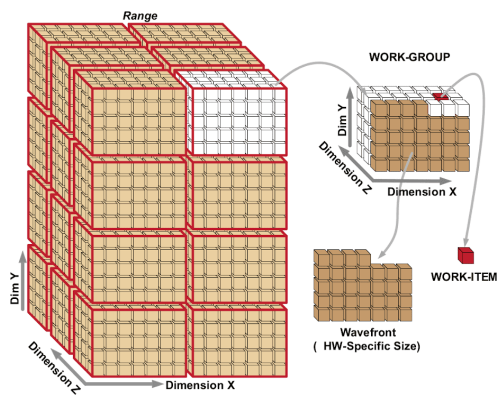
OpenCL for High-Performance Computing

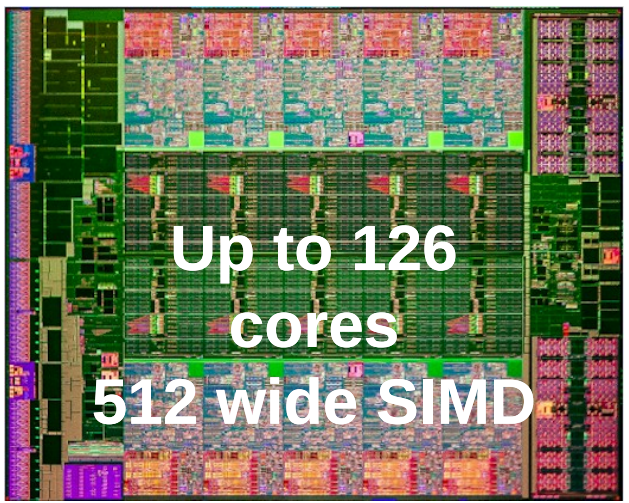
Created by Shromm Gaind



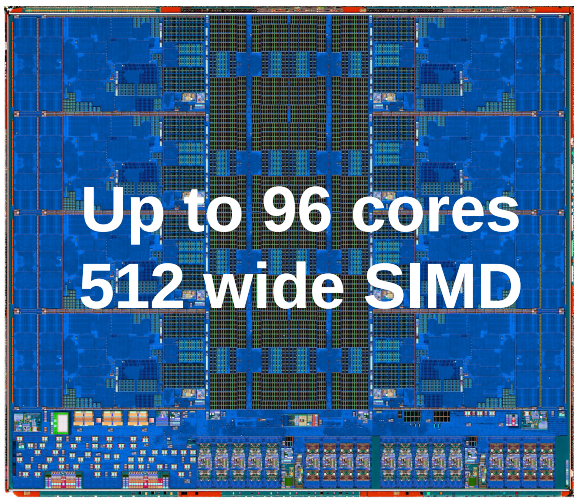
We Can't Just Wait for Faster CPUs

For decades, our code got faster for free. Those days are over. The industry's answer isn't faster cores, it's more cores.

