^2} \end{gathered} \tag{39}$$

$$\implies u(t_j + \Delta t, x_i) = \frac{\Delta t}{(\Delta x)^2} u(t_j, x_i + \Delta x) + \left(1 - \frac{2\Delta t}{(\Delta x)^2} \right) u(t_j, x_i) + \frac{\Delta t}{(\Delta x)^2} u(t_j, x_i - \Delta x) \tag{40}$$

Discretize the initial conditions:

$$u(0,x) = g(x) \xrightarrow{\text{discretize}} u(0, x_i) = g(x_i) \tag{41}$$

and so, for the first step,

$$u(\Delta t, x_i) = \frac{\Delta t}{(\Delta x)^2} g(x_i + \Delta x) + \left(1 - \frac{2\Delta t}{(\Delta x)^2} \right) g(x_i) + \frac{\Delta t}{(\Delta x)^2} g(x_i - \Delta x) \tag{42}$$

It would appear to be instructive to show what discretize is doing, as a commutative diagram:

$$\begin{tikzcd} \mathbb{R} \times N \arrow[r, "\text{discretize}"] \arrow[d] & \mathbb{Z}^+ \times (\{0 \dots L_x - 1\} \times \{0 \dots L_y - 1\} \times \{0 \dots L_z - 1\}) \subset \mathbb{Z}^+ \times \mathbb{Z}^d \arrow[d] \\ C^\infty(\mathbb{R} \times N) \arrow[r, "\text{discretize}"] & C^\infty(\mathbb{Z}^+ \times (\{0 \dots L_x - 1\} \times \{0 \dots L_y - 1\} \times \{0 \dots L_z - 1\})) \subset C^\infty(\mathbb{Z}^+ \times \mathbb{Z}^d) \end{tikzcd} \tag{43}$$

Then there's this so-called "sparse" (I think this means that there are a lot more zeros as values for the entries in a matrix than there are nonzero values), tridiagonal (diagonal and next to the diagonal, diagonals) matrices representation of the time-evolution transformation/operator. I'll call this transformation over to this matrix representation, this functor, *matricer*.

$$C^\infty(\mathbb{Z}^+ \times (\{0 \dots L_x - 1\} \times \{0 \dots L_y - 1\} \times \{0 \dots L_z - 1\})) \xrightarrow{\text{matricer}} \mathbb{Z}^+ \times \mathbb{R}^{L_x L_y L_z} = \mathbb{Z}^+ \times \text{Mat}_{\mathbb{R}}(L_x, L_y, L_z) \tag{44}$$

For the boundary conditions,

$$\begin{array}{lll} u(t,0) = a(t) & t \geq 0 & u(t,0) = a(t) \\ & \xrightarrow{\text{discretize}} & \\ u(t,L) = b(t) & t \geq 0 & u(t, x_{L_x - 1}) = b(t) \end{array}$$

For 1-dim. case, $V_j \in \text{Mat}_{\mathbb{R}}(L_x)$ (vector or "column" matrix)

$$V_j = \begin{bmatrix} u(t_j, x_2) \\ u(t_j, x_3) \\ \vdots \\ u(t_j, x_{L_x - 3}) \end{bmatrix}$$

and so the "time-evolution" operator/transformation is

$$\widehat{A} \in \text{Mat}_{\mathbb{R}}(L_x - 4, L_x - 4) = \text{End}(\mathbb{R}^{L_x - 4}, \mathbb{R}^{L_x - 4})$$

$$\widehat{A} = \begin{bmatrix} \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} & 0 & 0 & \dots & 0 \\ 0 & \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} & 0 & \dots & 0 \\ & & & \ddots & & & \\ 0 & 0 & \dots & 0 & \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} \end{bmatrix} \tag{45}$$

Note that in the specialized 1-dim. case where $a(t) = b(t) = 0$ (boundary conditions for both ends is of value 0), then we can, in this specialized case, define the matrix $\widehat{A}$ to be

$$\widehat{A} \in \text{Mat}_{\mathbb{R}}(L_x - 2, L_x - 2) = \text{End}(\mathbb{R}^{L_x - 2}, \mathbb{R}^{L_x - 2})$$

$$\widehat{A} = \begin{bmatrix} 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} & 0 & 0 & \dots & 0 & 0 \\ \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} & 0 & 0 & \dots & 0 \\ 0 & \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} & 0 & \dots & 0 \\ & & & \ddots & & & \\ 0 & 0 & \dots & 0 & \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} & \frac{\Delta t}{(\Delta x)^2} \\ 0 & 0 & 0 & \dots & 0 & \frac{\Delta t}{(\Delta x)^2} & 1 - 2\frac{\Delta t}{(\Delta x)^2} \end{bmatrix} \tag{46}$$

e.g.

$$g(x) = \sin\left(\frac{\pi}{l_x} x \right)$$

with an analytic solution of

$$u(t,x) = \sin\left(\frac{\pi}{l_x} x \right) \exp\left(-\left(\frac{\pi}{l_x} \right)^2 t \right)$$

It was bizarre to me that in Hjorth-Jensen (2015) [7], Section 10.2.1 Explicit Scheme, on pp. 307, Hjorth-Jensen went through a lengthy and thorough explanation of this "matricer" operation, i.e. doing the time-evolution with a matrix on a vector of values at grid points, and yet in the pseudo-code, essentially, there is no trace of that matrix! It's essentially a local "stencil" operation. What the heck?

I present the "matrix form" code in my github repository: ‘diffusion1dexplicit.cpp‘. To be explicit, the code follows the previous writeup, with its notation, essentially one-to-one.

12.0.2. *Implicit scheme.* cf. Hjorth-Jensen (2015) [7], Section 10.2.2 Implicit Scheme

Consider

$$\begin{array}{ll} \text{backwards formula :} & \frac{\partial u}{\partial t}(t,x) \approx \frac{u(t_j, x_i) - u(t_j - \Delta t, x_i)}{\Delta t} \text{ or even} \\ \text{midpoint approximations :} & \frac{\partial u}{\partial t}(t,x) \approx \frac{u(t_j + \Delta t, x_i) - u(t_j - \Delta t, x_i)}{2\Delta t} \end{array} \tag{47}$$

Consider the same spatial discretization as before for the Laplacian:

$$\frac{\partial^2 u}{\partial (x^i)^2} \approx \frac{u(t, x+\Delta x) - 2u(t,x) + u(t, x-\Delta x)}{(\Delta x)^2} \xrightarrow{\text{discretize}} \frac{u(t_j, x_i + \Delta x) - 2u(t_j, x_i) + u(t_j, x_i - \Delta x)}{(\Delta x)^2}$$

and so for the backwards formula case,

$$\begin{gathered} \frac{u(t_j, x_i) - u(t_j - \Delta t, x_i)}{\Delta t} = \frac{u(t_j, x_i + \Delta x) - 2u(t_j, x_i) + u(t_j, x_i - \Delta x)}{(\Delta x)^2} \\ u(t_j - \Delta t, x_i) = -\frac{\Delta t}{(\Delta x)^2} u(t_j, x_i + \Delta x) + \left(1 + \frac{2\Delta t}{(\Delta x)^2} \right) u(t_j, x_i) - \frac{\Delta t}{(\Delta x)^2} u(t_j, x_i - \Delta x) \end{gathered} \tag{48}$$

resulting in the backwards time-evolution matrix $\widehat{A}$ (keeping in mind the special boundary condition of 0 value for u at both ends, for the sake of a simplified discussion):

$$\widehat{A} \in \text{Mat}_{\mathbb{R}}(L_x-2, L_x-2) = \text{End}(\mathbb{R}^{L_x-2}, \mathbb{R}^{L_x-2})$$

$$\widehat{A} = \begin{bmatrix} 1+2\frac{\Delta t}{(\Delta x)^2} & -\frac{\Delta t}{(\Delta x)^2} & 0 & 0 & \dots & 0 \\ -\frac{\Delta t}{(\Delta x)^2} & 1+2\frac{\Delta t}{(\Delta x)^2} & -\frac{\Delta t}{(\Delta x)^2} & 0 & \dots & 0 \\ & & & \ddots & & \\ 0 & 0 & \dots & -\frac{\Delta t}{(\Delta x)^2} & 1+2\frac{\Delta t}{(\Delta x)^2} & -\frac{\Delta t}{(\Delta x)^2} \\ 0 & 0 & 0 & \dots & -\frac{\Delta t}{(\Delta x)^2} & 1+2\frac{\Delta t}{(\Delta x)^2} \end{bmatrix} \qquad (49)$$

$$\widehat{A}u^j = u^{j-1}$$

and so

$$\widehat{A}^{-1}u^{j-1} = u^j$$

12.1. **Crank-Nicolson method.** Hjorth-Jensen (2015) [7], Section 10.2.3 Crank Nicholson scheme has a write up about the Crank-Nicolson method, but the derivation is unclear (and sloppy, in that after the Taylor expansions, he says that the terms magically add up to the desired result, and the "approximation" notation is vacuous in that nothing new was conveyed). Rather, look at Crank Nicolson Scheme for the Heat Equation for a clearer derivation, that drives home the point of looking at the time between time steps.

Consider the following Taylor expansions about $t^{1/2} := t + \frac{\Delta t}{2}$.

$$\begin{aligned} & u(t+\Delta t, x) = u(t + \frac{\Delta t}{2} + \frac{\Delta t}{2}, x) \equiv u(t^{1/2} + \frac{\Delta t}{2}, x) = u(t^{1/2}, x) + \frac{\Delta t}{2}\frac{\partial u}{\partial t}(t^{1/2}, x) + \frac{1}{2}\left(\frac{\Delta t}{2}\right)^2 \frac{\partial^2 u}{\partial t^2}(t^{1/2}, x) + O((\Delta t)^3) \\ & u(t, x) = u(t + \frac{\Delta t}{2} - \frac{\Delta t}{2}, x) \equiv u(t^{1/2} - \frac{\Delta t}{2}, x) = u(t^{1/2}, x) - \frac{\Delta t}{2}\frac{\partial u}{\partial t}(t^{1/2}, x) + \frac{1}{2}\left(\frac{\Delta t}{2}\right)^2 \frac{\partial^2 u}{\partial t^2}(t^{1/2}, x) + O((\Delta t)^3) \\ & u(t+\Delta t, x) - u(t, x) = \Delta t \frac{\partial u}{\partial t}(t^{1/2}, x) + O((\Delta t)^3) \end{aligned}$$

$$\implies \frac{\partial u}{\partial t}(t^{1/2}, x) = \frac{u(t+\Delta t, x) - u(t,x)}{\Delta t}$$

with $O((\Delta t)^2)$ order of accuracy.

To approximate

$$\frac{\partial^2 u}{\partial x^2}(t + \frac{\Delta t}{2}, x) \equiv \frac{\partial^2 u}{\partial x^2}(t^{1/2}, x)$$

use average of second, centered differences for $\frac{\partial^2 u}{\partial x^2}(t+\Delta t, x)$ and $\frac{\partial^2 u}{\partial x^2}(t, x)$.

$$\frac{\partial^2 u}{\partial x^2}(t + \frac{\Delta t}{2}, x) \approx \frac{1}{2}\left[\frac{u(t+\Delta t, x+\Delta x) - 2u(t+\Delta t, x) + u(t+\Delta t, x-\Delta x)}{(\Delta x)^2} + \frac{u(t, x+\Delta x) - 2u(t,x) + u(t, x-\Delta x)}{(\Delta x)^2} \right]$$

Then for

$$\frac{\partial u}{\partial t}(t^{1/2}, x) = C_0 \Delta u(t^{1/2}, x) \xrightarrow{discretize}$$

$$u(t+\Delta t, x) - u(t, x) =$$

$$= \frac{1}{2}C_0\frac{\Delta t}{(\Delta x)^2}(u(t+\Delta t, x+\Delta x) - 2u(t+\delta t, x) + u(t+\Delta t, x-\Delta x)) + \frac{1}{2}C_0\frac{\Delta t}{(\Delta x)^2}u(t, x+\Delta x) + \frac{-\Delta t}{(\Delta x)^2}C_0 u(t,x) + \frac{1}{2}\frac{\Delta t}{(\Delta x)^2}C_0 u(t, x-\Delta x) \implies$$

Let

$$\alpha := \frac{\Delta t}{(\Delta x)^2}$$

Then

$$\frac{-1}{2}\alpha u(t+\Delta t, x+\Delta x) + (1+\alpha)u(t+\Delta t, x) - \frac{\alpha}{2}u(t+\Delta t, x-\Delta x) = \frac{\alpha}{2}u(t, x+\Delta x) + (1-\alpha)u(t,x) + \frac{\alpha}{2}u(t, x-\Delta x) \qquad (50)$$

In general

$$\begin{gathered} \frac{-\alpha}{2}C_0 u(t+\Delta t, x+\Delta x) + (1+C_0\alpha)u(t+\Delta t, x) - \frac{C_0\alpha}{2}u(t+\Delta t, x-\Delta x) = \\ = \frac{C_0\alpha}{2}u(t, x+\Delta x) + (1-C_0\alpha)u(t,x) + \frac{C_0\alpha}{2}u(t, x-\Delta x) \end{gathered} \qquad (51)$$

This scheme necessitates a matrix representation. In matrix form,

$$\begin{gathered} \begin{bmatrix} 1+C_0\alpha & -\frac{\alpha C_0}{2} & 0 & \dots & 0 & 0 & \dots & 0 \\ \frac{-C_0\alpha}{2} & 1+C_0\alpha & \frac{-\alpha C_0}{2} & 0 & 0 & \dots & 0 & \\ 0 & \frac{-C_0\alpha}{2} & 1+C_0\alpha & \frac{-C_0\alpha}{2} & 0 & \dots & 0 & \\ & & & \ddots & & & & \\ 0 & 0 & \dots & 0 & \frac{-\alpha}{2} & 1+C_0\alpha & \frac{-\alpha}{2} & \\ 0 & 0 & \dots & 0 & 0 & \frac{-\alpha}{2} & 1+C_0\alpha & \end{bmatrix} u_i^{t+\Delta t} = \\ = \begin{bmatrix} 1-C_0\alpha & \frac{C_0\alpha}{2} & 0 & 0 & 0 & \dots & 0 \\ \frac{C_0\alpha}{2} & 1-C_0\alpha & \frac{C_0\alpha}{2} & 0 & 0 & \dots 0 & \\ 0 & \frac{C_0\alpha}{2} & 1-C_0\alpha & \frac{C_0\alpha}{2} & 0 & \dots & 0 \\ & & & \ddots & & & \\ 0 & 0 & 0 & \dots & \frac{C_0\alpha}{2} & 1-C_0\alpha & \frac{C_0\alpha}{2} \\ 0 & 0 & 0 & \dots & 0 & \frac{C_0\alpha}{2} & 1-C_0\alpha \end{bmatrix} u_i^t \\ \implies \widehat{B}u_i^{t+\Delta t} = \widehat{A}u_i^t \text{ or} \\ u_i^{t+\Delta t} = \widehat{B}^{-1}\widehat{A}u_i^t \end{gathered} \qquad (52)$$

12.2. **Jacobi method, SOR method, for the Laplace and Poisson equation.** 3.1 Poisson's Equation and Relaxation Methods of 410-505 Physics had a good, online, clear explanation of Jacobi method and improvements, namely the Successive Over Relaxation (SOR) method, applied to Laplace and Poisson equation, with clearly labelled diagrams: `http://www.physics.buffalo.edu/phy410-505/2011/topic3/app1/index.html`

13. Call by reference - Call by Value, Call by reference (in C and in C++)

cf. pp. 58, 2.10 Pointers Ch. 2 Scientific Programming in C, Fitzpatrick [6] `printfact3.c`, printfact3.c
pass pointer, pass by reference, call by pointer, call by reference
In C:

- *function prototype* -

$$\begin{tikzcd} \textbf{pointer} \arrow[r, "\text{function}"] \arrow[d, "*"] & \textbf{Types} \\ \textbf{Types} & \end{tikzcd} \qquad \begin{tikzcd} \text{pointer} \arrow[r, mapsto, "\text{function}"] \arrow[d, mapsto, "*"] & \text{void} \\ \text{double} & \end{tikzcd}$$

```
void factorial(double *)
```

where for factorial, it's just your choice of name for *function*.

- *function definition* -

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "\text{function}"] \arrow[d, "*"] & \mathbf{Types} \\
\mathbf{Types} &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{pointer} \arrow[r, "\text{function}"] \arrow[d, "*"] & \text{void} \\
\text{double} &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{fact} \arrow[r, mapsto, "\text{function}"] \arrow[d, mapsto, "*"] & \text{void} \\
*\text{fact} &
\end{tikzcd}$$

$\Longrightarrow$

```
void function(double *fact) { ... }
```

Inside the function definition,

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "*"] \arrow[d, "\&"] & \mathbf{Dat}_{\text{lvalues}} \arrow[r, "\text{typedef}"] & \mathbf{Types} \\
\mathbf{Memory} & &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{fact} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"] & *\text{fact} \arrow[r, mapsto, "\text{typedef}"] & \text{double} \\
\&\text{fact} & &
\end{tikzcd}$$

and so, for instance, in the function definition, you can do things like this:

```
*fact = 1
*fact *= (double) n
```

and so notice that from `*fact = 1`, `*fact` is a lvalue.

  – *function procedure*

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "*"] \arrow[d, "\&"] & \mathbf{Dat}_{\text{lvalues}} \arrow[loop right] & \text{function procedure} \\
\mathbf{Memory} & &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{fact} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"] & *\text{fact} \arrow[loop right] & \text{function procedure} \\
\&\text{fact} & &
\end{tikzcd}$$

$\Longrightarrow$

```
*fact *= (double) n
```

- "Using" the function, function "instantiation", "calling" the function, i.e. "running" the function

$$\begin{tikzcd}
\text{function procedure} & \mathbf{Types} \arrow[loop left] \arrow[r, "\&"] & \mathbf{Memory} \arrow[d, "\cong"] & \\
& & \mathbf{pointers} \arrow[r, "\text{function}"] & \mathbf{Types}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{function procedure} & \text{double} \arrow[loop left] \arrow[r, "\&"] & \text{Memory}(\text{Obj}Memory) \arrow[d, "\cong"] & \\
& & \text{pointer} \arrow[r, "\text{function}"] & \text{void}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{function procedure} & \text{fact} \arrow[loop left] \arrow[r, mapsto, "\&"] & \&\text{fact} \arrow[d, mapsto, "\cong"] & \\
& & \&\text{fact} \arrow[r, mapsto, "\text{function}"] & \text{function}(\&\ \text{fact})
\end{tikzcd}$$

where, again simply note the notation, that we're using *function* and *factorial*, *fact* for *nameofpointer*, interchangeably: see printfact3.c for the example I'm referring to.

Again, *in C*, consider *a pointer to a function* passed to another function as an argument. Take a look at passfunction.c simultaneously.

- *function prototype* -

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \mathbf{Types} \\
\text{Mor}_{\mathbf{Types}} &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{pointer} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[d, mapsto, "*"] & \text{void} \\
\text{Mor}_{\mathbf{Types}}(\text{double}, \text{double}) &
\end{tikzcd}$$

$\Longrightarrow$

```
void hostfunction(double (*)(double))
```

We could further generalize this syntax, simply for syntax and notation sake, as such:

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \mathbf{Types} \\
\text{Mor}_{\mathbf{Types}} &
\end{tikzcd}
\quad
\begin{tikzcd}
\text{pointer} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[d, mapsto, "*"] & \text{data-type} \\
\text{Mor}_{\mathbf{Types}}(\text{typei}, \text{typef}) &
\end{tikzcd}$$

$\Longrightarrow$

```
data-type hostfunction(typef (*)(typei))
```

For practice, consider more than 1 argument in our function, and the other argument, for practice, is a pointer, we're "passing by reference."

$$\begin{tikzcd}
& \mathbf{pointers} \times \mathbf{pointers} \arrow[r, "\text{hostfunction}"] \arrow[dl, "\text{pr}_1"'] \arrow[dr, "\text{pr}_2"] & \mathbf{Types} \\
\mathbf{pointers} \arrow[d, "*"] & & \mathbf{pointers} \arrow[d, "*"] \\
\text{Mor}_{\mathbf{Types}} & & \mathbf{Types}
\end{tikzcd}$$

$$\begin{tikzcd}
& \textit{pointer} \times \textit{pointer} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[dl, mapsto, "\text{pr}_1"'] \arrow[dr, mapsto, "\text{pr}_2"] & \text{void} \\
\textit{pointer} \arrow[d, mapsto, "*"] & & \textit{pointers} \arrow[d, mapsto, "*"] \\
\text{Mor}_{\mathbf{Types}}(\text{double}, \text{double}) & & \text{double}
\end{tikzcd}$$

$\Longrightarrow$

```
void hostfunction( double (*)(double), double *)
```

- *function definition*

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \mathbf{Types} \\
\text{Mor}_{\mathbf{Types}} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{pointer} \arrow[r, "\text{hostfunction}"] \arrow[d, "*"] & \text{void} \\
\text{Mor}_{\mathbf{Types}}(\text{double}, \text{double}) &
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{fun} \arrow[r, mapsto, "\text{hostfunction}"] \arrow[d, mapsto, "*"] & \text{void} \\
*\text{fun} &
\end{tikzcd}$$

$\Longrightarrow$

```
void hostfunction(double (*fun)(double)) { ... }
```

- *Inside* the function definition,

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "*\text{fun}"] & \mathbf{Types} \arrow[r, "="] & \mathbf{Types} \\
\text{double} \arrow[r, "*\text{fun}"] & \text{double} \arrow[r, "="] & \text{double} \\
x \arrow[r, mapsto, "*\text{fun}"] & (*\text{fun})(x) \arrow[r, mapsto, "="] & y = (*\text{fun})(x)
\end{tikzcd}$$

$\Longrightarrow$

```
y = (*fun)(x)
```

- "Using" the function - the *actual* syntax for "passing" a function into a function is interesting (peculiar?): you only need the *name* of the function.

