Parallelization of vector operations

For iterative methods it is important to parallelize vector operations:

- Scalar product
- Basic operation
- ‘axpy’ $x = ax + y$
- Sparse matrix $\times$ vector

- Few operations per memory access, relatively fine grained parallelism
- $\rightarrow$ benchmark this

- STREAM benchmark

- “Schönauer vector triad”: $d = a + b \times c$
  4 vectors, 2 Flops per index.
Small problems: scalar is fastest due to scheduling overhead
Medium problems: parallel is fast
Large problems: no big difference due to memory bandwidth: all laptop cores access the memory through the same bottleneck
“sweet spot” for parallel between $10^4$ and $5 \cdot 10^5 \ldots 5 \cdot 10^6$
**Memory performance: vector triad II**

- Benchmarking site of G. Hager
  - https://blogs.fau.de/hager/archives/tag/benchmarking
- lstopo for this laptop

**Throughput A(;) = B(;) + C(;) * D(;)**

- Intel Skylake Xeon E3-1275 v5 @ 3.0 GHz, Intel 17.0 up2

- Performance drops are correlated with cache sizes
- Most important: large L3 cache
Memory performance: vector triad III

**gcc vs. icc (2013)**

https://blogs.fau.de/hager/archives/tag/benchmarking

- Picture similar to early times of gnu compiler vs Intel
- Julia barrier implementation seems to need improvement
- “hand crafted” threading works better
Memory performance: vector triad IV

Julia vs gcc on Server

Julia vs GCC Thread Performance: \( A[i] = B[i] + C[i] \times D[i] \)
2 socket(s) 32 cores Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
RAM=503GiB Cache: L3=40MiB L2=256kB L1=32kB

Julia vs gcc on Laptop

Julia vs GCC Thread Performance: \( A[i] = B[i] + C[i] \times D[i] \)
1 socket(s) 6 cores Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz
RAM=31GiB Cache: L3=12MiB L2=256kB L1=32kB

- Laptop L3 cache too small for sustaining performance for large arrays
- Server has larger L3 cache and 2 sockets \( \equiv \) 2 lanes to memory, \( \Rightarrow \) still see parallel speed-up for the largest case
- Julias ‘Threads.@threads‘ performs quite well once the chunk size is large enough compared to the barrier implementation overhead
Laptop L3 cache too small for sustaining performance for large arrays

Server has larger L3 cache and 2 sockets $\equiv$ 2 lanes to memory, $\Rightarrow$ still see parallel speed-up for the largest case

Julia’s `Threads.@threads` performs quite well once the chunk size is large enough compared to the barrier implementation overhead
Principle useful for highly structured data
Example: textures, triangles for 3D graphics rendering
During the 90’s, *Graphics Processing Units* (GPUs) started to contain special purpose SIMD hardware for graphics rendering
3D Graphic APIs (DirectX, OpenGL) became transparent to programmers: rendering could be influenced by “shaders” which essentially are programs which are compiled on the host and run on the GPU
General Purpose Graphics Processing Units (GPGPU)

- Graphics companies like NVIDIA saw an opportunity to market GPUs for computational purposes.
- Emerging APIs which allow to describe general purpose computing tasks for GPUs: CUDA (Nvidia specific), OpenCL (ATI/AMD designed, general purpose), OpenACC based on compiler directives.
- GPGPUs are accelerator cards added to a computer with own memory, many vector processing pipelines and special bus interconnect (NVidia Quadro GV100: 32GB +5120 units, NVLink; Tensor cores).
- CPU-GPU connection via mainboard bus / special link.
CPU:
- Sets up data
- Triggers compilation of "kernels": the heavy duty loops to be executed on GPU
- Sends compiled kernels ("shaders") to GPU
- Sends data to GPU, initializes computation
- Receives data back from GPU

GPU:
- Receive data from host CPU
- Run the heavy duty loops in local memory
- Send data back to host CPU