- **For performance, concurrency is no longer optional. It's mandatory.**



Intel® Granite Rapids microarchitecture

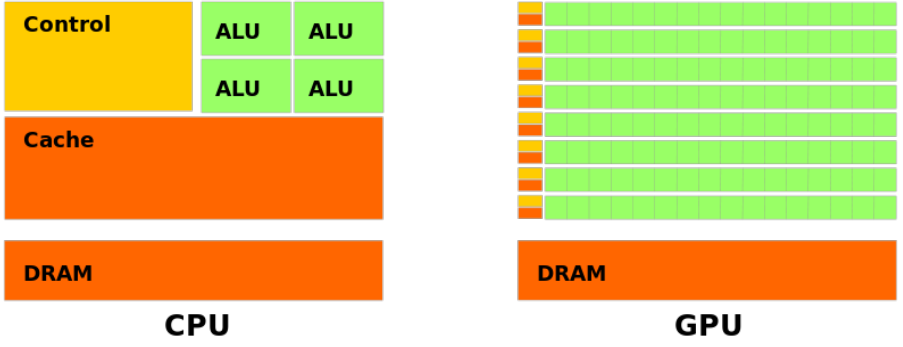


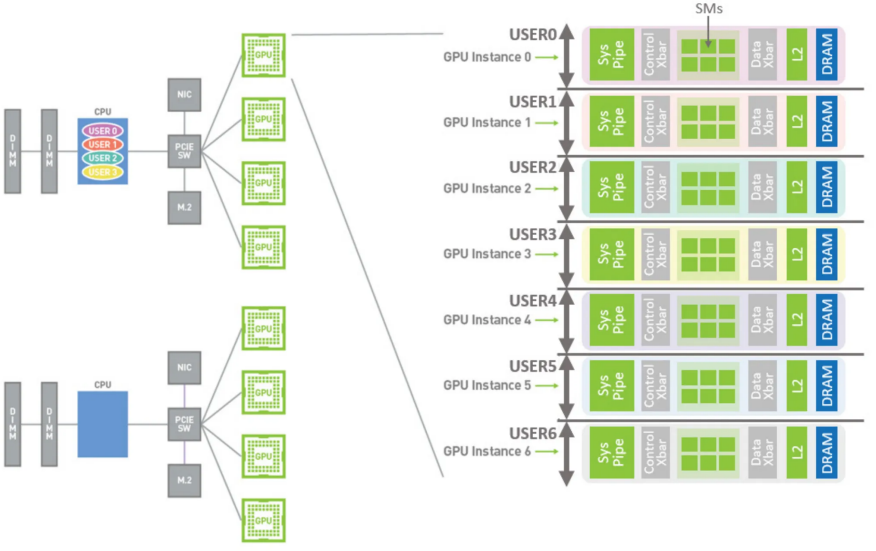
AMD® Zen 5 microarchitecture

Heterogeneous Computing

HPC is about mapping latency-sensitive control flow to the CPU and data-parallel, high-throughput work to the GPU.

- **Modern HPC systems are built on this principle.**

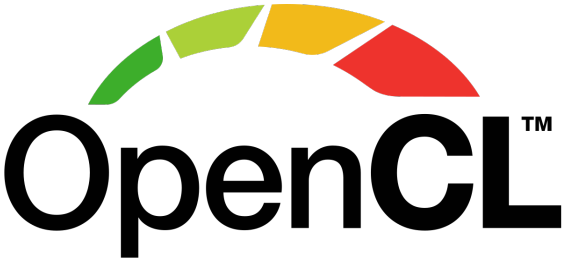




Why OpenCL for HPC

Powerful, Mature Ecosystem

Proprietary: NVIDIA hardware only



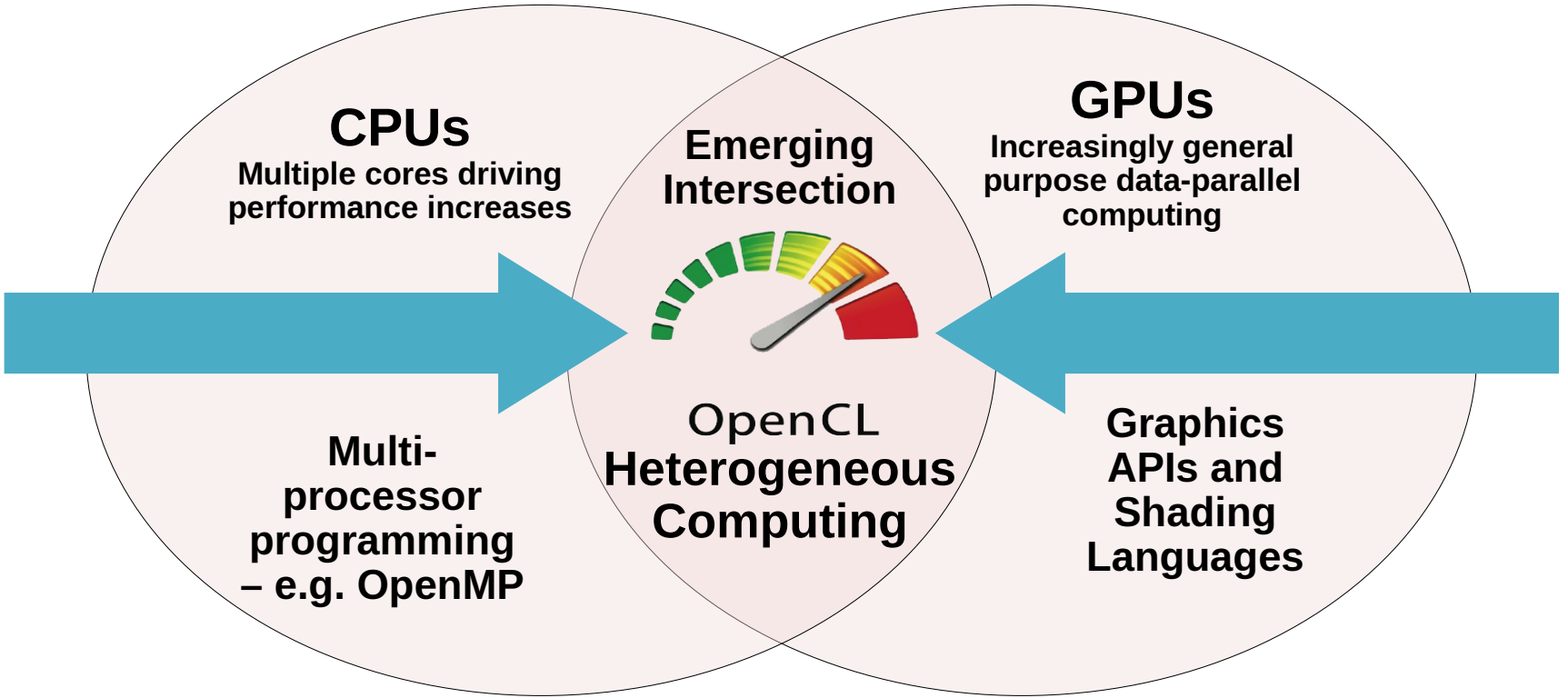
Khronos Open standard for heterogeneous processing