  Let's quickly recall how a function is prototyped, "declared" (or, i.e., defined), and used:

  - *function prototype* -

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "\text{fun1}"] & \mathbf{Types} \\
\text{double} \arrow[r, "\text{fun1}"] & \text{double}
\end{tikzcd}$$

$\Longrightarrow$

```
double fun1(double)
```

  - *function definition* -

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "\text{fun1}"] & \mathbf{Types} \\
\text{double} \arrow[r, "\text{fun1}"] & \text{double} \\
z \arrow[r, mapsto, "\text{fun1}"] & 3.0z * z - z (= 3z^2 - z)
\end{tikzcd}$$

$\Longrightarrow$

```
double fun1(double z) { ... }
```

  - Using function - `fun1(z)`

and so

$$\text{fun1} \in \text{Mor}_{\mathbf{Types}}(\text{double}, \text{double})$$

And so again, it's interesting in terms of syntax that all you need is the *name* of the function to pass into the arguments of the "host function" when using the host function:

$$\begin{tikzcd}
\text{Mor}_{\mathbf{Types}} \arrow[r, "\text{hostfunction}"] & \mathbf{Types} \\
\text{Mor}_{\mathbf{Types}}(\text{double}, \text{double}) \arrow[r, mapsto, "\text{hostfunction}"] & \text{void} \\
\text{fun1} \arrow[r, mapsto, "\text{hostfunction}"] & \text{hostfunction}(\text{fun1})
\end{tikzcd}$$

$\Longrightarrow$

```
hostfunction(fun1)
```

13.0.1. *C++ extensions, or how C++ pass by reference (pass a pointer to argument) vs. C.* Recall how C passes by reference, and look at Fitzpatrick [6], pp. 83-84 for the `square` function:

- *function prototype* -

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "\text{square}"] \arrow[d, "*"] & \mathbf{Types} \\
\mathbf{Types} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{pointer} \arrow[r, mapsto, "\text{square}"] \arrow[d, mapsto, "*"] & \text{void} \\
\text{double} &
\end{tikzcd}$$

$\Longrightarrow$

```
void square(double *)
```

- *function definition* -

$$\begin{tikzcd}
\mathbf{pointers} \arrow[r, "\text{square}"] \arrow[d, "*"] & \mathbf{Types} \\
\mathbf{Types} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{pointer} \arrow[r, "\text{function}"] \arrow[d, "*"] & \text{void} \\
\text{double} &
\end{tikzcd}
\qquad
\begin{tikzcd}
y \arrow[r, mapsto, "\text{square}"] \arrow[d, mapsto, "*"] & \text{void} \\
*y &
\end{tikzcd}$$

$\Longrightarrow$

```
void square(double *y) { ... }
```

Inside the function definition,

$$\begin{tikzcd}
\mathbf{pointer} \arrow[r, "*"] & \mathbf{Dat}_{\text{lvalues}} \arrow[r, "\text{typedef}"] & \mathbf{Types}
\end{tikzcd}
\qquad
\begin{tikzcd}
y \arrow[r, mapsto, "*"] & *y \arrow[r, mapsto, "\text{typedef}"] & \text{double}
\end{tikzcd}$$

and so, for instance, in the function definition, you can do things like this:

```
*y = x*x
```

- "Using" the function, function "instantiation", "calling" the function, i.e. "running" the function

$$\begin{tikzcd}
\text{function procedure} \circlearrowright \mathbf{Types} \arrow[r, "\&"] & \mathbf{Memory} \arrow[d, "\cong"] & \\
& \mathbf{pointers} \arrow[r, "\text{square}"] & \mathbf{Types}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{function procedure} \circlearrowright \text{double} \arrow[r, "\&"] & \text{Memory}(\text{Obj}\mathit{Memory}) \arrow[d, "\cong"] & \\
& \text{pointer} \arrow[r, "\text{square}"] & \text{void}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{function procedure} \circlearrowright \text{res} \arrow[r, mapsto, "\&"] & \&\text{res} \arrow[d, mapsto, "\cong"] & \\
& \&\text{res} \arrow[r, mapsto, "\text{square}"] & \text{square}(\&\text{res})
\end{tikzcd}$$

13.0.2. *C++ syntax for dealing with passing pointers (and arrays) into functions.* However, in *C++*, a lot of the dereferencing $*$ and referencing & is not explicitly said so in the syntax. In this syntax, passing by reference is indicated by prepending the & ampersand to the variable name, in function declaration (prototype and definition). We don't have to explicitly deference the argument in the function (it's done behind the scene) and syntax-wise (it seems), we only have to refer to the argument by regular local name.

Indeed, the syntax appears "shortcutted" greatly:

- *function prototype* -

$$\begin{tikzcd}
\mathbf{pointer} \times \mathbf{Types} \arrow[r, "\text{function}"] & \mathbf{Types}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{pointer}, \text{double} \arrow[r, mapsto, "\text{function}"] & \text{void}
\end{tikzcd}$$

$\Longrightarrow$

```
void function(double &)
```

- *function definition* -

$$\begin{tikzcd}
\mathbf{pointers} \times \mathbf{Types} \arrow[r, "\text{square}"] & \mathbf{Types}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{pointer}, \text{double} \arrow[r, "\text{function}"] & \text{void}
\end{tikzcd}
\qquad
\begin{tikzcd}
\&, y \arrow[r, mapsto, "\text{function}"] & \text{function}(\text{double } \&y)
\end{tikzcd}$$

$\Longrightarrow$

```
void function(double &y) { ... }
```

Inside the function definition,

$$\begin{tikzcd}
\text{double} \arrow[r, "\text{End}(\text{double}, \text{double})"] & \text{double}
\end{tikzcd}
\qquad
\begin{tikzcd}
y \arrow[r, mapsto, "\text{End}(\text{double}, \text{double})"] & y = x * x
\end{tikzcd}$$

and so, for instance, in the function definition, you can do things like this:

```
y = x*x
```

with no deferencing needed.

- "Using" the function, function "instantiation", "calling" the function, i.e. "running" the function

$$\begin{tikzcd}
\mathbf{Types} \arrow[r, "\text{function}"] & \mathbf{Types}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{double} \arrow[r, "\text{function}"] & \text{void}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{res} \arrow[r, mapsto, "\text{function}"] & \text{function}(\text{res})
\end{tikzcd}$$

13.0.3. *C++ note on arrays.* For dealing with arrays, Stroustrup (2013) [9], on pp. 12 of Chapter 1 The Basics, Section 1.8 Pointers, Arrays, and References, does the following:

- *array declaration* -

```
type a[n]; // type[n]; array of n type's
```

- "Using" arrays in function prototypes, i.e. passing into arguments of functions for *function prototypes*

```
data-type function( type * arrayname)
```

- "Using" arrays when "using" functions, i.e. passing into arguments when a function is "called" or "executed"

```
function( arrayname )
```

Fitzpatrick [6] mentions using `inline` for short functions, no more than 3 lines long, because of memory cost of calling a function.

13.0.4. *Need a CUDA, C, C++, IDE? Try Eclipse!* This website has a clear, lucid, and pedagogical tutorial for using Eclipse: Creating Your First C++ Program in Eclipse. But it looks like I had to pay. Other than the well-written tips on the webpage, I looked up stackexchange for my Eclipse questions (I had difficulty with the Eclipse documentation).

Others, like myself, had questions on how to use an IDE like Eclipse when learning CUDA, and "building" (is that the same as compiling?) and running only single files.

My workflow: I have a separate, in my file directory, folder with my github repository clone that's local.

I start a New Project, CUDA Project, in Eclipse. I type up my single file (I right click on the `src` folder and add a 'Source File'). I build it (with the Hammer, Hammer looking icon; yes there are a lot of new icons near the top) and it runs. I can then run it again with the Play, triangle, icon.

I found that if I have more than 1 (2 or more) file in the `src` folder, that requires the `main` function, it won't build right.

So once a file builds and it's good, I, in Terminal, `cp` the file into my local github repository. Note that from there, I could use the `nvcc` compiler to build, from there, if I wanted to.

Now with my file saved (for example, `helloworldkernel.cu`), then I can delete it, without fear, from my, say, `cuda-workplace`, from the right side, "C/C++ Projects" window in Eclipse.

14. On CUDA By Example

Take a look at 3.2.2 A Kernel Call, a Hello World in CUDA C, with a simple kernel, on pp. 23 of Sanders and Kandrot (2010) [10] and on github, [helloworldkernel.cu](https://github.com/ernestyalumni/CompPhys/blob/master/CUDA-By-Example/helloworldkernel.cu). Let's work out the functor interpretation for practice.

- *function definition* -

$$\mathbf{Types} \xrightarrow{\text{kernel}} \mathbf{Types}$$

$$\text{void} \xrightarrow{\text{kernel}} \text{void}$$

where `kernel` $\in$ `__global__`
$\Longrightarrow$

```
__global__ void kernel(void) { }
```

CUDA C adds the `__global__` qualifier to standard C to *alert the compiler that the function*, `kernelfunction`, should be compiled to run on the *device*, not the host (pp. 24 [10]).

- "Using", "calling", "running" function -

$$<<<>>>: (n_{\text{block}}, n_{\text{threads}}) \times \text{kernelfunction} \mapsto \text{kernelfunction} <<< n_{\text{block}}, n_{\text{threads}} >>> \in \text{End}(\text{Dat}_{\mathbf{Types}})$$
$$<<<>>>: \mathbb{N}^+ \times \mathbb{N}^+ \times \text{Mor}_{\text{GPU}} \to \text{End}(\text{Dat}_{GPU})$$

$\Longrightarrow$

```
kernel<<<1,1>>>();
```

cf. 3.2.3 Passing Parameters of Sanders and Kandrot (2010) [10]

Taking a look at [add-passb.cu](https://github.com/ernestyalumni/CompPhys/blob/master/CUDA-By-Example/add-passb.cu), let's work out the functor interpretation of `cudaMalloc`, `cudaMemcpy`.

In `main`, "declaring" a pointer:

```
int *dev_c
```

$\Longleftarrow$

$$\mathbf{pointers} \xrightarrow{*} \mathbf{Dat}_{\text{lvalues}} \xrightarrow{\text{typedef}} \mathbf{Types}$$

$$\texttt{dev\_c} \xmapsto{*} *\texttt{dev\_c} \xmapsto{\text{typedef}} \text{int}$$

We can also do, note, the `sizeof` function (which is a well-defined mapping, for once) on Obj**Types**:

$$\mathbf{pointers} \xrightarrow{*} \mathbf{Dat}_{\text{lvalues}} \xrightarrow{\text{typedef}} \mathbf{Types} \xrightarrow{\text{sizeof}} \mathbb{N}^+$$

$$\texttt{dev\_c} \xmapsto{*} *\texttt{dev\_c} \xmapsto{\text{typedef}} \text{int} \xmapsto{\text{sizeof}} \text{sizeof(int)}$$

Consider what Sanders and Kandrot says about the pointer to the pointer that (you want to) holds the address of the newly allocated memory. [10] Consider this diagram:

$$\mathbf{pointers} \xrightarrow{*} \mathbf{pointers} \xrightarrow{*} \mathbf{Types}$$

$$\mathbf{pointer} \xrightarrow{*} \mathbf{pointer} \xrightarrow{*} \text{void}$$

$$\&\texttt{dev\_c} \xrightarrow{*} *(\&\texttt{dev\_c}) \xrightarrow{*} (\text{void} * *)(\&\texttt{dev\_c})$$

I propose that what `cudaMalloc` does (actually) is the following:

$$\begin{tikzcd}
\mathbf{Memory}_{\text{GPU}} \arrow[r, "\text{cudaMalloc}"] & \mathbf{pointers} \arrow[r, "*"] \arrow[d, "*"] & \mathbf{pointers} \arrow[r, "*"] & \mathbf{Types} \\
& \mathbf{pointers}_{\text{GPU}} \arrow[r, "*"] & \mathbf{Types} &
\end{tikzcd}$$

(53)

$$\begin{tikzcd}
\text{Memory address}_{\text{GPU}} \arrow[r, mapsto, "\text{cudaMalloc}"] & \&\texttt{dev\_c} \arrow[r, mapsto, "*"] \arrow[d, mapsto, "*"] & *(\&\texttt{dev\_c}) \arrow[r, mapsto, "*"] & (\text{void} * *)(\&\texttt{dev\_c}) \\
& \texttt{dev\_c} \arrow[r, mapsto, "*"] & *\texttt{dev\_c} &
\end{tikzcd}$$

`dev_c` is now a *device pointer*, available to kernel functions on the GPU.

Syntax-wise, we can relate this diagram to the corresponding function "usage":

$$\mathbf{pointers} \times \mathbb{N}^+ \xrightarrow{\text{cudaMalloc}} \texttt{cudaError\_r}$$

$$((\text{void} * *)(\&\texttt{dev\_c}), (\text{sizeof(int)})) \xmapsto{\text{cudaMalloc}} \text{cudaSuccess (for example)}$$

$\Longrightarrow$

```
cudaMalloc((void**)&dev_c, sizeof(int))
```

For practice, consider now `cudaMemcpy` in the functor interpretation, and its definition as such:

`cudaMemcpy` is a "functor category", s.t. we equip the functor `cudaMemcpy` with a collection of objects $\text{Obj}_{\text{cudaMemcpy}}$, s.t., for example, `cudaMemcpyDevicetoHost` $\in \text{Obj}_{\text{cudaMemcpy}}$, where

$$(\,\text{cudaMemcpy}(-,-,n_{\text{thread}},\text{cudaMemcpyDevicetoHost}) : \mathbf{Memory}_{GPU} \to \mathbf{Memory}_{CPU}\,) \in \text{Hom}(\mathbf{Memory}_{GPU}, \mathbf{Memory}_{CPU})$$

where $\text{Obj}\mathbf{Memory}_{GPU} \equiv$ collection of all possible memory (addresses) on GPU.

It should be noted that, syntax-wise, $\&c \in \text{Obj}\mathbf{Memory}_{CPU}$ and $\&c$ belongs in the "first slot" of the arguments for cudaMemcpy, whereas `dev_c` $\in \mathbf{pointers}_{GPU}$ a *device pointer*, is "passed in" to the "second slot" of the arguments for cudaMemcpy.

15. Threads, Blocks, Grids

cf. Chapter 5 Thread Cooperation, Section 5.2. Splitting Parallel Blocks of Sanders and Kandrot (2010) [10].

Consider first a 1-dimensional block.

- `threadIdx.x` $\Longleftarrow M_x \equiv$ number of threads per block in x-direction. Let $j_x = 0 \dots M_x - 1$ be the index for the thread. Note that $1 \leq M_x \leq M_x^{\max}$, e.g. $M_x^{\max} = 1024$, max. threads per block
- `blockIdx.x` $\Longleftarrow N_x \equiv$ number of blocks in x-direction. Let $i_x = 0 \dots N_x - 1$
- `blockDim` stores number of threads along each dimension of the block M_x.

Then if we were to "linearize" or "flatten" in this x-direction,

$$k = j_x + i_x M_x$$

where k is the kth thread. $k = 0 \dots N_x M_x - 1$.

Take a look at heattexture1.cu which uses the GPU texture memory. Look at how `threadIdx`/`blockIdx` is mapped to pixel position.

As an exercise, let's again rewrite the code in mathematical notation:

- `threadIdx.x` $\Longleftarrow j_x$, $0 \leq j_x \leq M_x - 1$
- `blockIdx.x` $\Longleftarrow i_x$, $0 \leq i_x \leq N_x - 1$
- `blockDim.x` $\Longleftarrow M_x$, number of threads along each dimension (here dimension x) of a block, $1 \leq M_x \leq M_x^{\max} = 1024$
- `gridDim.x` $\Longleftarrow N_x$, $1 \leq N_x$

resulting in

- $k_x = j_x + i_x M_x \Longrightarrow$

```
int x =  threadIdx.x + blockIdx.x * blockDim.x ;
```

- $k_y = j_y + i_y M_y \Longrightarrow$

```
int y =  threadIdx.y + blockIdx.y * blockDim.y ;
```

and so for a "flattened" thread index $J \in \mathbb{N}$,

$$J = k_x + N_x \cdot M_x \cdot k_y$$

$\Longrightarrow$

```
offset = x + y * blockDim.x * gridDim.x ;
```

Suppose vector is of length N. So we *need* N parallel threads to launch, in total.
e.g. if $M_x = 128$ threads per block, $N/128 = N/M_x$ blocks to get our total of N threads running.

Wrinkle: integer division! e.g. if $N = 127$, $\frac{N}{128} = 0$.

Solution: consider $\frac{N+127}{128}$ blocks. If $N = l \cdot 128 + r$, $l \in \mathbb{N}$, $r = 0 \dots 127$.

$$\frac{N+127}{128} = \frac{l \cdot 128 + r + 127}{128} = \frac{(l+1)128 + r - 1}{128} =$$
$$= l + 1 + \frac{r-1}{128} = \begin{cases} l & \text{if } r = 0 \\ l+1 & \text{if } r = 1 \dots 127 \end{cases}$$
$$\frac{N + (M_x - 1)}{M_x} = \frac{l \cdot M_x + r + M_x - 1}{M_x} = \frac{(l+1)M_x + r - 1}{M_x} =$$
$$= l + 1 + \frac{r-1}{M_x} = \begin{cases} l & \text{if } r = 0 \\ l+1 & \text{if } r = 1 \dots M_x - 1 \end{cases}$$

So $\frac{N+(M_x-1)}{M_x}$ is the smallest multiple of M_x greater than or equal to N, so $\frac{N+(M_x-1)}{M_x}$ **blocks are needed or more than needed to run a total of N threads.**

Problem: Max. grid dim. in 1-direction is 65535, $\equiv N_i^{\max}$.

So $\frac{N+(M_x-1)}{M_x} = N_i^{\max} \Longrightarrow N = N_i^{\max} M_x - (M_x - 1) \leq N_i^{\max} M_x$. i.e. number of threads N is limited by $N_i^{\max} M_x$.
Solution.

- number of threads per block in x-direction $\equiv M_x \Longrightarrow$ `blockDim.x`
- number of blocks in grid $\equiv N_x \Longrightarrow$ `gridDim.x`
- $N_x M_x$ total number of threads in x-direction. Increment by $N_x M_x$. So next scheduled execution by GPU at the $k = N_x M_x$ thread.

Sanders and Kandrot (2010) [10] made an important note, on pp. 176-177 Ch. 9 Atomics of Section 9.4 Computing Histograms, an important *rule of thumb* on the number of blocks.

First, consider N^{threads} total threads. The extremes are either N^{threads} threads on a single block, or N^{threads} blocks, each with a single thread.

Sanders and Kandrot gave this tip:

number of blocks, i.e. `gridDim.x` $\Longleftarrow N_x \sim 2\times$ number of GPU multiprocessors, i.e. twice the number of GPU multiprocessors. In the case of my GeForce GTX 980 Ti, it has 22 Multiprocessors.

15.1. **global thread Indexing: 1-dim., 2-dim., 3-dim.** Consider the problem of *global thread indexing*. This was asked on the NVIDIA Developer's board (cf. Calculate GLOBAL thread Id). Also, there exists a "cheatsheet" (cf. CUDA Thread Indexing Cheatsheet). Let's consider a (mathematical) generalization.

Consider again (cf. 15) the following notation:

- `threadIdx.x` $\Longleftarrow i_x$, $0 \leq i_x \leq M_x - 1$, $\quad i_x \in \{0 \dots M_x - 1\} \equiv I_x$, of "cardinal length/size" of $|I_x| = M_x$
- `blockIdx.x` $\Longleftarrow j_x$, $0 \leq j_x \leq N_x - 1$, $\quad j_x \in \{0 \dots N_x - 1\} \equiv J_x$, of "cardinal length/size" of $|J_x| = N_x$
- `blockDim.x` $\Longleftarrow M_x$
- `gridDim.x` $\Longleftarrow N_x$

Now consider formulating the various cases, of a grid of dimensions from 1 to 3, and blocks of dimensions from 1 to 3 (for a total of 9 different cases) mathematically, as the CUDA Thread Indexing Cheatsheet did, similarly:

- *1-dim. grid* of *1-dim. blocks.* Consider $J_x \times I_x$. For $j_x \in J_x$, $i_x \in I_x$, then $k_x = j_x M_x + i_x$, $k_x \in \{0 \dots N_x M_x - 1\} \equiv K_x$.