For high performance one needs explicit management of these steps

Bottleneck: Data transfer CPU ↔ GPU

High efficiency only with good match between data structure and layout of GPU memory (2D rectangular grid)
NVIDIA Cuda

- Established by NVIDIA GPU vendor
- Works only on NVIDIA cards
- Claimed to provide optimal performance
CUDA Data organization

- Threads can be arranged in 1, 2, or 3 dimensional Blocks and can execute a kernel within given 1/2/3D index range
- Blocks are arranged in a 2D Grid

https://commons.wikimedia.org/wiki/File:Block-thread.svg
CUDA Kernel code

- The kernel code is the code to be executed on the GPU aka “Device”
- It needs to be compiled using special CUDA compiler

```c
#include <cuda_runtime.h>

/*
 * CUDA Kernel Device code
 * Computes the vector addition of A and B into C.
 * The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```
int main(void)
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate host vectors
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // Allocate device vectors
    float *d_A = NULL;
    float *d_B = NULL;
    float *d_C = NULL;
    assert(cudaMalloc((void **)&d_A, size)==cudaSuccess);
    assert(cudaMalloc((void **)&d_B, size)==cudaSuccess);
    assert(cudaMalloc((void **)&d_C, size)==cudaSuccess);
    ...
}
... 

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

assert(cudaGetLastError()==cudaSuccess);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
free(h_B);
free(h_C);
cudaDeviceReset();
OpenCL

- “Open Computing Language”
- Vendor independent
- More cumbersome to code
Example: OpenCL: computational kernel

```c
__kernel void square(
   __global float* input, __global float* output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

Declare functions with `__kernel` attribute
Defines an entry point or exported method in a program object

Use address space and usage qualifiers for memory
Address spaces and data usage must be specified for all memory objects

Built-in methods provide access to index within compute domain
Use `get_global_id` for unique work-item id, `get_group_id` for work-group, etc

```c
// Fill our data set with random float values
int count = 1024 * 1024;
for (i = 0; i < count; i++)
    data[i] = rand() / (float)RAND_MAX;

// Connect to a compute device, create a context and a command queue
cl_device_id device;
clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(0, 1, &device, NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

// Create and build a program from our OpenCL-C source code
cl_program program = clCreateProgramWithSource(context, 1, (const char **) &src, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create a kernel from our program
cl_kernel kernel = clCreateKernel(program, "square", NULL);
```

// Allocate input and output buffers, and fill the input with data
cl_mem input = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, NULL);

// Create an output memory buffer for our results
cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);

// Copy our host buffer of random values to the input device buffer
clEnqueueWriteBuffer(queue, input, CL_TRUE, 0, sizeof(float) * count, data, 0, NULL, NULL);

// Get the maximum number of work items supported for this kernel on this device
size_t global = count; size_t local = 0;
clGetKernelWorkGroupInfo(kernel, device, CL_KERNEL_WORK_GROUP_SIZE, sizeof(int), &local, NULL);

// Set the arguments to our kernel, and enqueue it for execution
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);

// Force the command queue to get processed, wait until all commands are complete
clFinish(queue);

// Read back the results
clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(float) * count, results, 0, NULL, NULL);

// Validate our results
int correct = 0;
for(i = 0; i < count; i++)
    correct += (results[i] == data[i] * data[i]) ? 1 : 0;

// Print a brief summary detailing the results
printf("Computed '%d/%d' correct values!\n", correct, count);

OpenCL Summary

- Need good programming experience and system management skills in order to set up tool chains with properly matching versions, vendor libraries etc.
  - (I was not able to get this running on my laptop in finite time...)
- Very cumbersome programming, at least as explicit as MPI
- Data structure restrictions limit class of tasks which can run efficiently on GPUs.
Compiler directive based GPU programming

- OpenMP
  - OpenMP 4.0
  - Implementation in commercial compilers
  - GCC, Clang implementations under development