  The condition that k_x be a valid global thread index is that K_x has equal cardinality or size as $J_x \times I_x$, i.e.

  $$|J_x \times I_x| = |K_x|$$

  (this must be true). This can be checked by checking the most extreme, maximal, case of $j_x = N_x - 1$, $i_x = M_x - 1$:

  $$k_x = j_x M_x + i_x = (N_x - 1)M_x + M_x - 1 = N_x M_x - 1$$

  and so k_x ranges from 0 to $N_x M_x - 1$, and so $|K_x| = N_x M_x$.

  Summarizing all of this in the following manner:

  $$\begin{tikzcd} J_x \times I_x \arrow[r] & K_x \equiv K^{N_x M_x} = \{0 \dots N_x M_x - 1\} \\ (j_x, i_x) \arrow[r, mapsto] & k_x = j_x M_x + i_x \end{tikzcd}$$

  For the other cases, this generalization we've just done is implied.
- *1-dim. grid* of *2-dim. blocks*

  $$\begin{tikzcd} J_x \times (I_x \times I_y) \arrow[r] & K^{N_x M_x M_y} \equiv \{0 \dots N_x M_x M_y - 1\} \\ (j_x, (i_x, i_y)) \arrow[r, mapsto] & k = j_x M_x M_y + (i_x + i_y M_x) = j_x |I_x \times I_y| + (i_x + i_y M_x) \in \{0 \dots N_x M_x M_y - 1\} \end{tikzcd}$$

  The "most extreme, maximal" case that can be checked to check that the "cardinal size" of $K^{N_x M_x M_y}$ is equal to $J_x \times (I_x \times I_y)$ is the following, and for the other cases, will be implied (unless explicitly written or checked out):

  $$k = j_x M_x M_y + (i_x + i_y M_x) = (N_x - 1)M_x M_y + ((M_x - 1) + (M_y - 1)M_x) = (N_x M_x M_y - 1)$$

The thing to notice is this emerging, general pattern, what could be called a "global view" of understanding the threads and blocks model of the GPU (cf. njuffa's answer:

$$\text{total number of threads} = \text{block index (Id)} \cdot \text{total number of threads per blcok} + \text{thread index on the block}$$

But as we'll see, that's not the only way of "flattening" the index, or transforming into a 1-dimensional index.

- *1-dim. grid* of *3-dim. blocks*

$$\begin{tikzcd} J_x \times (I_x \times I_y \times I_z) \arrow[r] & K^{N_xM_xM_yM_z} \\ (j_x,(i_x,i_y,i_z)) \arrow[r, mapsto] & k = j_x(M_xM_yM_z) + (i_x + i_yM_x + i_ZM_xM_y) \in \{0 \dots N_xM_xM_yM_z - 1\} \end{tikzcd}$$

- *2-dim. grid* of *1-dim. blocks*

$$\begin{tikzcd} (J_x \times J_y) \times I_x \arrow[r] & L^{N_xN_y} \times I_x \arrow[r] & K^{N_xN_yM_x} \\ ((j_x,j_y),i_x) \arrow[r, mapsto] & ((j_x + N_xj_y), i_x) \arrow[r, mapsto] & k = (j_x + N_xj_y)\cdot M_x + i_x \in \{0 \dots N_xN_yM_x - 1\} \end{tikzcd}$$

- *2-dim. grid* of *2-dim. blocks*

$$\begin{tikzcd} (J_x \times J_y) \times (I_x, I_y) \arrow[r] & L^{N_xN_y} \times (I_x, I_y) \arrow[r] & K^{N_xN_yM_x} \\ ((j_x,j_y),(i_x,i_y)) \arrow[r, mapsto] & ((j_x + N_xj_y),(i_x,i_y)) \arrow[r, mapsto] & k = (j_x + N_xj_y)\cdot M_xM_y + i_x + M_xi_y \end{tikzcd}$$

But this *isn't the only way of obtaining* a "flattened index." Exploit the commutativity and associativity of the Cartesian product:

$$\begin{tikzcd} J_x \times J_y \times I_x \times I_y = (J_x \times I_x) \times (J_y \times I_y) \arrow[r] & K^{N_xM_x} \times K^{N_yM_y} \arrow[r] & K^{N_xN_yM_xM_y} \\ ((j_x,j_y,i_x,i_y) = ((j_x,i_x),(j_y,i_y)) \arrow[r, mapsto] & (i_x + M_xj_x, i_y + M_yj_j) \equiv (k_x,k_y) \arrow[r, mapsto] & \begin{array}{c} k = k_x + k_yN_xM_x = \\ = (i_x + M_xj_x) + (i_y + M_yj_y)M_xN_x \end{array} \end{tikzcd}$$

Indeed, checking the "maximal, extreme" case,

$$k = k_x + k_yN_xM_x = M_xN_x - 1 + (M_yN_y - 1)(N_xM_x) = M_yM_yN_xM_x - 1$$

and so k ranges from 0 to $M_yM_yN_xM_x - 1$.

- *3-dim. grid* of *3-dim. blocks*

$$\begin{tikzcd} \begin{array}{c} (J_x \times J_y \times J_z) \times (I_x \times I_y \times I_z) = \\ = (J_x \times I_x) \times (J_y \times I_y) \times (J_z \times I_z) \end{array} \arrow[r] & K^{N_xM_x} \times K^{N_yM_y} \times K^{N_zM_z} \arrow[r] & K^{N_xN_yN_zM_xM_yM_z} \\ \begin{array}{c} ((j_x,j_y,j_z),(i_x,i_y,i_z)) = \\ = ((j_x,i_x),(j_y,i_y),(j_z,i_z)) \end{array} \arrow[r, mapsto] & \begin{array}{c} (i_x + M_xj_x, i_y + M_yj_j, i_z + M_zj_z) \equiv \\ \equiv (k_x,k_y,k_z) \end{array} \arrow[r, mapsto] & k = k_x + k_yN_xM_x + k_zN_xM_xN_yM_y \end{tikzcd}$$

Indeed, checking the "extreme, maximal" case for k:

$$\begin{gathered} k = k_x + k_yN_xM_x + k_zN_xM_xN_yN_y = \\ = (N_xM_x - 1) + (N_yM_y - 1)N_xM_x + (N_zM_z - 1)N_xM_xN_yM_y = N_xN_yN_zM_xM_yM_z - 1 \end{gathered}$$

15.2. **(CUDA) Constant Memory.**

cf. Chapter 6 Constant Memory of Sanders and Kandrot (2010) [10]

Refer to the ray tracing examples in Sanders and Kandrot (2010) [10], and specifically, here: raytrace.cu, rayconst.cu. Without constant memory, then this had to be done:

- *definition* (in the code) - Consider **struct** as a subcategory of **Types** since **struct** itself is a category, equipped with objects and functions (i.e. methods, modules, etc.).

  So for **struct**, Obj**struct** ∋ `Sphere`. $\Longrightarrow$

```
struct sphere { ... }
```

- Usage, "instantiation", i.e. creating, or "making" it (the `struct`):

$$\begin{tikzcd} \textbf{pointers} \arrow[r, "*"] & \textbf{Types} \end{tikzcd}$$

$$\begin{tikzcd} \textbf{pointer} \arrow[r, "*"] \arrow[d, "\&"'] & \textbf{struct} \\ \textbf{Memory}_{CPU} & \end{tikzcd}$$

$$\begin{tikzcd} s \arrow[r, mapsto, "*"] \arrow[d, mapsto, "\&"'] & *s \arrow[r, mapsto, "\text{typedef}"] & \texttt{Sphere} \\ \textbf{Memory}\text{address}_{CPU} & & \end{tikzcd}$$

$\Longrightarrow$

```
Sphere *s
```

Recalling Eq. 53, for `SPHERES` == 40 (i.e. for example, 40 spheres)

```
cudaMalloc((void **) &s, sizeof(Sphere)*SPHERES)
```

$\Longleftarrow$

$$\begin{tikzcd} \textbf{Memory}_{\text{GPU}} \arrow[r, "\text{cudaMalloc}"] & \textbf{pointers} \arrow[r, "*"] \arrow[d, "*"] & \textbf{pointers} \arrow[r, "*"] & \textbf{Types} \\ & \textbf{pointers}_{\text{GPU}} \arrow[r, "*"] & \textbf{Types} & \end{tikzcd}$$

$$\begin{tikzcd} \text{Memory address}_{\text{GPU}} \arrow[r, mapsto, "\text{cudaMalloc}"] & \&s \arrow[r, mapsto, "*"] \arrow[d, mapsto, "*"] & *(\&s) \arrow[r, mapsto, "*"] & (\text{void} **)(\&s) \\ & s \arrow[r, mapsto, "*"] & *s & \end{tikzcd}$$

and syntax-wise,

$$\textbf{pointers} \times \mathbb{N}^+ \xrightarrow{\text{cudaMalloc}} \texttt{cudaError\_r}$$

$$((\text{void} **)(s), \text{sizeof}(\text{Sphere}) * \text{SPHERES}) \xmapsto{\text{cudaMalloc}} \text{cudaSuccess (for example)}$$

Now consider

```
cudaMemcpy(s, temp_s, sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice)
```

$$\mathbf{Memory}_{CPU} \xrightarrow{\text{cudaMemcpy}(s, temps, \text{sizeof}(\text{Sphere}) * \text{SPHERES}, \text{cudaMemcpyHostToDevice})} \mathbf{Memory}_{GPU}$$

$$\begin{tikzcd}
\mathbf{Memory}\text{address}_{CPU} \arrow[r, mapsto] \arrow[d, mapsto, "*"'] & \mathbf{Memory}\text{address}_{GPU} \arrow[d, mapsto, "*"'] \\
\texttt{temp\_s} \arrow[r, mapsto] & s
\end{tikzcd}$$

The lesson then is this, in light of how long ray tracing takes with constant memory and without constant memory - **cudaMemcpy** between host to device, CPU to GPU, is a costly operation. Here, in this case, we're copying from the host memory to memory on the GPU. It copies to a global memory on the GPU.

Now, using **constant memory**,
we no longer need to do `cudaMalloc`, allocate memory on the GPU, for s, pointer to a `Sphere`.

Instead, we have

```
__constant__ Sphere s[SPHERES];
```

In this particular case, we want it to have global scope.
Note, it is still on host memory.

Notice that

$$\mathbf{Memory}_{CPU} \xrightarrow{\text{cudaMemcpyHostToDevice}(-, -, \text{sizeof}(\text{Sphere}) * \text{SPHERES})} \mathbf{Memory}_{GPU}$$

$$\begin{tikzcd}
\mathbf{Memory}\text{address}_{CPU} \arrow[r, mapsto, "\text{cudaMemcpyHostToDevice}(-{,} -{,} \text{sizeof}(\text{Sphere}) * \text{SPHERES})"] \arrow[d, mapsto, "*"'] & \mathbf{Memory}\text{adddress}_{GPU} \arrow[d, mapsto, "*"'] \\
\texttt{temp\_s} \arrow[r, mapsto] \arrow[d, mapsto, "\text{typedef}"'] & s \arrow[d, mapsto, "\text{typedef}"'] \\
\text{array of Sphere's} \arrow[r, mapsto] & \text{array of Sphere's}
\end{tikzcd}$$

So notice that we have a bijection, and on one level, we can think of the bijection from `temp_s`, an array of Sphere's to s, an array of Sphere's. So notice that the types and memory size of `temp_s` and s must match.

And for this case, that's all there is to *constant memory*. What's going on involves the so-called *warp*, a collection of threads, "woven together" and get executed in lockstep. NVIDIA hardware broadcasts a single memory read to each half-warp. "If every thread in a half-warp requests data from the same address in constant memory, your GPU will generate only a single read request and subsequently broadcast the data to every thread." (cf. Sanders and Kandrot (2010) [10]). Furthermore, "the hardware can aggressively cache the constant data on the GPU."

15.3. (CUDA) Texture Memory.

15.4. Do (smooth) manifolds admit a triangulation? Topics in Geometric Topology (18.937)
Piecewise Linear Topology (Lecture 2)

Part 6. Computational Fluid Dynamics (CFD); Computational Methods

16. On Computational Methods for Aerospace Engineering, via Darmofal, Spring 2005

Notes to follow along Darmofal (2005) [11]

16.1. On Lecture 1, Numerical Integration of Ordinary Differential Equations. For the 1-dim. case,

$$m_p \frac{du}{dt} = m_p g - D(u)$$

Recall the velocity vector field $u = u(t,x) \in \mathfrak{X}(\mathbb{R} \times \mathbb{R})$. This is *not* what we want in this case; we want for particles the tangent bundle.

$$\begin{aligned}
D = D(u) &= \frac{1}{2} \rho_g \pi a^2 u^2 C_D(\text{Re}) \\
\text{Re} &= \frac{2\rho_g u a}{\mu_g} \\
C_D &= \frac{24}{\text{Re}} + \frac{6}{1 + \sqrt{\text{Re}}} + 0.4
\end{aligned}$$

Darmofal (2005) [11] then made a brief aside/note on linearization.
Consider perturbation method (linearization)

$$u(t) = u_0 + \widetilde{u}(t)$$

e.g. constant (in time).

If $\frac{du}{dt} = f(u,t)$,

$$\frac{d\widetilde{u}}{dt} = f(u_0 + \widetilde{u}, t) = f(u_0, t) + \left. \frac{\partial f}{\partial u} \right|_{u_0, 0} \widetilde{u} + \left. \frac{\partial f}{\partial t} \right|_{u_0, 0} t + \mathcal{O}(t^2, \widetilde{u}t, \widetilde{u}^2)$$

$a = 0.01\,m$, $\rho_p = 917\,\text{kg}/m^3$ $\qquad$ $\rho_g = 0.9\,\text{kg}/m^3$
$m_p = \rho_p \frac{4}{3}\pi a^3 = 0.0038\,\text{kg}$
$\mu_g = 1.69 \times 10^{-5}\,\text{kg}/(m\,\text{sec})$
$g = 9.8\,m/s^2$
In the 3-dim. case,

$$m_p \frac{d\mathbf{u}}{dt} = m_p g - D(u) \frac{\mathbf{u}}{|u|}$$

Consider curve $\begin{aligned} & x : \mathbb{R} \to N = \mathbb{R} \\ & x(t) \in \mathbb{R} \end{aligned}$, $u(t) \equiv \frac{dx}{dt} \in \Gamma(TN) = \Gamma(T\mathbb{R})$

$$\frac{du}{dt} = g - \frac{D(u)}{m_p}$$

$$\frac{du}{dt} = (u(t + \Delta t) - u(t)) \frac{1}{\Delta t} + \mathcal{O}(\Delta t)$$

16.2. **Multi-step methods generalized.** This subsection corresponds to Lecture 3: Convergence of Multi-Step Methods, but is a further generalization to the presented multi-step methods.

The problem to solve, the ODE to compute out, is

$$\frac{du}{dt}(t) = f(u(t),t) \tag{54}$$

Make the following ansatz:

$$\frac{du}{dt}(t) = \sum_{\nu=0}^N \frac{1}{h} C_\nu u(t-\nu h) = \sum_{\xi=1}^P \beta_\xi f(u(t-\xi h), t-\xi h) \tag{55}$$

Do the Taylor expansion:

$$\sum_{\nu=0}^N \frac{1}{h} C_\nu \left[u(t) + \left(\frac{du}{dt}\right)(t)\cdot(-\nu h) + \sum_{j=2}^n \frac{u^{(j)}(t)}{j!}(-\nu h)^j + \mathcal{O}(h^n) \right] = \sum_{\xi=1}^P \beta_\xi \frac{du}{dt}(t-\xi h) =$$
$$= \sum_{\xi=1}^P \beta_\xi \left[\frac{du}{dt} + \sum_{j=2}^n \frac{u^{(j)}(t)}{j!}(-\xi h)^j + \mathcal{O}(h^n) \right]$$

16.3. **Convection (Discretized).** While I am following Lecture 7 of Darmofal (2005) [11], I will generalize to a "foliated, spatial" (smooth) manifold N, parametrized by time $t \in \mathbb{R}$, $\mathbb{R}\times N$, with $\text{dim}N = n = 1,2$ or 3 and to *CUDA* C/C++ parallel programming.

Consider n-form $m \in \Omega^N(\mathbb{R}\times N)$, $\text{dim}N = n$. Then

$$\begin{aligned} \frac{d}{dt}m = \frac{d}{dt}\int_V \rho \text{vol}^n = \int_V \mathcal{L}_{\frac{\partial}{\partial t}+\mathbf{u}} \rho\text{vol}^n = \int_V \frac{\partial \rho}{\partial t}\text{vol}^n + \mathbf{d}i_{\mathbf{u}}\rho\text{vol}^n = \int_V \left(\frac{\partial\rho}{\partial t} + \text{div}(\rho u)\right)\text{vol}^n = \\ = \int_V \frac{\partial\rho}{\partial t}\text{vol}^n + \int_{\partial V} \rho i_{\mathbf{u}} \text{vol}^n = \dot{m} \end{aligned} \tag{56}$$

where recall

$$\text{div} : \mathfrak{X}(\mathbb{R}\times N) \to C^\infty(\mathbb{R}\times N)$$
$$\text{div}(\rho\mathbf{u}) = \frac{1}{\sqrt{g}}\frac{\partial(\sqrt{g}u^i\rho)}{\partial x^i}$$

16.3.1. *1-dimensional case for Convection from mass (scalar) conservation.* Consider Cell i, between $x_{i-\frac{1}{2}}$ and $x_{i+\frac{1}{2}}$, i.e. $[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}] \subset \mathbb{R}$. In this case, Eq. 56, for mass conservation with sources, becomes

$$\int_V \frac{\partial\rho}{\partial t}\text{vol}^n + \int_{\partial V}\rho i_{\mathbf{u}}\text{vol}^n = \int_V \frac{\partial\rho}{\partial t}dx + \int_{\partial V}\rho u^i = \int_{x_L}^{x_R}\frac{\partial\rho}{\partial t}dx + (\rho(x_R)u(x_R) - \rho(x_L)u(x_L)) = \frac{d}{dt}\int_{x_L}^{x_R}\rho(x)dx$$

In the case of $\frac{d}{dt}m = 0$, on a single cell i,

$$\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \frac{\partial\rho}{\partial t}dx + \rho(x)u(x)|_{x_{i+\frac{1}{2}}} - \rho(x)u(x)|_{x_{i-\frac{1}{2}}} = 0$$

This is one of the first main approximations Darmofal (2005) [11] makes, in Eq. 7.10, Section 7.3 Finite Volume Method for Convection, for the *finite volume method*:

$$\overline{m}_i := \frac{1}{\Delta x_i}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \rho(x)dx \tag{57}$$

where $\Delta x_i \equiv x_{i+\frac{1}{2}} - x_{i+\frac{1}{2}}$.

And so

$$\Delta x_i \frac{\partial}{\partial t}\overline{m}_i + \rho(x)u(x)|_{x_{i+\frac{1}{2}}} - \rho(x)u(x)|_{x_{i-\frac{1}{2}}} = 0 \tag{58}$$

We want to discretize this equation also in time.

Consider as first approximation,

$$\overline{m}(x,t) = \overline{m}_i(t) \qquad \forall\, x_{i-\frac{1}{2}} < x < x_{i+\frac{1}{2}} \tag{59}$$

Consider then initial time t, time step Δt.

16.3.2. *1-dimensional "Upwind" Interpolation for Finite Volume.* This is the "major" approximation for the so-called "Upwind" interpolation approximation:

$$\rho(x_{i+\frac{1}{2}}, t+\Delta t) = \begin{cases} \overline{m}_i(t) & \text{if } u(x_{i+\frac{1}{2}},t) > 0 \\ \overline{m}_{i+1}(t) & \text{if } u(x_{i+\frac{1}{2}},t) < 0 \end{cases} \tag{60}$$

Then use the so-called "forward" time approximation for $\frac{d}{dt}\overline{m}_i(t)$:

$$\Delta x_i \frac{\overline{m}_i(t+\Delta t) - \overline{m}_i(t)}{\Delta t} + (\rho u)(t, x_{i+\frac{1}{2}}) - (\rho u)(t, x_{i-\frac{1}{2}}) = 0$$

Darmofal (2005) [11] didn't make this explicit in Lecture 7, but in the approximation for $\rho(x_{i+\frac{1}{2}}, t+\Delta t)$, Eq. 60, it's supposed that it's valid at time t: $\rho(x_{i+\frac{1}{2}},t) \approx \rho(x_{i+\frac{1}{2}}, t+\Delta t)$, since it's the value of ρ for time moving forward from t (this is implied in Darmofal's code convect1d

$$\rho(x_{i+\frac{1}{2}},t)u(x_{i+\frac{1}{2}},t) = \begin{cases} \overline{m}_i(t)u(x_{i+\frac{1}{2}},t) & \text{if } u(x_{i+\frac{1}{2}},t) > 0 \\ \overline{m}_{i+1}(t)u(x_{i+\frac{1}{2}},t) & \text{if } u(x_{i+\frac{1}{2}},t) < 0 \end{cases}$$

Then

$$\begin{aligned} &\frac{\Delta x_i}{\Delta t}(\overline{m}_i(t+\Delta t) - \overline{m}_i(t)) + \\ &+ \begin{cases} \overline{m}_i(t)u(x_{i+\frac{1}{2}},t) & \text{if } u(x_{i+\frac{1}{2}},t) > 0 \\ \overline{m}_{i+1}(t)u(x_{i+\frac{1}{2}},t) & \text{if } u(x_{i+\frac{1}{2}},t) < 0 \end{cases} - \\ &- \begin{cases} \overline{m}_{i-1}(t)u(x_{i-\frac{1}{2}},t) & \text{if } u(x_{i-\frac{1}{2}},t) > 0 \\ \overline{m}_i(t)u(x_{i-\frac{1}{2}},t) & \text{if } u(x_{i-\frac{1}{2}},t) < 0 \end{cases} = \\ &= 0 \end{aligned} \tag{61}$$

A note on 1-dimensional gridding: Consider total length $L_0 \in \mathbb{R}^+$.
For N^{cells} total cells in x-direction. $i = 0 \dots N^{\text{cells}} - 1$.

$$\begin{aligned} x_{i-\frac{1}{2}} &= i\Delta x & i &= 0,1\dots N^{\text{cells}}-1 \\ x_{i+\frac{1}{2}} &= (i+1)\Delta x & i &= 0,1\dots N^{\text{cells}}-1 \\ x_i &= x_{i-\frac{1}{2}} + \frac{x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}}{2} = \frac{x_{i+\frac{1}{2}} + x_{i-\frac{1}{2}}}{2} = (2i+1)\frac{\Delta x}{2} & i &= 0,1\dots N^{\text{cells}}-1 \end{aligned}$$

At this point, instead of what is essentially the so-called "Upwind Interpolation", which Darmofal is doing in Lecture 7 of Darmofal (2005) [11], and on pp. 76, Chapter 4 Finite Volume Methods, Subsection 4.4.1 Upwind Interpolation (UDS) of Ferziger and Peric (2002) [12], which is essentially a zero-order approximation, let's try to do better.

Consider the interval $[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}] \subset \mathbb{R}$.

For the 1-dimensional case of (pure) convection,

$$\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \frac{\partial\rho(t,x)}{\partial t}dx + \rho(t,x_{i+\frac{1}{2}})u(t,x_{i+\frac{1}{2}}) - \rho(t,x_{i-\frac{1}{2}})u(t,x_{i-\frac{1}{2}}) = \frac{d}{dt}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}}\rho(x)dx$$

Given $\rho(t, x_{i-\frac{1}{2}}), \rho(t, x_{i+\frac{1}{2}}) \in \mathbb{R}$, do (polynomial) interpolation:

$$\begin{gathered} \mathbb{R} \times \mathbb{R} \xrightarrow{\text{interpolation}} \mathbb{R}[x] \equiv \mathcal{P}_{n=1}(\mathbb{R}) \\ \rho(t, x_{i-\frac{1}{2}}), \rho(t, x_{i+\frac{1}{2}}) \mapsto \frac{(x - x_{i-\frac{1}{2}})\rho(t, x_{i+\frac{1}{2}}) - (x - x_{i+\frac{1}{2}})\rho(t, x_{i-\frac{1}{2}})}{h} = \rho_{n=1}(t, x) \end{gathered}$$

where $h \equiv x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}$ and $\mathcal{P}_{n=1}(\mathbb{R})$ is the set of all polynomials of order $n = 1$ over field $\mathbb{R}$ (real numbers).

In general,

$$\begin{gathered} \mathbb{R} \times \mathbb{R} \xrightarrow{\text{interpolation}} \mathbb{R}[x] \equiv \mathcal{P}_{n=1}(\mathbb{R}) \\ \rho(t, x_L), \rho(t, x_R) \mapsto \frac{(x - x_L)\rho(t, x_R) - (x - x_R)\rho(t, x_L)}{(x_R - x_L)} = \rho_{n=1}(t, x) \end{gathered}$$

We interchange the operations of integration and partial derivative - I (correct me if I'm wrong) give two possible reasons why we can do this: the spatial manifold N is fixed in time t, and if the grid cell itself is fixed in time, then the partial derivative in time can be moved out of the integration limits.

So, interchanging $\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} dx$ and $\frac{\partial}{\partial t}$:

$$\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \frac{\partial \rho(t, x)}{\partial t} dx = \frac{\partial}{\partial t} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \rho(t, x) dx$$

So then

$$\Longrightarrow \frac{\partial}{\partial t} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \rho_{n=1}(t, x) = \frac{\partial}{\partial t}(\rho(t, x_{i+\frac{1}{2}}) + \rho(t, x_{i-\frac{1}{2}})) \frac{\Delta x}{2}$$

where $\Delta x = x_{i+\frac{1}{2}} - x_{i+\frac{1}{2}}$.

In general,

$$\frac{\partial}{\partial t} \int_{x_L}^{x_R} \rho_{n=1}(t, x) = \frac{\partial}{\partial t}(\rho(t, x_R) + \rho(t, x_L)) \frac{(x_R - x_L)}{2}$$

Then, discretizing,

$$\tag{62} \begin{gathered} \Longrightarrow \left[(\rho(t + \Delta t, x_{i+\frac{1}{2}}) + \rho(t + \Delta t, x_{i-\frac{1}{2}})) - (\rho(t, x_{i+\frac{1}{2}}) + \rho(t, x_{i-\frac{1}{2}})) \right] \frac{\Delta x}{2} \left(\frac{1}{\Delta t} \right) + \rho(t, x_{i+\frac{1}{2}}) u(t, x_{i+\frac{1}{2}}) - \rho(t, x_{i-\frac{1}{2}}) u(t, x_{i-\frac{1}{2}}) = \\ = \dot{m}_{[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}]}(t) \end{gathered}$$

To obtain $\rho(t, x_{i-\frac{1}{2}})$, consider

$$\frac{\partial \rho}{\partial t} + \text{div}(\rho u) = \frac{d\rho}{dt} = 0$$

which is valid at every point on N.

Consider for $\text{dim}N = 1$,

$$\frac{\partial \rho}{\partial t}(t, x) + \frac{\partial(\rho u)}{\partial x}(t, x)$$

Now, we want $x = x_{i-\frac{1}{2}}$.

Consider

$$\frac{\partial \rho(t, x_{i-\frac{1}{2}})}{\partial t} \approx \frac{\rho(t + \Delta t, x_{i-\frac{1}{2}}) - \rho(t, x_{i-\frac{1}{2}})}{\Delta t}$$

Next, consider the (polynomial) interpolation for the $\frac{\partial(\rho u)}{\partial x}(t, x)$ term:

$$\begin{gathered} \mathbb{R} \times \mathbb{R} \times \mathbb{R} \xrightarrow{\text{interpolate}} \mathbb{R}[x] \equiv \mathcal{P}_{n=2}(\mathbb{R}) \\ \rho(t, x_{i-\frac{3}{2}}) u(t, x_{i-\frac{3}{2}}), \rho(t, x_{i-\frac{1}{2}}) u(t, x_{i-\frac{1}{2}}), \rho(t, x_{i+\frac{1}{2}}) u(t, x_{i+\frac{1}{2}}) \xmapsto{\text{interpolate}} (\rho u)_{n=2}(t, x) \end{gathered}$$

Thus, we can calculate, by plugging into,

$$\frac{\partial(\rho u)_{n=2}(t, x_{i-\frac{1}{2}})}{\partial x}$$

In general, for

$$\begin{gathered} \mathbb{R} \times \mathbb{R} \times \mathbb{R} \xrightarrow{\text{interpolate}} \mathbb{R}[x] \equiv \mathcal{P}_{n=2}(\mathbb{R}) \\ \rho(t, x_{LL}) u(t, x_{LL}), \rho(t, x_L) u(t, x_L), \rho(t, x_R) u(t, x_R) \xmapsto{\text{interpolate}} (\rho u)_{n=2}(t, x) \end{gathered}$$

we have

$$\begin{gathered} \frac{\partial(\rho u)_{n=2}(t, x_L)}{\partial x} = \frac{1}{(x_L - x_{LL})(x_L - x_R)(x_{LL} - x_R)} \cdot \\ \cdot \left((x_L - x_{LL})^2 (\rho u)(x_R) + (x_L - x_{LL})(x_{LL} - x_R)(\rho u)(x_L) - (x_L - x_R)^2 (\rho u)(x_{LL}) + (x_L - x_R)(x_{LL} - x_R)(\rho u)(x_L) \right) \end{gathered}$$

Thus,

$$\tag{63} \Longrightarrow \begin{gathered} \rho(t + \Delta t, x_{i-\frac{1}{2}}) - \rho(t, x_{i-\frac{1}{2}}) + \frac{\partial(\rho u)_{n=2}}{\partial x}(t, x_{i-\frac{1}{2}}) \Delta t = 0 \text{ or} \\ \rho(t + \Delta t, x_{i-\frac{1}{2}}) = \rho(t, x_{i-\frac{1}{2}}) - \frac{\partial(\rho u)_{n=2}}{\partial x}(t, x_{i-\frac{1}{2}}) \Delta t \end{gathered}$$

Now a note on the 1-dimensional grid, "gridding": for cell $i = 0, \dots N^{\text{cell}} - 1$, N^{cell} cells total in the x-direction, then

$$\begin{aligned} x_{i-\frac{1}{2}} &= ih \\ x_{i-\frac{1}{2}} &= (i+1)h \end{aligned}$$

and so $x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}} = h$, meaning the cell width or cell size is h.

Thus, in summary,

$$\rho(t + \Delta t, x_{i-\frac{1}{2}}) = \rho(t, x_{i-\frac{1}{2}}) - (\rho(t, x_{i+\frac{1}{2}}) u(t, x_{i+\frac{1}{2}}) - \rho(t, x_{i-\frac{3}{2}}) u(t, x_{i-\frac{3}{2}})) \left(\frac{1}{2h} \right) \Delta t$$

$$\tag{64} \begin{gathered} \left[(\rho(t + \Delta t, x_{i+\frac{1}{2}}) + \rho(t + \Delta t, x_{i-\frac{1}{2}})) - (\rho(t, x_{i+\frac{1}{2}}) + \rho(t, x_{i-\frac{1}{2}})) \right] \frac{h}{2} \left(\frac{1}{\Delta t} \right) + \rho(t, x_{i+\frac{1}{2}}) u(t, x_{i+\frac{1}{2}}) - \rho(t, x_{i-\frac{1}{2}}) u(t, x_{i-\frac{1}{2}}) = \\ = \dot{m}_{[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}]}(t) \end{gathered}$$

If one was to include Newtonian gravity, consider this general expression for the time derivative of the momentum flux Π:

$$\tag{65} \begin{aligned} \Pi &= \int_{B(t)} \rho u^i \text{vol}^n \otimes e_i \\ \dot{\Pi} &= \int_{B(t)} \frac{\partial(\rho u^i)}{\partial t} \text{vol}^n \otimes e_i + \int_{B(t)} d(\rho u^i i_u \text{vol}^n) \otimes e_i = \int_{B(t)} \frac{\partial(\rho u^i)}{\partial t} \text{vol}^n \otimes e_i + \int_{\partial B(t)} \rho u^i i_u \text{vol}^n \otimes e_i \end{aligned}$$

In 1-dim.,

$$\dot{\Pi} = \int_{B(t)} \frac{\partial(\rho u)}{\partial t} dx + \int_{\partial B} \rho u^2 = \int_B \frac{GMdm}{r^2} = GM \int_B \frac{\rho \text{vol}^n}{r^2} = GM \int_B \frac{\rho dx}{(R - x)^2}$$

Considering a first-order polynomial interpolation for ρ,$\rho_{n=1}$,

$$\frac{\partial}{\partial t}((\rho u)(t, x_{i+\frac{1}{2}}) + (\rho u)(t, x_{i-\frac{1}{2}})) \frac{h}{2} + \rho u^2(t, x_{i+\frac{1}{2}}) - \rho u^2(t, x_{i-\frac{1}{2}}) = GM \int \frac{\rho_{n=2} dx}{(R - x)^2}$$

Note that we need another equation, at $x = x_{i-\frac{1}{2}}$, similar to above:

$$\begin{gathered} \frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2)}{\partial x} = \frac{GM\rho}{(R - x)^2} \\ \Longrightarrow \rho u(t + \Delta t, x_{i-\frac{1}{2}}) - \rho u(t, x_{i-\frac{1}{2}}) + (\rho u^2(t, x_{i+\frac{1}{2}}) - \rho u^2(t, x_{i-\frac{3}{2}})) \left(\frac{1}{2h} \right) \Delta t = \Delta t \int GM \frac{\rho dx}{(R - x)^2} \end{gathered}$$

As a recap, the 1-dimensional setup is as follows:

$$\begin{aligned} \mathbb{R} \times N = \mathbb{R} \times \mathbb{R} & \xrightarrow{\text{discretization}} \mathbb{Z} \times \mathbb{Z} \\ (t, x) & \xmapsto{\text{discretization}} (t_0 + (\Delta t) j, x_{i - \frac{1}{2}} = ih), \qquad i, j \in \mathbb{Z} \end{aligned}$$

Initial conditions for $\rho \in C^\infty(\mathbb{R} \times \mathbb{R})$: $\rho(t_0, x) \in C^\infty(\mathbb{R} \times \mathbb{R})$.

Choices for $u \in \mathfrak{X}(\mathbb{R} \times \mathbb{R})$:

- $u(t, x) = u(x)$ (i.e. time-independent velocity vector field)
- $u(t, x)$ determined by Newtonian gravity (that's an external force on the fluid)

16.3.3. *Note on 1-dimensional gridding.* For, $[0, 1] \subset \mathbb{R}$
N cells,
Then $1/N = \Delta x$. Then consider

$$x_j = j \Delta x \qquad j = 0, 1, \dots N$$

16.4. **2-dim. and 3-dim. "Upwind" interpolation for Finite Volume.** I build on Lecture 7 of Darmofal (2005) [11].
Consider a rectangular grid.

Consider cell C^2_{ij}, $i = 0 \dots N_x - 1$, $j = 0 \dots N_y - 1$. Then there's $N_x \cdot N_y$ total cells, $\begin{matrix} N_x \text{ cells in } x\text{-direction} \\ N_y \text{ cells in } y\text{-direction} \end{matrix}$.

There are 2 possibilities: rectangles of all the same size, with width l^x and length l^t each, or each rectangle for each cell C^2_{ij} is different, of dimensions $l^x_i \times l^y_j$.

Consider cells centered at $x_{2i+1} = l^x \frac{(2i+1)}{2} = \sum_{k=0}^{i-1} l^x_k + \frac{l^2_i}{2}$.
On the "left" sides, $x_{2i} = l^x i = \sum_{k=0}^{i-1} l^x_k$
"right" sides, $x_{2(i+1)} = l^x(i+1) = \sum_{k=0}^{i} l^x_k$.
So cells are centered at

$$(x_{2i+1}, y_{2j+1}) = (l^x \frac{(2i+1)}{2}, l^y \frac{(2j+1)}{2}) = \left(\sum_{k=0}^{i-1} l^x_k + \frac{l^x_i}{2}, \sum_{k=0}^{j-1} l^y_k + \frac{l^y_j}{2} \right)$$

So this cell C^2_{ij}, a 2-(cubic) simplex has 4 1-(cubic) simplices (edges): so 1-(cubic) simplices $\{ C^1_{i \pm 1, j}, C^1_{i, j \pm 1} \}$
The center of these simplices are the following:

$$x_{C^1_{i+1,j}} = (x_{2i+1+1}, y_{2j+1}) = (l^x(i+1), l^y \frac{(2j+1)}{2}) = \left(\sum_{k=0}^{i} l^x_k, \sum_{k=0}^{j-1} l^y_k + \frac{l^y_j}{2} \right) \text{ so then}$$

$$x_{C^1_{i \pm 1,j}} = (x_{2i+1 \pm 1}, y_{2j+1}) = (l^x \left(\frac{2i+1 \pm 1}{2} \right), l^y \frac{(2j+1)}{2}) = \left(\sum_{k=0}^{\frac{2i-1 \pm 1}{2}} l^x_k, \sum_{k=0}^{j-1} l^y_k + \frac{l^y_j}{2} \right)$$

$$x_{C^1_{i,j \pm 1}} = (x_{2i+1}, y_{2j+1 \pm 1}) = (l^x \left(\frac{2i+1}{2} \right), l^y \frac{(2j+1 \pm 1)}{2}) = \left(\sum_{k=0}^{i-1} l^x_k + \frac{l^x_i}{2}, \sum_{k=0}^{\frac{2j-1 \pm 1}{2}} l^y_k \right)$$

We want the flux. So for

$$\overline{\rho}_{ij} := \frac{1}{l^x_i l^y_j} \int_{C^2_{ij}} \rho \text{vol}^2$$

then the flux through 1-(cubic) simplices (faces), $\int \rho i_{\mathbf{u}} \text{vol}^2$,

$$\int_{C^1_{i+1,j}} \rho i_{\mathbf{u}} \text{vol}^2 = \begin{cases} l^y_j \overline{\rho}_{ij} u^x(x_{C^1_{i+1,j}}) & \text{if } u^x(x_{C^1_{i+1,j}}) > 0 \\ l^y_j \overline{\rho}_{i+1,j} u^x(x_{C^1_{i+1,j}}) & \text{if } u^x(x_{C^1_{i+1,j}}) < 0 \end{cases}$$

$$\int_{C^1_{i-1,j}} \rho i_{\mathbf{u}} \text{vol}^2 = \begin{cases} -l^y_j \overline{\rho}_{i-1,j} u^x(x_{C^1_{i-1,j}}) & \text{if } u^x(x_{C^1_{i-1,j}}) > 0 \\ -l^y_j \overline{\rho}_{i,j} u^x(x_{C^1_{i-1,j}}) & \text{if } u^x(x_{C^1_{i-1,j}}) < 0 \end{cases}$$

Likewise,

$$\int_{C^1_{i,j+1}} \rho i_{\mathbf{u}} \text{vol}^2 = \begin{cases} l^x_i \overline{\rho}_{ij} u^y(x_{C^1_{i,j+1}}) & \text{if } u^y(x_{C^1_{i,j+1}}) > 0 \\ l^x_i \overline{\rho}_{i,j+1} u^y(x_{C^1_{i,j+1}}) & \text{if } u^y(x_{C^1_{i,j+1}}) < 0 \end{cases}$$

and so on.

16.4.1. *3-dim. "Upwind" interpolation for finite volume.* For a rectangular prism (cubic),
for cell C^3_{ijk}, $i = 0 \dots N_x - 1$, $j = 0 \dots N_y - 1$, $k = 0 \dots N_z - 1$, $N_x \cdot N_y \cdot N_z$ total cells.
Cells centered at

$$(x_{2i+1}, y_{2j+1}, z_{2j+1}) = (l^x \frac{(2i+1)}{2}, l^y \frac{(2j+1)}{2}, l^z \frac{(2k+1)}{2}) = \left(\sum_{l=0}^{i-1} l^x_l + \frac{l^x_i}{2}, \sum_{l=0}^{j-1} l^y_l + \frac{l^y_j}{2}, \sum_{l=0}^{k-1} l^z_l + \frac{l^y_k}{2} \right)$$

For the 3-(cubic) simplex, C^3_{ijk}, it has 6 2-(cubic) simplices (faces). So for C^3_{ijk}, consider $\{ C^2_{i \pm 1, jk}, C^2_{ij \pm 1, k}, C^2_{ijk \pm 1} \}$.
The center of these faces, such as for $C^2_{i \pm 1, jk}$, $x_{C^2_{i \pm 1, jk}}$, for instance,

$$x_{C^2_{i \pm 1,jk}} = (x_{2i+1 \pm 1}, y_{2j+1}, z_{2k+1}) = (l^x \left(\frac{2i+1 \pm 1}{2} \right), l^y \frac{(2j+1)}{2}, l^z \frac{(2j+1)}{2}) = \left(\sum_{l=0}^{\frac{2i-1 \pm 1}{2}} l^x_l, \sum_{l=0}^{j-1} l^y_l + \frac{l^y_j}{2}, \sum_{l=0}^{l-1} l^z_l + \frac{l^z_k}{2} \right)$$

We want the flux. So for

$$\overline{\rho}_{ijk} := \frac{1}{l^x_i l^y_j l^z_k} \int_{C^3_{ijk}} \rho \text{vol}^3$$

then the flux through 2-(cubic) simplices (faces), $\int \rho i_{\mathbf{u}} \text{vol}^3$,

$$\int_{C^2_{i+1,jk}} \rho i_{\mathbf{u}} \text{vol}^3 = \begin{cases} l^y_j l^z_k \overline{\rho}_{ijk} u^x(x_{C^2_{i+1,jk}}) & \text{if } u^x(x_{C^2_{i+1,jk}}) > 0 \\ l^y_j l^z_k \overline{\rho}_{i+1,jk} u^x(x_{C^2_{i+1,jk}}) & \text{if } u^x(x_{C^2_{i+1,jk}}) < 0 \end{cases}$$

and so on.