- OpenACC
  - Idea similar to OpenMP: use compiler directives
  - Future merge with OpenMP initially intended, now they seem to be competitors
  - Intended for different accelerator types (Nvidia GPU ...)
  - Commercial compiler vendors, e.g. PGI (with free academic license valid one year)
  - GCC, Clang implementations under development
OpenACC code

“Shader”:

```c
void vecaddgpu( float *restrict r, float *a, float *b, int n, int nrepeat)
{
    int irepeat;
    #pragma acc kernels loop present(r,a,b)
    for (irepeat=0; irepeat<nrepeat; irepeat++)
        for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i] + irepeat;
}
```

Invocation from CPU

```c
a = (float*)malloc( n*sizeof(float) );
b = (float*)malloc( n*sizeof(float) );
r = (float*)malloc( n*sizeof(float) );
e = (float*)malloc( n*sizeof(float) );
#pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
{
    vecaddgpu( r, a, b, n, nrepeat );
}
```

Compile with PGI compiler (https://www.pgroup.com/)

```
pgcc -ta=tesla -fast -o add2 add2.c
```
Other ways to program GPU

- Directly use graphics library
- Modern OpenGL with shaders
- Julia: CuArrays directly compile to kernel instructions
Gray-Scott model for Reaction-Diffusion: two species.

- $U$ is created with rate $f$ and decays with rate $f$
- $U$ reacts with $V$ to more $V$
- $V$ decays with rate $f + k$.
- $U, V$ move by diffusion

\[
\begin{align*}
1 & \xrightarrow{f} U \\
U + 2V & \xrightarrow{1} 3V \\
V & \xrightarrow{f + k} 0 \\
F & \xrightarrow{f} 0
\end{align*}
\]

Stable states:
- No $V$
- "Much of $V$, then it feeds on $U$ and re-creates itself"

Reaction-Diffusion equation from mass action law:

\[
\frac{\partial}{\partial t} u - D_u \Delta u + uv^2 - f(1 - u) = 0
\]
... GPUs are fast so we choose the explicit Euler method:

\[
\frac{1}{\tau}(u_{n+1} - u_n) - D_u \Delta u_n + u_n v_n^2 - f(1 - u_n) = 0
\]

\[
\frac{1}{\tau}(v_{n+1} - v_n) - D_v \Delta v_n - u_n v_n^2 + (f + k)v_n = 0
\]

- Finite difference/finite volume discretization on grid of size \( h \)

\[
-\Delta u \approx \frac{1}{h^2} (4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1})
\]
The shader

```glsl
precision mediump float;
uniform sampler2D u_image;
uniform vec2 u_size;
const float F = 0.05, K = 0.062, D_a = 0.2, D_b = 0.1;
const float TIMESTEP = 1.0;
void main() {
    vec2 p = gl_FragCoord.xy,
        n = p + vec2(0.0, 1.0),
        e = p + vec2(1.0, 0.0),
        s = p + vec2(0.0, -1.0),
        w = p + vec2(-1.0, 0.0);

    vec2 val = texture2D(u_image, p / u_size).xy,
        laplacian = texture2D(u_image, n / u_size).xy
           + texture2D(u_image, e / u_size).xy
           + texture2D(u_image, s / u_size).xy
           + texture2D(u_image, w / u_size).xy
           - 4.0 * val;

    vec2 delta = vec2(D_a * laplacian.x - val.x*val.y*val.y + F * (1.0-val.x),
                      D_b * laplacian.y + val.x*val.y*val.y - (K+F) * val.y);

    gl_FragColor = vec4(val + delta * TIMESTEP, 0, 0);
}
```
Why does this work so well here?

- Data structure fits very well to topology of GPU
  - rectangular grid
  - 2 unknowns to be stored in x,y components of vec2
- No communication with CPU in the first place
- GPU speed allows to “break” time step limitation of explicit Euler
- Data stay within the graphics card: once we loaded the initial value, all computations, and rendering use data which are in the memory of the graphics card.
- Depending on the application, choose the best way to proceed
- e.g. deep learning (especially training speed)