To reiterate the so-called "upwind" interpolation method, in generality, recall that we are taking this equation:

$$\int_{C^n_{ij}} \frac{\partial \rho}{\partial t} \text{vol}^n + \int_{\partial C^n_{ij}} \rho u_{\mathbf{u}} \text{vol}^n = \dot{M}_{ij}$$

and discretizing it to obtain

$$\frac{\partial}{\partial t} \overline{\rho}_{ij} |\text{vol}^n| + \int_{\partial C^n_{ij}} \rho i_{\mathbf{u}} \text{vol}^n = \dot{M}_{ij}$$

$$\Longrightarrow \frac{\partial}{\partial t} \overline{\rho}_{ij} = \frac{-1}{|\text{vol}^n|} \int_{\partial C^n_{ij}} \rho i_{\mathbf{u}} \text{vol}^n + \frac{1}{|\text{vol}^n|} \dot{M}_{ij}$$

17. Finite Difference

References/Links that I used:

- Chapter 6 The finite difference method, by Pascal Frey
- Numerical Methods for Partial Differential Equations by Volker John
- *Wikipedia* "Finite Difference". Wikipedia has a section on Difference operators which appears powerful and general, but I haven't understood how to apply it. In fact, see my jupyter notebook on the `CompPhys` github, `finitediff.ipynb` on how to calculate the coefficients in arbitrary (differential) order, and (error) order (of precision, error, i.e. $O(h^p)$) for finite differences, approximations of derivatives.

From `finitediff.ipynb`, I derived this formula

$$f'(x) = \frac{1}{h} \sum_{\nu=1}^3 C_\nu \cdot (f(x+\nu h) - f(x-\nu h)) + \mathcal{O}(h^7) \text{ for}$$

$$\begin{aligned} C_1 &= \frac{3}{4} \\ C_2 &= \frac{-3}{20} \\ C_3 &= \frac{1}{60} \end{aligned} \tag{66}$$

17.1. Finite Difference with Shared Memory (CUDA C/C++).

References/Links that I used:

- Finite Difference Methods in CUDA C++, Part 2, by Dr. Mark Harris
- GPU Computing with CUDA Lecture 3 - Efficient Shared Memory Use, Christopher Cooper of Boston University, August, 2011. UTFSM, Valparaíso, Chile.

cf. Finite Difference Methods in CUDA C++, Part 2, by Dr. Mark Harris

In x-derivative, $\frac{\partial f}{\partial x}$, $\forall$ thread block, $(j_x, j_y) \in \{\{0 \dots N_x - 1\} \times \{0 \dots N_y - 1\}$. $m_x \times s_{\text{Pencils}}$ elements $\in$ tile e.g. $64 \times$sPencils.

In y-derivative, $\frac{\partial f}{\partial y}$, (x,y)-tile of sPencils $\times 64 = s_{\text{Pencils}} \times m_y$.

Likewise, $\frac{\partial f}{\partial z} \to (x,z)-$tile of sPencils $\times 64 = s_{\text{Pencils}} \times m_z$.

Consider for the y derivative, the code for `__global__ void derivative_y(*f, *d_f)` (finitediff.cu):

`int i` $\Longleftarrow i = j_x M_x + i_x \in \{0 \dots N_x M_x - 1\}$ (since $j_x M_x + i_x = (M_x - 1) + (N_x - 1) M_x$, i.e. the "maximal" case)

`int j` $\Longleftarrow j = i_y \in \{0 \dots M_y - 1\}$

`int k` $\Longleftarrow k = j_y \in \{0 \dots N_y - 1\}$

`int si` $\{^{\Longleftarrow si = i_x \in \{0 \dots M_x - 1}\}$

`int sj` $\Longleftarrow sj = j + 4 \in \{4 \dots M_y + 3\}$. Notice that $r = 4$. Then generalize to $s_j = j + r \in \{r, \dots, M_y + r - 1\}$

`int globalIdx` $\Longleftarrow l = k m_x m_y + j m_x + i \in \{0, \dots, (N_y - 1) m_x m_y + (M_y - 1) m_x + N_x M_x - 1\}$ since

$$(N_y - 1) m_x m_y + (M_y - 1) m_X + N_x M_x - 1$$

`s_f[sj][si]` $\Longleftarrow s_f[s_j][s_i] \equiv (s_f)_{s_j, s_i} = f(l)$, $l \in \{0 \dots (N_y - 1) m_x m_y + (M_y - 1) m_x + N_x M_x - 1\}$ with

$$s_f \in \text{Mat}_{\mathbb{R}}(M_y + r, M_x)$$

If $j < 4$, $j < r$, $j = s_j - r \in \{0 \dots r - 1\}$ and so

$$(s_f)_{(s_j - r), s_i} = (s_f)_{s_j + m_y - 1 - r, s_i}$$

$$\Longleftrightarrow$$

$$\{^{\{0, \dots r-1\} \times \{0 \dots M_x - 1\} \leftarrow \{m_y - 1, \dots M_y + m_y - 2\} \times \{0 \dots M_x - 1}\}$$

Then, the actual approximation method:

$$\frac{\partial f}{\partial y}(l) = \frac{\partial f}{\partial y}(i,j,k) = \sum_{\nu=1}^r c_\nu ((s_f)_{s_j + \nu, s_i} - (s_f)_{s_j - \nu, s_i})$$

The shared memory tile here is

`__shared__ float s_f[m_y+8][sPencils]` $\Longleftarrow s_f \in \text{Mat}_{\mathbb{R}}(m_y + 2r, s_{\text{Pencil}})$

By using the shared memory tile, each element from global memory is read only once. (cf. Finite Difference Methods in CUDA C++, Part 2, by Dr. Mark Harris)

Consider expanding the number of pencils in the shared memory tile, e.g. 32 pencils.

Harris says that "with a 1-to-1 mapping of threads to elements where the derivative is calculated, a thread block of 2048 threads would be required." Consider then letting each thread calculate the derivative for multiple points.

So Harris uses a thread block of $32 \times 8 \times 1 = 256$ threads per block, and have each thread calculate the derivative at 8 points, as opposed to a thread block of $4 * 64 * 1 = 256$ thread block, with each thread calculate the derivative at only 1 point.

Perfect coalescing is then regained.

GPU Computing with CUDA Lecture 3 - Efficient Shared Memory Use, Christopher Cooper

17.2. **Note on finite-difference methods on the shared memory of the device GPU, in particular, the pencil method, that attempts to improve upon the double loading of boundary "halo" cells (of the grid).** cf. Finite Difference Methods in CUDA C++, Part 1, by Dr. Mark Harris

Take a look at the code finite_difference.cu. The full code is there. In particular, consider how it launches blocks and threads in the kernel function (and call) `__global__ derivative_x`, `derivative_y`, `derivative_z`. `setDerivativeParameters` has the arrays containing `dim3` "instantiations" that have the grid and block dimensions, for x-,y-,z-derivatives and for "small" and "long pencils". Consider "small pencils" for now. The relevant code is as follows:

```
 grid[0][0]  = dim3(my / sPencils, mz, 1);
  block[0][0] = dim3(mx, sPencils, 1);

  grid[0][1]  = dim3(my / lPencils, mz, 1);
  block[0][1] = dim3(mx, sPencils, 1);

  grid[1][0]  = dim3(mx / sPencils, mz, 1);
  block[1][0] = dim3(sPencils, my, 1);

  grid[1][1]  = dim3(mx / lPencils, mz, 1);
  // we want to use the same number of threads as above,
  // so when we use lPencils instead of sPencils in one
  // dimension, we multiply the other by sPencils/lPencils
  block[1][1] = dim3(lPencils, my * sPencils / lPencils, 1);

  grid[2][0]  = dim3(mx / sPencils, my, 1);
  block[2][0] = dim3(sPencils, mz, 1);

  grid[2][1]  = dim3(mx / lPencils, my, 1);
  block[2][1] = dim3(lPencils, mz * sPencils / lPencils, 1);
```

Let $N_i \equiv$ total number of cells in the grid in the ith direction, $i = x, y, z$. N_i corresponds to `m*` in the code, e.g. N_x is `mx`.

Note that in this code, what seems to be attempted is calculating the derivatives of a 3-dimensional grid, but using only 2-dimensions on the memory of the device GPU. In my experience, with the NVIDIA GeForce GTX 980 Ti, the maximum number of threads per block in the z-direction and the maximum number of blocks that can be launched in the z-direction is severely limited compared to the x and y directions (use `cudaGetDeviceProperties`, or run the code queryb.cu; I find

(67)

```
 Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions:   (2147483647, 65535, 65535)
```

).

Let M_i be the number of threads on a block in the ith direction. Let N_i^{threads} be the total number of threads in the ith direction on the grid, i.e. the number of threads in the ith grid-direction. $i = x, y$. This is *not* the desired grid dimension N_i.

Surely, for a desired grid of size $N_x \times N_y \times N_z \equiv N_x N_y N_z$, then a total of $N_x N_y N_z$ threads are to be computed.

Denote $s_{\text{pencil}} \in \mathbb{Z}^+$ to be `sPencils`; example value is $s_{\text{pencil}} = 4$. Likewise, denote $l_{\text{pencil}} \in \mathbb{Z}^+$ to be `lPencils`; example value is $l_{\text{pencil}} = 32 > s_{\text{pencil}} = 4$.

Then, for instance the small pencil case, for the x-derivative, we have

$$\begin{aligned} &\text{grid dimensions } (N_y/s_{\text{pencil}}, N_z, 1) \\ &\text{block dimensions } (N_x, s_{\text{pencil}}, 1) \end{aligned} \tag{68}$$

Then the total number of threads launched in each direction, x and y, is

$$\begin{aligned} N_x^{\text{threads}} &= \frac{N_y}{s_{\text{pencil}}} N_x \\ N_y^{\text{threads}} &= \frac{N_z}{s_{\text{pencil}}} \end{aligned}$$

While it is true that the total number of threads computed matches our desired grid:

$$N_x^{\text{threads}} \cdot N_y^{\text{threads}} = \frac{N_y}{s_{\text{pencil}}} N_x N_z s_{\text{pencil}} = N_x N_y N_z$$

take a look at the block dimensions that were demanded in Eq. **??**, $(N_x, s_{\text{pencil}}, 1)$. The total number of threads to be launched in this block is $N_x \cdot s_{\text{pencil}}$. Suppose $N_x = 1920$. Then easily $N_x \cdot s_{\text{pencil}} >$ allowed maximum number of threads per block. In my case, this number is 1024.

Likewise for the case of x-direction, but with long pencils. The blocks and threads to be launched on the grid and blocks for the kernel function (`derivative_x`) is

$$\begin{aligned} &\text{grid dimensions } (N_y/l_{\text{pencil}}, N_z, 1) \\ &\text{block dimensions } (N_x, s_{\text{pencil}}, 1) \end{aligned} \tag{69}$$

The total number of threads to be launched in each block is also $N_x \cdot s_{\text{pencil}}$ and for large N_x, this could easily exceed the maximum number of threads per block allowed.

Also, be aware that the shared memory declaration is

```
__shared__ float s_f[sPencils][mx+8]
```

N_x (i.e. `mx`) can be large and we're requiring a 2-dim. array of size $(N_x + 8) * s_{\text{pencil}}$ of floats, for each block. As, from Code listing 67, much more blocks can be launched than threads on a block, and so trying to launch more blocks could possibly be a better solution.

18. Mapping scalar (data) to colors; Data Visualization

Links I found useful:

Taku Komura has good lectures on visualization with computers; it was heavily based on using VTK, but I found the principles and overview he gave to be helpful: here's Lecture 6 Scalar Algorithms: Colour Mapping. (Komura's teaching in the UK, hence spelling "colour")

Good article on practical implementation of a rainbow: `https://www.particleincell.com/2014/colormap/`, i.e. Converting Scalars to RGB Colormap.

I will formulate the problem mathematically (and clearly).

What we want is this, a bijection:

$$\mathbb{R} \longrightarrow [0,255]^3 \equiv RGB$$

$$[0,1) \longrightarrow RGB$$

$$U \subset \mathbb{R} \longrightarrow [0,1) \longrightarrow RGB \tag{70}$$

$$x \in U \longmapsto f = \frac{x - \text{minval}}{\text{maxval - minval}} \longmapsto (r,g,b)$$

with

$$\begin{aligned} \text{minval} &:= \min_{x \in U} x \\ \text{maxval} &:= \max_{x \in U} x \end{aligned} \quad \text{and}$$

$$r,g,b \in [0,\dots,255] \subset \mathbb{Z}$$

Let $n =$ number of "mapping segments" or "segments" (matplotlib terminology). e.g. $n = 5$.

Consider $\frac{1}{n}$, e.g. $\frac{1}{n} = \frac{1}{5} = 0.20$.

Let

$$y := \lfloor 255(nf - \lfloor nf \rfloor) \rfloor \in [0,255] \tag{71}$$

Then

$$(r,g,b) = \begin{cases} (255, y, 0) & \text{if } \lfloor nf \rfloor = 0 \text{ or } 0 \leq nf < 1 \\ (255 - y, 255, 0) & \text{if } \lfloor nf \rfloor = 1 \text{ or } 1 \leq nf < 2 \\ (0, 255, y) & \text{if } \lfloor nf \rfloor = 2 \text{ or } 2 \leq nf < 3 \\ (0, 255 - y, 255) & \text{if } \lfloor nf \rfloor = 3 \text{ or } 3 \leq nf < 4 \\ (y, 0, 255) & \text{if } \lfloor nf \rfloor = 4 \text{ or } 4 \leq nf < 5 \\ (255, 0, 255) & \text{if } \lfloor nf \rfloor = 5 \end{cases} \tag{72}$$

To understand how this color mapping is implemented in `matplotlib`, take a look at

`class matplotlib.colors.LinearSegmentedColormap(name, segmentdata, N=256, gamma=1.0)` of colors - Matplotlib 1.5.3 documentation ,

and look at

row i: x y0 y1

/

/

row i+1: x y0 y1

`http://scipy.github.io/old-wiki/pages/Cookbook/Matplotlib/Show_colormaps` and `http://stackoverflow.com/questions/16834861/create-own-colormap-using-matplotlib-and-plot-color-scale`, for more examples of creating color maps, color bars.

19. On Griebel, Dornseifer, and Neunhoeffer's *Numerical Simulation in Fluid Dynamics: A Practical Introduction*

Griebel, Dornseifer, and Neunhoeffer (1997) [13]

See also Software of Research group of Prof. Dr. M. Griebel, Institute für Numerische Simulation `http://wissrech.ins.uni-bonn.de/research/software/`

19.1. Boundary conditions.

cf. Sec. 2.1. The Mathematical Mode: The Navier-Stokes Equations, Ch. 2 The Mathematical Description of Flows, Griebel, Dornseifer, and Neunhoeffer (1997) [13], pp. 12-13

Let

$$\begin{aligned} \varphi_n &\equiv \text{ component of velocity orthogonal to boundary (in exterior normal direction)} \\ \varphi_t &\equiv \text{ component of velocity parallel to boundary (in tangential direction)} \end{aligned}$$

derivatives in normal direction:

$$\frac{\partial \varphi_n}{\partial n}$$
$$\frac{\partial \varphi_t}{\partial n}$$

Let fixed boundary $\Gamma := \partial\Omega$. Consider

(1) *No-slip condition.*

No fluid penetrates boundary.

fluid is at rest there; i.e.

$$\begin{aligned} \varphi_n(x,y) &= 0 \\ \varphi_t(x,y) &= 0 \end{aligned} \tag{73}$$

(2) *Free-slip condition.*

No fluid penetrates boundary.

Contrary to no-slip condition,

there's no frictional losses at boundary, i.e.

$$\begin{aligned} \varphi_n(x,y) &= 0 \\ \frac{\partial \varphi_t}{\partial n}(x,y) &= 0 \end{aligned} \tag{74}$$

Free-slip condition often imposed along line or plane of symmetry in problem, thereby reducing size of domain, where flow needs to be comprised by half.

19.2. Specific problems and related boundary conditions, boundary specifications.

19.2.1. *Plate an an angle to the inflow.* For

$$\begin{aligned} y_0 &:= a_0 L_y \\ y_1 &:= a_1 L_y \end{aligned}$$

for $0 \leq a_0 < a_1 \leq 1$, e.g. $a_0 = \frac{2}{5}$, $a_1 = \frac{3}{5}$.

Consider an inclined plate to be an obstacle

$$\sum_{i=y_0+1}^{y_1-1} \sum_{j=i-1}^{i+1} \texttt{FLAG}_{ij}$$

Consider $y = Ax + b$, $y, x, A, b \in \mathbb{R}$, and consider

$$\begin{aligned} y &\geq Ax + b_0 \\ y &\leq Ax + b_1 \end{aligned}$$

and so for the 2 points "at the bottom edge of this inclined plate",

$$\begin{gathered}
(y_0+1, y_0+1-1) = (y_0+1, y_0) \\
(y_1-1, y_1-1-1) = (y_1-1, y_1-2) \\
y_0 = A(y_0+1) + b_0 \\
y_1 - 2 = A(y_1-1) + b_0 \\
y_1 - 2 - y_0 = A(y_1 - y_0) - 2A \\
A = \frac{y_1 - y_0 - 2}{y_1 - y_0 - 2} = 1
\end{gathered}$$

$b_0 = -1$.

So $y \geq x - 1$.

Likewise $y \leq x + 1$ (plug in values).

So in parallel, consider $\forall \, i_x = \{ y_0 + 1, y_0 + 2, \dots y_1 - 1 \}$, access memory values at $j = i_x - 1, i_x, i_x + 1$.

Even though the "striding", or stride, for accessing $j = i_x - 1, i_x, i_x + 1$ is $(L_x + 2)$, each of the threads for $\forall \, i_x = \{ y_0 + 1, y_0 + 2, \dots y_1 - 1 \}$ will actually be concurrent.

19.3. **Shared memory tiling scheme applied to the staggered grid; i.e. shared memory tiling scheme for only the "inner cells", excluding halo of radius 1 boundary "cells".** I will first review the shared memory tiling scheme over the entire grid of absolute size $Lx \times Ly$.

For

$$\begin{gathered}
k_x := i_x + j_x M_x \in \{ 0, 1 \dots N_x * M_x - 1 \} \\
k_y := i_y + j_y M_y \in \{ 0, 1 \dots N_y * N_y - 1 \}
\end{gathered}$$

Let

$$\begin{gathered}
S_x := M_x + 2r \in \mathbb{Z}^+ \\
S_y := M_y + 2r \in \mathbb{Z}^+
\end{gathered}$$

Then

$$\begin{aligned}
&\forall \, i \in \{ i = i_x, i_x + M_x, \dots | i_x \leq i < S_x \} \\
&\qquad \forall \, j \in \{ j = i_y, i_y + M_y \dots | i_y \leq j < S_y \},
\end{aligned}$$

$$\begin{gathered}
l_x := i - r + M_x j_x \in \{ -r, -r+1, \dots -r + M_x - 1 \} + M_x j_x \\
l_y := j - r + M_y j_y \in \{ -r, -r+1, \dots -r + M_y - 1 \} + M_y j_y
\end{gathered}$$

and then load input data into shared memory:

$$s_{\text{in}}[i + jS_x] := f^{(c)}(l_x + l_y L_x) \text{ with } \begin{matrix} 0 \leq i < S_x & 0 \leq l_x < L_x \\ 0 \leq j < S_y & 0 \leq l_y < L_y \end{matrix} \tag{75}$$

If $k_x \geq L_x$ or $k_y \geq L_y$, then end or return (check condition).

The actual stencil calculation is performed as follows:

```
$\forall \, \nu_y = 0, 1, \dots W - 1$
    $k_y^{\text{st}} := s_y + \nu_y - r$
    $\forall \, \nu_x := 0, 1, \dots W - 1$
        $k_x^{\text{st}} := s_x + \nu_x - r$
```

$$g^{(c)}(k) = \sum_{\nu_y=0}^{W-1} \sum_{\nu_x=0}^{W-1} c_{\nu = \nu_x + W\nu_y} s_{\text{in}}[k_x^{\text{st}} + k_y^{\text{st}} S_x] \tag{76}$$

with $c_{\nu = \nu_x + W\nu_y} \equiv c(\nu_x, \nu_y)$ and $k := k_x + L_x * k_y$.

Now, we want to deal with applying the shared memory tiling scheme on a staggered grid, so threads aren't launched on the entire (staggered) grid, but only on inner cells.

Examine each step of the loading input data into shared memory procedure above: for $i_x \equiv$ `threadIdx.x`, $i_y \equiv$ `threadIdx.y`,

$$\begin{gathered}
i_x = 0, 1 \dots M_x - 1 \\
i_y = 0, 1 \dots M_y - 1
\end{gathered}$$

and for the `for` loops,

$$\begin{gathered}
i \in \{ i_x, i_x + M_x, \dots | i_x \leq i < S_x := M_x + 2r \} \\
i \in \{ 0, M_x \}, \{ 1, M_x + 1 \}, \dots \{ 2r - 1, M_x + 2r - 1 \}, \{ 2r \}, \{ 2r + 1 \}, \dots \{ M_x - 1 \},
\end{gathered}$$

e.g. for $r = 1$,

$$i \in \{ 0, M_x \}, \{ 1, M_x + 1 \}, \{ 2 \}, \{ 3 \}, \dots \{ M_x - 1 \},$$

and similarly,

$$\begin{gathered}
j \in \{ i_y, i_y + M_y, \dots | i_y \leq j < S_y := M_y + 2r \} \\
j \in \{ 0, M_y \}, \{ 1, M_y + 1 \}, \dots \{ 2r - 1, M_y + 2r - 1 \}, \{ 2r \}, \{ 2r + 1 \}, \dots \{ M_y - 1 \},
\end{gathered}$$

Then, l_x, l_y are defined:

$$\begin{gathered}
l_x := i - r + M_x j_x \in \\
\{ -r + M_x j_x, -r + M_x(j_x + 1) \}, \{ 1 - r + M_x j_x, 1 - r + M_x(j_x + 1) \}, \dots \{ r - 1 + M_x j_x, r - 1 + M_x(j_x + 1) \} \dots \\
\dots \{ r + M_x j_x \}, \{ r + 1 + M_x j_x \}, \dots \{ -1 - r + M_x(j_x + 1) \}
\end{gathered} \tag{77}$$

$$\begin{gathered}
l_y := j - r + M_y j_y \in \\
\{ -r + M_y j_y, -r + M_y(j_y + 1) \}, \{ 1 - r + M_y j_y, 1 - r + M_y(j_y + 1) \}, \dots \{ r - 1 + M_y j_y, r - 1 + M_y(j_y + 1) \} \dots \\
\dots \{ r + M_y j_y \}, \{ r + 1 + M_y j_y \}, \dots \{ -1 - r + M_y(j_y + 1) \}
\end{gathered} \tag{78}$$

Now consider staggered grid:

$$\begin{gathered}
k_x \in \{ 0, 1 \dots N_x M_x - 1 \} \text{ but where } L_x < N_x M_x \leq L_x + M_x - 1 \\
k_y \in \{ 0, 1 \dots N_y M_y - 1 \} \text{ but where } L_y < N_y M_y \leq L_y + M_y - 1
\end{gathered}$$

but the "absolute" indices we want to loop over, for $\| verb \| \in \mathbb{Z}^{(l_x+2)\times(L_y+2)}$ are only `FLAG`'s "inner cells".

$$\begin{gathered}
i := 1, 2 \dots L_x \\
j := 1, 2 \dots L_y
\end{gathered}$$

Note the 1-to-1 correspondence:

$$\begin{gathered}
i = k_x + 1 \\
j = k_y + 1
\end{gathered}$$

To me, at least, I'm bewildered with how to proceed, as this is a difficult problem, so I will begin to check corner cases and easy, simple cases.

Let $r = 1$.

Let $j_x = j_y = 0$.

Then for

$$\begin{gathered}
i \in \{ 0, M_x \}, \{ 1, M_x + 1 \}, \{ 2 \}, \{ 3 \}, \dots \{ M_x - 1 \} \\
j \in \{ 0, M_y \}, \{ 1, M_y + 1 \}, \{ 2 \}, \{ 3 \}, \dots \{ M_y - 1 \}
\end{gathered}$$

and so consider

$$\begin{gathered}
l_x = i + 1 - r + M_x j_x = i + M_x j_x \\
l_y = j + 1 - r + M_y j_y = j + M_y j_y
\end{gathered}$$

Clearly,for $j_x = j_y = 0$, the correct cells are loaded from input $f(l_x + (L_x + 2) l_y) \xleftarrow{\text{flatten}} f(l_x, l_y)$ (note the "special", i.e. particular "striding" or stride of $L_x + 2$ for a staggered grid).

For $J_x = 1$, $j_y = 0$,

$$l_x = i + M_x \in \{M_x, 2M_x\}, \{M_x + 1, 2M_x + 1\}, \{2 + M_x\}, \{3 + M_x\}, \dots \{2M_x - 1\}$$

which accesses $l_x = M_x$, which lies within the "radius" for "halo" cells of the "next" thread block over. By induction, we set

$$\begin{aligned} l_x &:= i + M_x j_x \\ l_y &:= j + M_y j_y \end{aligned}$$

which correctly loads the input array.

Next, we have to do the actual stencil computation.

For "filterwidth" $W = 2r + 1$, consider $\begin{aligned} \nu_x &= 0, 1 \dots W - 1 \\ \nu_y &= 0, 1 \dots W - 1 \end{aligned}$. For $r = 1$, $\nu_x, \nu_y = 0, 1, 2$

We want to consider when $\nu_x, \nu_y = \{0, 2\} = \{0, W - 1\}$, $\forall\, s_x = i_x + r$, $s_y = i_y + r$. So consider

$$\begin{aligned} &s_{\text{in}}(s_x + 0 - r, s_y + 1 - r), \qquad && s_{\text{in}}(s_x + 2 - r, s_y + 1 - r) \\ &s_{\text{in}}(s_x + 1 - r, s_y + 0 - r), \qquad && s_{\text{in}}(s_x + 1 - r, s_y + 2 - r) \end{aligned}$$

and corresponding flags for

```
B_W  B_O
B_S  B_N
```

respectively (Ost is east in Deutsche, German, hence the "O").

cf. 2.2.2. Conservation of Momentum, Ch. 2 The Mathematical Description of Flows, Griebel, Dornseifer, and Neunhoeffer (1997) [13], pp. 16

For incompressible fluids,

$$\rho(\mathbf{x}, t) = \rho_\infty = \text{ const}$$

$$\frac{\partial \mathbf{u}}{\partial t} + u^j \frac{\partial \mathbf{u}}{\partial x^j} + \frac{1}{\rho_\infty} \text{grad} p = \frac{\mu}{\rho_\infty} \boldsymbol{\Delta} \mathbf{u} + \mathbf{g} \tag{79}$$

with

$$\begin{aligned} &\text{dynamic viscosity } \mu \\ &\text{kinematic viscosity } \nu = \frac{\mu}{\rho_\infty} \end{aligned}$$

19.3.1. *Dynamic Similarity of Flows.* cf. 2.3 Dynamic Similarity of Flows, Ch. 2 The Mathematical Description of Flows, Griebel, Dornseifer, and Neunhoeffer (1997) [13], pp. 17

For incompressible flows,

$$\begin{aligned} (x^i)^* &:= \frac{x^i}{L} \\ t^* &:= \frac{u_\infty t}{L} \\ (u^i)^* &:= \frac{u^i}{u_\infty} \\ p^* &:= \frac{p - p_\infty}{\rho_\infty u_\infty^2} \end{aligned} \tag{80}$$

$$\frac{\partial \mathbf{u}^*}{\partial t^*} \left(\frac{u_\infty}{L/u_\infty} \right) + \frac{u_\infty^2}{L} (u^j)^* \frac{\partial \mathbf{u}^*}{\partial (x^j)^*} + \frac{1}{\rho_\infty} \frac{\rho_\infty u_\infty^2}{L} \text{grad}^* p^* = \frac{\mu}{\rho_\infty} \frac{1}{L^2} u_\infty \boldsymbol{\Delta}^* (\mathbf{u})^* + \mathbf{g} \Longrightarrow$$

$$\frac{\partial \mathbf{u}^*}{\partial t^*} + (u^j)^* \frac{\partial \mathbf{u}^*}{\partial (x^j)^*} + \text{grad}^* p^* = \frac{\mu}{\rho_\infty u_\infty L} \boldsymbol{\Delta}^* \mathbf{u}^* + \frac{L}{u_\infty^2} \mathbf{g} \tag{81}$$

$$\begin{aligned} \text{Re} &:= \frac{\rho_\infty u_\infty L}{\mu} \qquad && \text{(Reynolds number)} \\ \text{Fr} &:= \frac{u_\infty}{\sqrt{L \|\mathbf{g}\|}} \qquad && \text{(Froude number)} \end{aligned} \tag{82}$$

Compare this Eq. 81, derived with dynamic similarity, to Eq. (2.2a) of Griebel, Dornseifer, and Neunhoeffer (1997) [13], and my version of the dimensionless momentum conservation equation:

$$\frac{\partial \mathbf{u}}{\partial t} + u^j \frac{\partial \mathbf{u}}{\partial x^j} + \text{grad} p = \frac{1}{\text{Re}} \boldsymbol{\Delta} \mathbf{u} + \mathbf{g}^* \qquad \text{(incompressible)} \tag{83}$$

However, Griebel, Dornseifer, and Neunhoeffer (1997) [13] writes it as

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \frac{\partial u^j \mathbf{u}}{\partial x^j} + \text{grad} p &= \frac{1}{\text{Re}} \boldsymbol{\Delta} \mathbf{u} + \mathbf{g}^* \\ \text{div}(\mathbf{u}) &= 0 \end{aligned} \tag{84}$$

as, separately, a *momentum equation* and *continuity equation.* Griebel, Dornseifer, and Neunhoeffer (1997) [13] ends up implementing this version and so I needed to keep in mind the continuity equation condition.

19.4. **Discretization of the Navier-Stokes Equations.** cf. 3.1.2 Discretization of the Navier-Stokes Equations, Ch. 3 The Numerical Treatment of the Navier-Stokes Equations, Griebel, Dornseifer, and Neunhoeffer (1997) [13], pp. 26

19.4.1. *Gridding (revisited); staggered grid.* Consider again C^2_{ij} a 2-(cubic) simplex.

$$\begin{aligned} i &= 1, 2, \dots L_x \\ j &= 1, 2, \dots L_y \end{aligned} \xrightarrow{\text{Griebel, et.al's notation}} \begin{aligned} i &= 1, 2, \dots i_{\text{max}} \\ j &= 1, 2, \dots j_{\text{max}} \end{aligned}$$

Assume rectangles of all same size, width δx, length δy.

To clarify (or drive home the point), cell (i, j), C^2_{ij} occupies

$$[(i-1)\delta x, i\delta x] \times [(j-1)\delta y, j\delta y] \qquad \forall \begin{aligned} i &= 1, 2, \dots L_x \\ j &= 1, 2, \dots L_y \end{aligned}$$

cell centers:

$$x_{i-0.5, j-0.5} \equiv (x_{i-0.5,j-0.5}) = ((i - 0.5)\delta x, (j - 0.5)\delta y)$$

4 1-(cubic) simplices (edges)

$$\begin{aligned} C^1_{i,j-0.5} &= \{i\delta x\} \times [(j - 0.5)\delta y, j\delta y] \qquad && C^1_{i-0.5,j} = [(i - 0.5)\delta x, i\delta x] \times \{j\delta y\} \\ C^1_{i-1,j-0.5} &= \{(i-1)\delta x\} \times [(j - 0.5)\delta y, j\delta y] \qquad && C^1_{i-0.5,j-1} = [(i - 0.5)\delta x, i\delta x] \times \{(j-1)\delta y\} \end{aligned}$$

(i, j) assigned to pressure at cell center, u^x velocity at right edge, u^y-velocity at upper edge of cell, i.e.

$$\begin{aligned} &p : (i,j) \mapsto p_{i,j} \mapsto ((i - 0.5)\delta x, (j - 0.5)\delta y) \qquad && \forall (i,j) \in \{1, 2, \dots L_x\} \times \{1, 2, \dots L_y\}, \text{ so } (i,j) \mapsto x_{ij} \in \Omega \\ &u^x : (i,j) \mapsto u^x_{i,j} \mapsto (i\delta x, (j - 0.5)\delta y) \qquad && \forall (i,j) \in \{1, 2, \dots L_x\} \times \{1, 2, \dots L_y\}, \text{ so } (i,j) \mapsto x_{ij} \in \Omega \\ &u^y : (i,j) \mapsto u^y_{i,j} \mapsto ((i - 0.5)\delta x, j\delta y) \qquad && \forall (i,j) \in \{1, 2, \dots L_x\} \times \{1, 2, \dots L_y\}, \text{ so } (i,j) \mapsto x_{ij} \in \Omega \end{aligned}$$

where domain Ω cell (i, j), $\begin{aligned} i &= 1 \dots L_x \\ j &= 1 \dots L_y \end{aligned}$. $[(i-1)\delta x, i\delta x] \times [(j-1)\delta y, j\delta y]$.

To reiterate,

$$(i-1, j-0.5) \quad \begin{matrix} (i - 0.5, j) \\ (i - 0.5, j - 0.5) \\ (i - 0.5, j - 1) \end{matrix} \quad (i, j - 0.5) \xleftarrow{\text{discretization}} ((i-1)\delta x, (j - 0.5)\delta y) \quad \begin{matrix} ((i - 0.5)\delta x, j\delta y) \\ ((i - 0.5)\delta x, (j - 0.5)\delta y) \\ ((i - 0.5)\delta x, (j - 1)\delta y) \end{matrix} \quad (i\delta x, (j - 0.5)\delta y)$$

19.4.2. *Discretization of the Spatial Derivatives; Treatment of the Spatial Derivatives.* cf. 3.1.1 Simple Discretization Formulas, Ch. 3 The Numerical Treatment of the Navier-Stokes Equations, Griebel, Dornseifer, and Neunhoeffer (1997) [13].

$$\begin{aligned} &\text{forward difference} && \left[\frac{du}{dx}\right]_i^r := \frac{u(x_{i+1}) - u(x_i)}{\delta x} \\ &\text{backward difference} && \left[\frac{du}{dx}\right]_i^l := \frac{u(x_i) - u(x_{i-1})}{\delta x} \\ &\text{central difference} && \left[\frac{du}{dx}\right]_i^c := \frac{u(x_{i+1}) - u(x_{i-1})}{2\delta x} \end{aligned}$$

19.4.3. *Stability problems, unphysical oscillations.* Upwind difference or upwinding

$$\left[\frac{du}{dx}\right]_i^{\text{up}} := \frac{(1+\epsilon)(u_i - u_{i-1}) + (1-\epsilon)(u_{i+1} - u_i)}{2\delta x} \quad \text{with } \epsilon := \text{sign}(k) \tag{85}$$

cf. Eq. (3.9) Griebel, Dornseifer, and Neunhoeffer (1997) [13]

where k is in $\frac{d^2u}{dx^2} + k\frac{du}{dx} = f$ in Ω

Consider the weighted average of both.

$$\begin{gathered} \gamma \in [0,1] \\ \gamma \cdot \text{ upwind difference } + (1-\gamma) \cdot \text{ central difference} \\ \gamma \cdot \left[\frac{du}{dx}\right]_i^{\text{up}} + (1-\gamma)\left[\frac{du}{dx}\right]_i^c \end{gathered}$$

donor-cell scheme. Consider $\frac{d(ku)}{dx}$, $k \in C^\infty(\mathbb{R})$.

e.g. $\mathbb{R} \xrightarrow{\text{discretization}} \mathbb{Z}$. Consider $x_i \equiv i\delta x$, $i \in \mathbb{Z}$.

Consider $\begin{aligned} k_l &:= k_{i-0.5} \\ k_r &:= k_{i+0.5} \end{aligned}$

So then

$$\left[\frac{d(ku)}{dx}\right]_i^{dc} := \frac{k_r u_r - k_l u_l}{\delta x} \tag{86}$$

cf. Eq. (3.11) Griebel, Dornseifer, and Neunhoeffer (1997) [13]

And so defining

$$\begin{aligned} u_r &:= \begin{cases} u_i & k_r > 0 \\ u_{i+1} & k_r < 0 \end{cases} \\ u_l &:= \begin{cases} u_{i-1} & k_l > 0 \\ u_i & k_l < 0 \end{cases} \end{aligned} \tag{87}$$

can be rewritten as

$$\begin{aligned} \left[\frac{d(ku)}{dx}\right]_i^{dc} &= \frac{1}{2\delta x}((k_r - |k_r|)u_{i+1} + (k_r + |k_r| - k_l + |k_l|)u_i + (-k_l - |k_l|)u_{i-1}) = \\ &= \frac{1}{2\delta x}(k_r(u_i + u_{i+1}) - k_l(u_{i-1} + u_i) + |k_r|(u_i - u_{i+1}) - |k_l|(u_{i-1} - u_i)) \end{aligned} \tag{88}$$

cf. Eq. (3.12), pp. 25, Griebel, Dornseifer, and Neunhoeffer (1997) [13].

Consider terms such as

$$\frac{\partial(u^j\mathbf{u})}{\partial x^j}$$

Take the average:

$$\begin{aligned} \left[\frac{\partial(u^x u^y)}{\partial y}\right]_{i,j} &:= \frac{1}{\delta y}\left(\frac{(u^y_{i,j} + u^y_{i+1,j})}{2}\frac{(u^x_{i,j} + u^x_{i,j+1})}{2} - \frac{(u^y_{i,j-1} + u^y_{i+1,j-1})}{2}\frac{(u^x_{i,j-1} + u^x_{i,j})}{2}\right) \\ \left[\frac{\partial((u^x)^2)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\left(\frac{u_{i,j} + u_{i+1,j}}{2}\right)^2 - \left(\frac{u_{i-1,j} + u_{i,j}}{2}\right)^2\right) \end{aligned} \tag{89}$$

To understand these formulas, **take a look at Fig. 3.6. on pp. 28** of Griebel, Dornseifer, and Neunhoeffer (1997) [13].

Continuity equation.

$$\text{div}\mathbf{u} = 0 \tag{90}$$

cf. Eq. (2.2c)

$$\text{div}\mathbf{u} = 0 \xrightarrow{\text{discretization}} \begin{aligned} \left[\frac{\partial u}{\partial x}\right]_{i,j} &:= \frac{u_{i,j} - u_{i-1,j}}{\delta x} \\ \left[\frac{\partial u^y}{\partial y}\right]_{i,j} &:= \frac{u^y_{i,j} - u^y_{i,j-1}}{\delta y} \end{aligned}$$

Also note this result that'll be used for the Poisson equation describing pressure p:

$$\boxed{\text{divgrad}p = \Delta p = \text{div}\left(-\frac{\partial(u^j\mathbf{u})}{\partial x^j} + \frac{1}{\text{Re}}\Delta\mathbf{u} + \mathbf{g}^*\right)} \tag{91}$$

where

$$\xrightarrow{\text{div}\mathbf{u}} \text{div}\left(\frac{\partial\mathbf{u}}{\partial t}\right) = \frac{\partial}{\partial t}\text{div}\mathbf{u} = 0$$

was used.

19.4.4. *F,G terms (which include central difference, c, and donor-cell, dc, methods).* Page 29 of cf. 3.1.2 Discretization of the Navier-Stokes Equations, Ch. 3 The Numerical Treatment of the Navier-Stokes Equations, Griebel, Dornseifer, and Neunhoeffer (1997) [13] is *gold* for understanding the implementation Griebel, Dornseifer, and Neunhoeffer had used and how to implement the computation or calculationg of F, G.

$\forall$ cell (i,j), $i = 1\dots i_{\text{max}} - 1$, $j = 1\dots j_{\text{max}}$, for u at the midpoint of the *right edge* of the cell, then from Eq. (3.19)a,

$$\begin{aligned} \left[\frac{\partial(u^2)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\left(\frac{u_{ij} + u_{i+1j}}{2}\right)^2 - \left(\frac{u_{i-1j} + u_{ij}}{2}\right)^2\right) + \\ &+\gamma\frac{1}{\delta x}\left(\frac{|u_{ij} + u_{i+1j}|}{2}\frac{(u_{ij} - u_{i+1j})}{2} - \frac{|u_{i-1j} + u_{ij}|}{2}\frac{(u_{i-1j} - u_{ij})}{2}\right) \end{aligned} \tag{92}$$

Indeed, in this case

$$\begin{aligned} k_r &:= \frac{u_{ij} + u_{i+1j}}{2} \\ k_l &:= \frac{u_{i-1j} + u_{ij}}{2} \end{aligned}$$

and Eq. 88 tells us (cf. Eq. 3.12 of Griebel, Dornseifer, and Neunhoeffer (1997) [13]), which again is

$$\left[\frac{d(ku)}{dx}\right]_i^{dc} = \frac{1}{2\delta x}(k_r(u_i + u_{i+1}) - k_l(u_{i-1} + u_i) + |k_r|(u_i - u_{i+1}) - |k_l|(u_{i-1} - u_i))$$

and so in this case

$$\left[\frac{\partial (u^2)}{\partial x} \right]^{dc}_{ij} = \frac{1}{\delta x} \left[\left(\frac{u_{ij} + u_{i+1j}}{2} \right) \left(\frac{u_{ij} + u_{i+1j}}{2} \right) - \left(\frac{u_{i-1j} + u_{ij}}{2} \right) \left(\frac{u_{i-1j} + u_{ij}}{2} \right) + \frac{|u_{ij} + u_{i+1j}|}{2} \frac{(u_{ij} - u_{i+1j})}{2} + \frac{|u_{i-1j} + u_{ij}|}{2} \frac{(u_{i+1j} - u_{ij})}{2} \right] =$$
$$= \left[\frac{\partial (u^2)}{\partial x^2} \right]^{c}_{ij} + \frac{1}{\delta x} \left(\frac{|u_{ij} + u_{i+1j}|}{2} \frac{(u_{ij} - u_{i+1j})}{2} + \frac{|u_{i-1j} + u_{ij}|}{2} \frac{(u_{i+1j} - u_{ij})}{2} \right)$$

Consider Fig. 3.6 on pp. 28 of Griebel, Dornseifer, and Neunhoeffer (1997) [13], “Values required for the discretization of the u-momentum equation.

$$\begin{array}{ccccc} & & u_{ij+1} & & \\ & v_{ij} & & v_{i+1j} & \\ u_{i-1j} & & u_{ij} & & u_{i+1j} \\ & v_{ij-1} & & v_{i+1j-1} & \\ & & u_{ij-1} & & \end{array} \Longrightarrow \begin{array}{ccc} & u^x_{ij+1} & \\ u^x_{i-1j} & \mathbf{u}_{ij} & \mathbf{u}_{i+1j} \\ & \mathbf{u}_{ij-1} & u^y_{i+1j-1} \end{array}$$

$$\begin{array}{ccccc} & & v_{ij+1} & & \\ & u_{i-1j+1} & & u_{ij+1} & \\ v_{i-1j} & & v_{ij} & & v_{i+1j} \\ & u_{i-1j} & & u_{ij} & \\ & & v_{ij-1} & & \end{array} \Longrightarrow \begin{array}{ccc} u^x_{i-1j+1} & \mathbf{u}_{ij+1} & \\ \mathbf{u}_{i-1j} & \mathbf{u}_{ij} & u^y_{i+1j} \\ & u^y_{ij-1} & \end{array}$$

$$\begin{aligned} F &:= (u^x)^n + \delta t \left[\frac{1}{\text{Re}} \Delta u^x - \frac{\partial (u^j u^x)}{\partial x^j} + (g^*)^x \right] \\ G &:= (u^y)^n + \delta t \left[\frac{1}{\text{Re}} \Delta u^y - \frac{\partial (u^j u^y)}{\partial x^j} + (g^*)^y \right] \end{aligned} \tag{93}$$

So. *Compute $u^{(n+1)}$, $v^{(n+1)}$ according to (3.34), (3.35).* cf. pp. 34, 3.2.2 The Discrete Momentum Equations, Griebel, Dornseifer, and Neunhoeffer (1997) [13].

$$u^{(n+1)}_{i,j} = F^{(n)}_{i,j} - \frac{\delta t}{\delta x} (p^{(n+1)}_{i+1,j} - p^{(n+1)}_{i,j}) \qquad \begin{aligned} i &= 1 \dots i_{\max} - 1 \\ j &= 1 \dots j_{\max} \end{aligned} \tag{94}$$

cf. (3.34) Griebel, Dornseifer, and Neunhoeffer (1997) [13]

$$v^{(n+1)}_{i,j} = G^{(n)}_{i,j} - \frac{\delta t}{\delta y} (p^{(n+1)}_{i,j+1} - p^{(n+1)}_{i,j}) \qquad \begin{aligned} i &= 1 \dots i_{\max} \\ j &= 1 \dots j_{\max} - 1 \end{aligned} \tag{95}$$

cf. (3.34) Griebel, Dornseifer, and Neunhoeffer (1997) [13]

with F, G, cf. (3.29), and with

F discretized at right edge of cell (i, j)
G discretized at upper edge of cell (i, j)

which is

$$F_{i,j} := u_{i,j} + \delta t \left(\frac{1}{\text{Re}} \left(\left[\frac{\partial^2 u}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial (u^2)}{\partial x} \right]_{i,j} - \left[\frac{\partial (uv)}{\partial y} \right]_{i,j} + g_x \right) \qquad \begin{aligned} i &= 1 \dots i_{\max} - 1 \\ j &= 1 \dots j_{\max} \end{aligned} \tag{96}$$

cf. Eq. (3.36) Griebel, Dornseifer, and Neunhoeffer (1997) [13], and

$$G_{i,j} := v_{i,j} + \delta t \left(\frac{1}{\text{Re}} \left(\left[\frac{\partial^2 v}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial (uv)}{\partial x} \right]_{i,j} - \left[\frac{\partial (v^2)}{\partial y} \right]_{i,j} + g_y \right) \qquad \begin{aligned} i &= 1 \dots i_{\max} \\ j &= 1 \dots j_{\max} - 1 \end{aligned} \tag{97}$$

cf. Eq. (3.37) Griebel, Dornseifer, and Neunhoeffer (1997) [13].

Consider where the set of cells C_{ij} intersect or overlap that are needed for $u^{n+1}_{i,j}$ and $v^{(n+1)}_{i,j}$.

$$\{(i,j)| \begin{aligned} i &= 1 \dots i_{\max} - 1 \\ j &= 1 \dots j_{\max} - 1 \end{aligned} \} \Longleftrightarrow u^{(n+1)}_{i,j} = ((u^x)^{(n+1)}_{i,j}, (u^y)^{(n+1)}_{i,j})$$

For Eq. (3.36), (3.37) of Griebel, Dornseifer, and Neunhoeffer (1997) [13], which are quoted for the implementation or algorithm,

$$F_{i,j} := u_{i,j} + \delta t \left(\frac{1}{\text{Re}} \left(\left[\frac{\partial^2 u}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial (u^2)}{\partial x} \right]_{i,j} - \left[\frac{\partial (uv)}{\partial y} \right]_{i,j} + g_x \right) \qquad \begin{aligned} i &= 1 \dots i_{\max} - 1 \\ j &= 1 \dots j_{\max} \end{aligned} \tag{98}$$

cf. Eq. (3.36) Griebel, Dornseifer, and Neunhoeffer (1997) [13], and

$$G_{i,j} := v_{i,j} + \delta t \left(\frac{1}{\text{Re}} \left(\left[\frac{\partial^2 v}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial (uv)}{\partial x} \right]_{i,j} - \left[\frac{\partial (v^2)}{\partial y} \right]_{i,j} + g_y \right) \qquad \begin{aligned} i &= 1 \dots i_{\max} \\ j &= 1 \dots j_{\max} - 1 \end{aligned} \tag{99}$$

cf. Eq. (3.37) Griebel, Dornseifer, and Neunhoeffer (1997) [13].

Eq. (3.38) of Griebel, Dornseifer, and Neunhoeffer (1997) [13] is essentially the discretization of the Poisson equation (that takes advantage of the fact that we're dealing with the incompressible case):

$$\Delta p = \frac{1}{\delta t} \text{div}((F, G)) \xrightarrow{\text{discretization}} \tag{100}$$

$$\frac{p^{(n+1)}_{i+1,j} - 2p^{(n+1)}_{i,j} + p^{(n+1)}_{i-1,j}}{(\delta x)^2} + \frac{p^{(n+1)}_{i,j+1} - 2p^{(n+1)}_{i,j} + p^{(n+1)}_{i,j-1}}{(\delta y)^2} = \frac{1}{\delta t} \left(\frac{F^{(n)}_{i,j} - F^{(n)}_{i-1,j}}{\delta x} + \frac{G^{(n)}_{i,j} - G^{(n)}_{i,j-1}}{\delta y} \right) \qquad \begin{aligned} i &= 1 \dots i_{\max} \\ j &= 1 \dots j_{\max} \end{aligned} \tag{101}$$

cf. Eq. (3.38) of Griebel, Dornseifer, and Neunhoeffer (1997) [13], pp. 35, 3.2.3. The Poisson Equation for the Pressure.

19.4.5. *Poisson equation for pressure (for this incompressible case), and towards SOR (successive Over Relaxation) method.*

$$\begin{gathered} \frac{\epsilon^E_i (p^{(n+1)}_{i+1,j} - p^{(n+1)}_{i,j}) - \epsilon^W_i (p^{(n+1)}_{i,j} - p^{(n+1)}_{i-1,j})}{(\delta x)^2} + \frac{\epsilon^N_j (p^{(n+1)}_{i,j+1} - p^{(n+1)}_{i,j}) - \epsilon^S_j (p^{(n+1)}_{i,j} - p^{(n+1)}_{i,j-1})}{(\delta y)^2} = \\ = \frac{1}{\delta t} \left(\frac{F^{(n)}_{i,j} - F^{(n)}_{i-1,j}}{\delta x} + \frac{G^{(n)}_{i,j} - G^{(n)}_{i,j-1}}{\delta y} \right) \\ i = 1 \dots i_{\max} \\ j = 1 \dots j_{\max} \end{gathered} \tag{102}$$

where

$$\epsilon^W_i := \begin{cases} 0 & i = 1 \\ 1 & i > 1 \end{cases}$$

$$\epsilon^E_i := \begin{cases} 1 & i < i_{\max} \\ 0 & i = i_{\max} \end{cases}$$

$$\epsilon^S_j := \begin{cases} 0 & j = 1 \\ 1 & j > 1 \end{cases}$$

$$\epsilon^N_j := \begin{cases} 1 & j < j_{\max} \\ 0 & j = j_{\max} \end{cases}$$

cf. Eq. (3.43) Griebel, Dornseifer, and Neunhoeffer (1997) [13].

e.g., for $1 < i < i_{\max}$, and $1 < j < j_{\max}$,

$$\frac{(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}) - (p_{i,j}^{(n+1)} - p_{i-1,j}^{(n+1)})}{(\delta x)^2} + \frac{(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}) - (p_{i,j}^{(n+1)} - p_{i,j-1}^{(n+1)})}{(\delta y)^2} = \\ = \frac{1}{\delta t} \left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y} \right)$$

Indeed it takes into account the discretized boundary conditions, which was obtained from the so-called *Chorin projection method* (which is based on the Hodge-Helmholtz decomposition).

$$\begin{aligned} p_{0,j} &= p_{i,j}; & p_{imax+1,j} &= p_{imax,j}, & j &= 1 \dots j_{\max} \\ p_{i,0} &= p_{i,1}; & p_{i,jmax+1} &= p_{i,jmax}, & i &= 1 \dots i_{\max} \end{aligned} \tag{103}$$

cf. Eq. (3.41) Griebel, Dornseifer, and Neunhoeffer (1997) [13].

19.4.6. *Successive Over Relaxation (SOR) method.* $\forall \, it = 1 \dots it_{\max}$

$i = 1 \dots i_{\max}$,

$j = 1 \dots j_{\max}$

$$\begin{gathered} p_{i,j}^{it+1} := (1-\omega) p_{i,j}^{it} + \\ + \frac{\omega}{\left(\frac{\epsilon_i^E + \epsilon_i^W}{(\delta x)^2} + \frac{\epsilon_j^N + \epsilon_j^S}{(\delta y)^2} \right)} \cdot \left(\frac{\epsilon_i^E p_{i+1,j}^{it} + \epsilon_i^W p_{i-1,j}^{it+1}}{(\delta x)^2} + \frac{\epsilon_j^N p_{i,j+1}^{it} + \epsilon_j^S p_{i,j-1}^{it+1}}{(\delta y)^2} - \text{rhs}_{ij} \right) \end{gathered} \tag{104}$$

cf. Eq. (3.44) Griebel, Dornseifer, and Neunhoeffer (1997) [13].

19.4.7. *Implementation, routines, algorithms for incompressible Navier-Stokes equations solver.* Consider the serial version of the incompressible Navier-Stokes equations solver with finite difference:

```
• $t := 0$, $n := 0$
• initial values of $u, v, p$ i.e. $\mathbf{u}, p$
• While $t < t_{\text{end}}$
    – select $\delta t$ (according to (3.50) if stepsize control is used)
    – set boundary values for $u, v$
    – Compute $F^{(n)}$ and $G^{(n)}$ according to (3.36), (3.37)
    – Compute RHS of pressure Eq. (3.38)
    – Set $it := 0$
    – While $it < it_{\max}$ and $\|r^{it}\| >$ eps (resp., $\|r^{it}\| >$ eps $\|p^0\|$)
        ∗ Perform an SOR cycle according to (3.44)
        ∗ Compute the residual norm for the pressure equation, $\|r^{it}\|$
        ∗ $it := it + 1$
    – Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to (3.34), (3.35)
    – $t := t + \delta t$
    – $n := n + 1$
```

Algorithm 1. Base version (or serial version), pp. 40, of Griebel, Dornseifer, and Neunhoeffer (1997) [13].

Compare this with the parallelized version:

```
• $t := 0$, $n := 0$
• initial values of $u, v, p$ i.e. $\mathbf{u}, p$
• While $t < t_{\text{end}}$
    – select $\delta t$ (according to (3.50) if stepsize control is used)
    – set boundary values for $u, v$
    – Compute $F^{(n)}$ and $G^{(n)}$ according to (3.36), (3.37)
    – Compute RHS of pressure Eq. (3.38)
    – Set $it := 0$
    – While $it < it_{\max}$ and $\|r^{it}\| >$ eps (resp., $\|r^{it}\| >$ eps $\|p^0\|$)
        ∗ Perform an SOR cycle according to (3.44)
        ∗ Exchange the pressure values in the boundary strips
        ∗ Compute the partial residual and send these to the master process
        ∗ Master process computes the residual norm of the pressure equation $\|r^{it}\|$ and broadcasts it to all processes
        ∗ $it := it + 1$
    – Compute (update) $u^{(n+1)}$ and $v^{(n+1)}$ according to (3.34), (3.35)
    – $t := t + \delta t$
    – $n := n + 1$
```

Algorithm 3. Parallel version, pp. 115 of Griebel, Dornseifer, and Neunhoeffer (1997) [13]

20. Solving Poisson equation's by the preconditioned conjugate gradient method

The Poisson equation comes into play when determining (solving for) the pressure p in the incompressible Navier-Stokes equations for fluid flow. Let us review how this comes about.

Recall the full incompressible Navier-Stokes equations, which comes from momentum conservation, and mass conservation, respectively:

$$\begin{gathered} \frac{\partial \mathbf{u}}{\partial t} + \frac{\partial u^j \mathbf{u}}{\partial x^j} + \text{grad} p = \frac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{g}^* \\ \text{div} \mathbf{u} = 0 \end{gathered} \tag{105}$$

Using the incompressibility condition

$$\text{div} \mathbf{u} = 0$$

then exchanging the partial derivative over time with divergence div (we should be able to do this if there is a foliation over time $t \in \mathbb{R}$, i.e. "time-slices" of (spatial) Riemannian manifold N):

$$\text{div} \frac{\partial \mathbf{u}}{\partial t} = \frac{\partial}{\partial t} \text{div} \mathbf{u} = 0 \tag{106}$$

and so applying the divergence div to Eq. 105, we obtain

$$\text{div} \text{grad} p = \Delta p = \text{div} \left(\frac{-\partial (u^j \mathbf{u})}{\partial x^j} + \frac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{g}^* \right) \tag{107}$$

Now, considering the right-hand side (RHS) of Eq. 107, we effectively add 0 via the incompressible condition $\text{div} \mathbf{u} = 0$:

$$\begin{gathered} \text{div} \left(\frac{-\partial (u^j \mathbf{u})}{\partial x^j} + \frac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{g}^* \right) = \text{div} \left(\frac{-\partial (u^j \mathbf{u})}{\partial x^j} + \frac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{g}^* \right) + \text{div} \frac{\partial \mathbf{u}}{\partial t} = \text{div} \left(\frac{-\partial (u^j \mathbf{u})}{\partial x^j} + \frac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{g}^* + \frac{\partial \mathbf{u}}{\partial t} \right) = \\ =: \text{div} \left(\frac{\partial}{\partial t} (F, G) \right) = \frac{\partial}{\partial t} \text{div}(F, G) \end{gathered}$$

where we've defined $F, G \in C^\infty(\mathbb{R} \times N)$:

$$\begin{gathered} F = F(t, x) \in C^\infty(\mathbb{R} \times N) \\ G = G(t, x) \in C^\infty(\mathbb{R} \times N) \\ \text{where} \\ \frac{-\partial (u^j \mathbf{u})}{\partial x^j} + \frac{1}{\text{Re}} \Delta \mathbf{u} + \mathbf{g}^* + \frac{\partial \mathbf{u}}{\partial t} =: \frac{\partial}{\partial t} (F, G) \end{gathered} \tag{108}$$

to be components of this time-dependent vector field over N. I've only defined 2 components F, G for the 2-dimensional case, but easily, arbitrarily finite number of components can be defined.

This F, G is the "continuous" version of the discretized F, G in Griebel, Dornseifer, and Neunhoeffer (1997) [13].

Thus, we see how the Poisson equation comes into play for incompressible Navier-Stokes equations, in order to solve for the pressure p:

$$\Delta p = \frac{\partial}{\partial t} \text{div}(F,G) \tag{109}$$

Let's compare this equation with the general form of the Poisson equation, for $p, f \in C^\infty(N)$, for (Riemannian) manifold N (e.g. $N = \mathbb{R}^2, \mathbb{R}^3$):

$$\text{divgrad}p = \Delta p = f \tag{110}$$

Let's drop the time-dependence (because we sought to consider the quantities of p and f at a specific point in time t, and not their changes in time, we can do this) and focus upon solving the Poisson equation..

For $p \in C^\infty(N)$, consider how the (second order) central (finite) difference discretization of the Laplacian leads to a so-called "sparse" matrix.

M. Ament, G. Knittel, D. Weiskopf, W. Straßer. *A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform* (2010) http://www.vis.uni-stuttgart.de/~amentmo/docs/ament-pcgip-PDP-2010.pdf

20.0.8. *Lid-Driven Cavity.* cf. 5.1. Lid-Driven Cavity. Ch. 5 Example Applications, Griebel, Dornseifer, and Neunhoeffer (1997) [13].

On $S_N \subset \partial\Omega \subset \Omega \subset \mathbb{R}^2$ ("northern" N boundary of domain Ω, which is a submanifold of Ω or of $\mathbb{R}^2$,

$$\begin{aligned} u^x(x) &= \overline{u} \\ u^y(x) &= 0 \end{aligned} \qquad \forall\, x \in S_N \tag{111}$$

For the *discretization*, consider the staggered grid and how u^x or i.e. the x-component of velocity field $\mathbf{u}$, u, is at the center of each edge (or i.e. face), and by convention, the cell index (i,j) corresponds to the "right" face. So

$$\begin{array}{ccc} u^x_{i-1,j_{\max}+1} & & u^x_{i,j_{\max}+1} \\ & u^x_{i-\frac{1}{2},j_{\max}+\frac{1}{2}} & \\ u^x_{i-1,j_{\max}} & & u^x_{i,j_{\max}} \end{array}$$

and so

$$\begin{gathered} u^x_{i,j_{\max}+\frac{1}{2}} = \frac{u^x_{i,j_{\max}+1} + u^x_{i,j_{\max}}}{2} = \overline{u} \text{ or} \\ u^x_{i,j_{\max}+1} = 2\overline{u} - u^x_{i-1,j_{\max}}; \qquad i = 1 \dots i_{\max} \end{gathered} \tag{112}$$

Part 7. Finite Element; Finite Element Method, Finite Element Analysis; Finite Element Exterior Calculus

cf. Lecture 11 "Method of Weighted Residuals", Darmofal (2005) [11]

cf. Ch. 23 of "Electrostatics via Finite Elements" from Landau, Páez, and Bordeianu (2015) [16].

$$\phi_i(x) = \begin{cases} 0 & \text{for } x < x_{i-1} \text{ or } x > x_{i+1} \\ \frac{x - x_{i-1}}{h_{i-1}} & \text{for } x_{i-1} \leq x \leq x_i \\ \frac{x_{i+1} - x}{h_i} & \text{for } x_i \leq x \leq x_{i+1} \end{cases} \tag{113}$$

Consider

$$*\mathbf{d}\phi(x) \in \Omega^{d-1}(N)$$

e.g. $d = 3$, $d - 1 = 2$.

Consider $\omega \in \Omega^p(N)$, in general, $f \in C^\infty(N)$, in general.

$$\begin{aligned} \int_\Omega d(f\omega) = \int_\Omega d(f\omega_{i_1 \dots i_p} dx^{i_1} \wedge \dots \wedge dx^{i_p}) = \int_\Omega \frac{\partial(f\omega_{i_1\dots i_p})}{\partial x^d} dx^j \wedge dx^{i_1} \wedge \dots \wedge dx^{i_p} = \int_\Omega (d\omega) f + \int_\Omega (df) \wedge \omega = \\ = \int_{\partial\Omega} f\omega \end{aligned}$$

$\text{dim}\Omega = p + 1$.

$$\implies \int_\Omega f(d\omega) + \int_\Omega (df) \wedge \omega = \int_{\partial\Omega} f\omega \tag{114}$$

So if $\Omega = -*\mathbf{d}U$; $U \in C^\infty(N)$, $E = -dU \in \Omega^1(N) = \Gamma(T^*N)$, $f = \Phi$

$$-\int_\Omega (\mathbf{d}{*}\mathbf{d})U(x)\Phi(x) = -\int_{\partial\Omega} (*\mathbf{d}U)\Phi(x) + \int_\Omega \mathbf{d}\Phi \wedge *\mathbf{d}U = \int_\Omega 4\pi\rho(x)\Phi\text{vol}^d$$

Writing out the components, which is true in full generality,

$$\begin{aligned} \mathbf{d}U &= \frac{\partial U}{\partial x^j} dx^j \\ *\mathbf{d}U &= \frac{\sqrt{g}}{(d-1)!} \epsilon_{i_1,i_2\dots i_d} g^{ij} \frac{\partial U}{\partial x^j} dx^{i_2} \wedge \dots \wedge dx^{i_d} \end{aligned}$$

Then, insightfully,

$$\mathbf{d}\Phi \wedge *\mathbf{d}U = \frac{\partial\Phi}{\partial x^{i_1}} g^{i_1 j} \frac{\partial U}{\partial x^j} \text{vol}^d \equiv \langle \text{grad}\Phi, \text{grad}U \rangle$$

$$\implies -\int_{\partial\Omega} (*\mathbf{d}U)\Phi(x) + \int_\Omega \mathbf{d}\Phi \wedge *\mathbf{d}U = -\int_{\partial\Omega} (*\mathbf{d}U)\Phi(x) + \int_\Omega \langle \text{grad}\Phi, \text{grad}U \rangle = \int_\Omega 4\pi\rho(x)\Phi\text{vol}^d \tag{115}$$

$\forall\, l = 0, 1, \dots N - 1$

$$\int_\Omega 4\pi\rho(x)\phi_l(x)\text{vol}^d = \int_{x_{i-1}}^{x_i} 4\pi\rho(x) \frac{x - x_{i-1}}{h_{i-1}} dx + \int_{x_i}^{x_{i+1}} 4\pi\rho(x) \frac{x_{i+1} - x}{h_i} dx$$

cf. Lecture 11, Method of Weighted Residuals, `lect11.pdf`, Darmofal (2005) [11].
From heat equation,

$$\text{div}(k\text{grad}T) = -q \tag{116}$$

or in components

$$\frac{1}{\sqrt{g}} \frac{\partial}{\partial x^k} \left(\sqrt{g} k \frac{\partial T}{\partial x^l} g^{kl} \right) = -q$$

In 1-dim.,

$$\frac{\partial}{\partial x^k} \left(k \frac{\partial T}{\partial x^k} \right) = -q$$

over $\Omega = \left[\frac{-L}{2}, \frac{L}{2} \right]$.

e.g. Ex. 11.1 (Steady heat diffusion),
Suppose $L = 2$,
thermal conductivity $k = 1$,
heat source $q(x) = 50e^x$.
$T(\pm 1) = 100$.

Integrate twice:

$$\begin{aligned} kT'' &= -50e^x \\ T' &= \frac{-50e^x}{k} + A_0 \\ T &= \frac{-50e^x}{k} + A_0 x + B_0 \end{aligned}$$

$$\begin{aligned} \frac{-50e}{k} + A_0 + B_0 = 100 \\ \frac{-50e^{-1}}{k} - A_0 + B_0 = 100 \\ B_0 = 100 + \frac{50}{k}\cosh 1 \\ A_0 = \frac{50}{k}\sinh 1 \end{aligned}$$

$$T = \frac{-50e^x}{k} + \frac{50}{k}\sinh 1 x + 100 + \frac{50}{k}\cosh 1$$

A common approach is to approximate the solution with a series of weighted functions.

cf. 11.2. The Method of Weighted Residuals of Darmofal (2005) [11], Lecture 11, `lect11.pdf`.

Consider

$$\int_{-1}^{1} w(x)R(\overline{T},x)dx$$

Choose N weight functions $w_j(x)$ for $1 \leq j \leq N$,
and setting N weighted residuals to 0

$$R_j(\overline{T}) = \int_{-1}^{1} w_j(x)R(\overline{T},x)dx \equiv \text{ weighted residual for } w_j \tag{117}$$

determine appropriate weight functions,
Galerkin method is to set weight functions euqal to functions used to approximate solution

$$w_j(x) = \phi_j(x) \qquad \text{(Galerkin)}$$

e.g. 1-dim. heat diffusion:

$$\begin{aligned} w_1(x) &= (1-x)(1+x) \\ w_2(x) &= x(1-x)(1+x) \end{aligned}$$

$$\begin{aligned} R_1(\overline{T}) &= \int_{-1}^{1} w_1(x)R(\overline{T},x)dx = \int_{-1}^{1}(1-x)(1+x)(-2\alpha_1 - 6\alpha_2 x + 50e^x)dx = \frac{-8}{3}\alpha_1 + 200e^{-1} = 0 \\ R_2(\overline{T}) &= \int_{-1}^{1} w_2(x)R(\overline{T},x)dx = \int_{-1}^{1} x(1-x)(1+x)(-2\alpha_1 - 6\alpha_2 x + 50e^x)dx = \frac{-8}{3}\alpha_2 + 100e^{-1} - 1200e^{-1} = 0 \end{aligned}$$

and so, I would say (EY : 20170204)

$$R_j(\overline{T}) = 0 \mapsto \alpha_j \tag{118}$$

cf. 12.3. 1-D Linear Elements and the Nodal Basis of Darmofal (2005) [11], Lecture 12, The Finite Element Method for One-Dimensional Diffusion `lect12.pdf`.

20.0.9. *nodal basis.* N elements, $N+1$ degrees of freedom.

$$\widetilde{T}(x) = \sum_{i=1}^{N+1} a_i\phi_i(x) \text{ or } \widetilde{T}(x) = \sum_{i=0}^{N} a_i\phi_i(x)$$

$$\text{if } \widetilde{T}(x) = \sum_{i=1}^{N+1}\widetilde{T}(x_i)\phi_i(x) \text{ or } \widetilde{T}(x) = \sum_{i=0}^{N}\widetilde{T}(x_i)\phi_i(x)$$

iff

$$\widetilde{T}(x_j) = \sum_{i=0}^{N} a_i\phi_i(x_j) = \sum_{i=0}^{N} a_i\delta_{ij} = a_j$$

Now

$$\phi_i(x) = \begin{cases} 0 & \text{for } x < x_{i-1} \\ \frac{x-x_{i-1}}{\Delta x_{i-1}} & \text{for } x_{i-1} < x < x_i \\ \frac{x_{i+1}-x}{\Delta x_i} & \text{for } x_i < x < x_{i+1} \\ 0 & \text{for } x > x_{i+1} \end{cases}$$

But, if we only have "nodal points" or grid points, or given points on $\mathbb{R}$ to evaluate $\widetilde{T}$ of $\{x_0, x_1, \dots x_N\}$ (for $N+1$ points), then how can we define the following basis functions?

$$\phi_0(x) = \begin{cases} \frac{x_1-x}{\Delta x_0} & \text{for } x_0 < x < x_1 \\ 0 & \text{for } x > x_1 \end{cases}$$

$$\phi_1(x) = \begin{cases} 0 & \text{for } x < x_0 \\ \frac{x-x_0}{\Delta x_0} & \text{for } x_0 < x < x_1 \\ \frac{x_2-x}{\Delta x_1} & \text{for } x_1 < x < x_2 \\ 0 & \text{for } x > x_2 \end{cases}$$

$$\phi_N(x) = \begin{cases} 0 & \text{for } x < x_{N-1} \\ \frac{x-x_{N-1}}{\Delta x_{N-1}} & \text{for } x_{N-1} < x < x_N \\ \frac{x_{N+1}-x}{\Delta x_N} & \text{for } x_N < x < x_{N+1} \\ 0 & \text{for } x > x_{N+1} \end{cases}$$

$$\phi_{N+1}(x) = \begin{cases} 0 & \text{for } x < x_N \\ \frac{x-x_N}{\Delta x_N} & \text{for } x_N < x < x_{N+1} \end{cases}$$

Consider defining the residual $R = R(\widetilde{T},x)$ as

$$R(\widetilde{T},x) := \text{div}(k\text{grad}T) + q$$

and consider multiplying it by so-called "weights", Φ, which is set to be ϕ_j's.

$$\begin{gathered} \int_\Omega d*(kdT)\Phi = \int_\Omega -q\Phi\text{vol}^d \Longrightarrow \int_\Omega (d*(kdT) + q\text{vol}^d)\Phi = 0 \\ \Longrightarrow \int_{\partial\Omega} *(kdT)\Phi 0 \int_\Omega \langle \text{grad}\Phi, k\text{grad}T\rangle + \int_\Omega q\Phi\text{vol}^d \end{gathered}$$

If $\Phi = \phi_j(x)$, s.t. $\phi_j(x) = 0$ on $\partial\Omega$,

$$-\int_\Omega \langle \text{grad}\phi_j, k\text{grad}T\rangle + \int_{\mathcal{O}_j} q\phi_j\text{vol}^d = 0$$

e.g. $\mathcal{O}_j = [x_{j-1}, x_{j+1}]$

if

$$\phi_j(x) = \begin{cases} 0 & \text{for } x < x_{i-1} \\ \frac{x - x_{i-1}}{\Delta x_{i-1}} & \text{for } x_{i-1} < x < x_i \\ \frac{x_{i+1} - x}{\Delta x_i} & \text{for } x_i < x < x_{i+1} \\ 0 & \text{for } x > x_{i+1} \end{cases}$$

$$\int_{x_{i-1}}^{x_i} \frac{k\text{grad}T}{\Delta x_{i-1}} \text{vol}^d - \int_{x_i}^{x_{i+1}} \frac{k\text{grad}T}{\Delta x_i} \text{vol}^d + \int_{\mathcal{O}_j} q\phi_j \text{vol}^d = 0$$

Suppose $\widetilde{T}(x) = \sum_{i=1}^{N+1} a_i \phi_k(x)$,

$$\int_{x_{i-1}}^{x_i} \frac{k}{\Delta x_{i-1}^2}(a_i - a_{i-1}) - \int_{x_i}^{x_{i+1}} \frac{k}{\Delta x_i^2}(-a_i + a_{i+1}) + \int_{\mathcal{O}_j} q\phi_j \text{vol}^d = 0 =$$

$$= \frac{a_i - a_{i-1}}{\Delta x_{i-1}^2} \int_{x_{i-1}}^{x_i} k - \frac{a_{i+1} - a_i}{\Delta x_i^2} \int_{x_i}^{x_{i+1}} k + \int_{\mathcal{O}_j} q\phi_j \text{vol}^d = 0 = k\frac{a_i - a_{i-1}}{\Delta x_{i-1}} - k\frac{a_{i+1} - a_i}{\Delta x_i} + \int_{\mathcal{O}_j} q\phi_j \text{vol}^d = 0$$

If $q = 50e^x$, and consider that

$$\int 50 e^x \phi_j \text{vol}^d = \frac{50}{\Delta x_{i-1}}(xe^x - e^x - x_{i-1}e^x)$$

Part 8. CUB, NCCL

21. CUB

21.1. Striped arrangement.

Recall our notation

$$i_x \in \{0, 1, \dots M_x - 1\} \subset \mathbb{Z} \Longleftrightarrow \texttt{threadIdx.x}$$

with M_x corresponding to `blockDim.x`

Consider

$$\begin{gathered} F : \{0, 1, \dots M_x - 1\} \subset \mathbb{Z} \to \mathbb{K}^d \\ F(i_x) = (F^{(0)}(i_x), F^{(d)}(i_x) \dots F^{(d)}(i_x)) \end{gathered}$$

with $d \Longleftrightarrow$ items per thread, or `ITEMS_PER_THREAD`.

5.3 Flexible data arrangement across threads, Striped arrangement mentions a *logical stride* $S_x \in \mathbb{Z}^+ \Longleftrightarrow$ `BLOCK_THREADS`.

It goes on to say (CUB documentation) "thread i owns items $(i), (i + \texttt{block-threads}), \dots (i + \texttt{block-threads} * (\texttt{items-per-thread} - 1))$.

Denote this `block-threads` $*$ `items-per-thread`, stripped arrangement, as $S_x d$.

What is this `BLOCK_THREADS`? The thread block size in threads, from template parameters for the cub LoadDirectStriped documentation. It wasn't obvious to me at first, until reading the documentation on its template parameters.

So

$$S_x = M_x$$

Is this enforced somehow? What happens if $S_x > 1024$, the hardware limit on number of threads on a single block?

So we are asked to consider

$$i_x, M_x, d \mapsto \{i_x, i_x + M_x, \dots i_x + M_x l^x, \dots i_x + M_x(d-1)\}_{l^x = 0, 1, \dots d-1}$$

Consider the "maximal" case:

$$i_x = M_x - 1, M_x, d \mapsto \{M_x - 1, 2M_x - 1, \dots M_x d - 1\}$$

So we have $M_x d$ total values to consider for this single (thread) block.

In reality, what's really going on is the flatten functor, necessitated by just how C/C++ doesn't have multi-dimensional arrays.

$$\begin{gathered} F : \{0, 1, \dots M_x - 1\} \subset \mathbb{Z} \to \mathbb{K}^d \mapsto F : \{0, 1 \dots M_x d - 1\} \subset \mathbb{Z} \to \mathbb{K}^1 \\ F \in L(\{0, 1 \dots M_x - 1\} \subset \mathbb{K}^d) \to L(\{0, 1 \dots M_x d - 1\}, \mathbb{K}) \end{gathered} \tag{119}$$

with $M_x \leq 1024$ (hardware enforced)

22. NCCL - Optimized primitives for collective multi-GPU communication

23. Theano's scan

cf. theano scan tutorial

23.1. Example 3: Reusing outputs from the previous iterations.

For input $\begin{aligned} X : \mathbb{Z}^+ &\to R - \text{Module} \\ t &\mapsto X(t) \end{aligned}$, e.g. R-Module such as $\mathbb{R}^d, V, \text{Mat}_{\mathbb{R}}(N_1, N_2), \tau_s^r(V)$, and a function f that acts at each iteration, i.e. $\forall\, t = 0, 1, \dots T$,

$$\begin{aligned} f : R - \text{Module} \times R - \text{Module} &\to R - \text{Module} \\ (X_1, X_0) &\mapsto X_1 + X_0 \end{aligned}$$

, then we want to express

$$f(X(t), X(t-1)) = X(t) + X(t-1) \qquad \forall\, t = 0, 1 \dots T - 1,$$

In the end, we should get

$$X \in (R - \text{Module})^T \xrightarrow{\text{scan}} Y \in (R - \text{Module})^T$$

$Y \equiv$ `output`

In summary, the dictionary between the mathematics and the Python theano code for scan seems to be the following:

If $k = 1$, $\forall\, t = 0, 1, \dots T - 1$,

(120)

$$\begin{aligned} F : R - \text{Module} \times R - \text{Module} &\to R - \text{Module} \\ F(X(t), X(t-1)) &\mapsto X(t) \end{aligned} \Longleftrightarrow \text{Python function (object) or Python lambda expression } \mapsto \texttt{scan(fn=}\quad\texttt{)}$$

$$(X(0), X(1), \dots X(T-1)) \in (R - \text{Module})^T \Longleftrightarrow \texttt{scan(sequences=}\quad\texttt{)}$$

$$X(-1) \in R - \text{Module} \Longleftrightarrow \texttt{scan(outputs\_info=[}\quad\texttt{])}$$

23.2. Example 4 : Reusing outputs from multiple past iterations.

$\forall\, t = 0, 1, \dots T - 1$, $T \Longleftrightarrow$ `n_steps`$= T$,

Consider

$$\begin{aligned} F : (R - \text{Module})^k &\to R - \text{Module} \\ F(X(t-k), X(t-(k-1)), \dots X(t-1)) &= X(t) \end{aligned} \Longleftrightarrow \texttt{fn} \in \text{Python function (object)} \tag{121}$$

If $k = 1$, we'll need to be given $X(0) \in R -$ Module. Perhaps consider $\forall\, t = -k, -(k-1), \dots -1, 0, 1 \dots T - 1$ ("in full").

For $k > 1$, we'll need to be given (or declare) $\{X(-k), X(-(k-1)), \dots X(-1)\}$.

So for $k = 1$, $X(-1) \in R -$ Module needed $\Longleftrightarrow$ e.g. `T.scalar()` if R-Module $= \mathbb{R}$.

for $k > 1$, $(X(-k), X(-(k-1)), \dots X(-1)) \in (R - \text{Module})^k \Longleftrightarrow$ e.g. `T.vector()`, into `''initial''` of a Python `dict`, if R-Module $= \mathbb{R}$.

Also, for $k > 1$,

$$(-k, -(k-1), \dots -1) \Longleftrightarrow \texttt{taps} = [-k, -(k-1), \dots -1](\text{a Python } \texttt{list})$$

scan, essentially, does this:

$$\begin{gathered} (X(-k), X(-(k-1)), \dots X(-1)) \mapsto (X(0), X(1) \dots X(T-1)) \\ F(X(t-k), X(t-(k-1)) \dots X(t-1)) = X(t), \qquad \forall\, t = 0, 1, \dots T - 1 \end{gathered} \tag{122}$$

given $F : (R - \text{Module})^k \to R -$ Module, with $T =$`n_steps`.

References

[1] Trevor Hastie, Robert Tibshirani, Jerome Friedman. **The Elements of Statistical Learning: Data Mining, Inference, and Prediction**, Second Edition (Springer Series in Statistics) 2nd ed. 2009. Corr. 7th printing 2013 Edition. ISBN-13: 978-0387848570. https://web.stanford.edu/~hastie/local.ftp/Springer/OLD/ESLII_print4.pdf

[2] Jared Culbertson, Kirk Sturtz. *Bayesian machine learning via category theory.* arXiv:1312.1445 [math.CT]

[3] John Owens. David Luebki. *Intro to Parallel Programming. CS344.* **Udacity**
http://arxiv.org/abs/1312.1445 Also, https://github.com/udacity/cs344

[4] John Cheng, Max Grossman, Ty McKercher. **Professional CUDA C Programming**. 1st Edition. Wrox; 1 edition (September 9, 2014). ISBN-13: 978-1118739327

[5] CS229 Stanford University. http://cs229.stanford.edu/materials.html

[6] Richard Fitzpatrick. "Computational Physics." http://farside.ph.utexas.edu/teaching/329/329.pdf

[7] M. Hjorth-Jensen, **Computational Physics**, University of Oslo (2015) http://www.mn.uio.no/fysikk/english/people/aca/mhjensen/

[8] Prof. Dr. Michael Bader (*Lecturer*); Alexander Pöppl, Valeriy Khakhutskyy. (*Tutorials*). HPC - Algorithms and Applications - Winter 16. Winter 16/17. **TUM**. https://www5.in.tum.de/wiki/index.php/HPC_-_Algorithms_and_Applications_-_Winter_16

[9] Bjarne Stroustrup. **A Tour of C++** (C++ In-Depth Series). Addison-Wesley Professional. 2013.

[10] Jason Sanders, Edward Kandrot. **CUDA by Example: An Introduction to General-Purpose GPU Programming** 1st Edition. Addison-Wesley Professional; 1 edition (July 29, 2010). ISBN-13: 978-0131387683

[11] David Darmofal. "16.901 Computational Methods in Aerospace Engineering, Spring 2005." (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu (Accessed 12 Jun, 2016). License: Creative Commons BY-NC-SA

[12] Joel H. Ferziger and Milovan Peric. **Computational Methods for Fluid Dynamics**. Springer; 3rd edition (October 4, 2013). ISBN-13: 978-3540420743
I used the 2002 edition since that was the only copy I had available.

[13] Michael Griebel, Thomas Dornsheifer, Tilman Neunhoeffer. **Numerical Simulation in Fluid Dynamics: A Practical Introduction (Monographs on Mathematical Modeling and Computation)**. SIAM: Society for Industrial and Applied Mathematics (December 1997). ISBN-13: 978-0898713985 QA911.G718 1997
See also Software of Research group of Prof. Dr. M. Griebel, Institute für Numerische Simulation http://wissrech.ins.uni-bonn.de/research/software/

[14] Kyle E. Niemeyer, Chih-Jen Sung. *Accelerating reactive-flow simulations using graphics processing units.* 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition 07-10 January 2013, Grapevine, Texas. American Institute of Aeronautics and Astronautics. AIAA 2013-0371

[15] Carlos Henrique Marchi, Roberta Suero, Luciano Kiyoshi Araki. *The Lid-Driven Square Cavity Flow: Numerical Solution with a 1024 x 1024 Grid.* **J. of the Braz. Soc. of Mich. Sci. & Eng. 186**, Vol. XXXI, No. 3, July-September 2009. ABCM

[16] Rubin H. Landau, Manuel J Páez, Cristian C. Bordeianu. **Computational Physics: Problem Solving with Python**. 3rd Edition. Wiley-VCH; 3 edition (September 8, 2015). ISBN-10: 3527413154 ISBN-13: 978-3527413157

[17] Vladimir I. Arnold, Valery V. Kozlov, Anatoly I. Neishtadt, E. Khukhro. **Mathematical Aspects of Classical and Celestial Mechanics (Encyclopaedia of Mathematical Sciences)** 3rd Edition. Encyclopaedia of Mathematical Sciences (Book 3). Springer; 3rd edition (November 14, 2006). ISBN-13: 978-3540282464