For host-side control (C/C++).

OpenCL C (based on C99 with extensions).

Mordern Implementations such as SYCL, DPC++

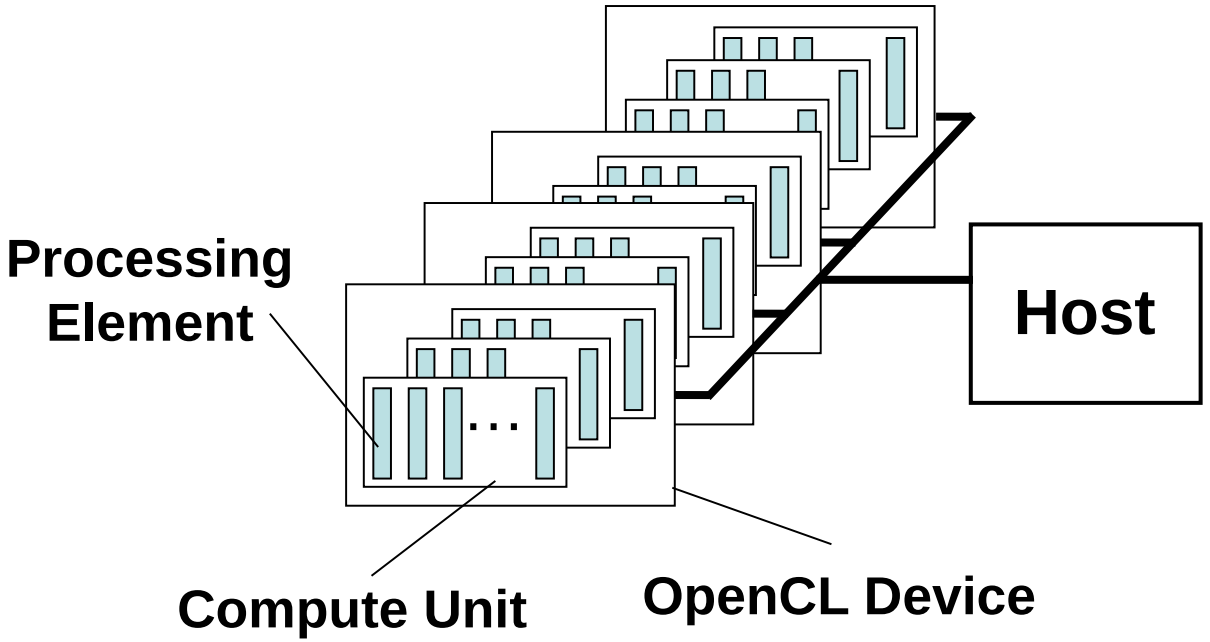
Industry Standards for Programming Heterogeneous Platforms



OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL Platform Model



Memory divided into ***host memory (CPU)*** and ***device memory (GPU, FGPA …)***

Kernels in GPU Programming

- Replace loops with functions (a kernel) executing at each point in a problem domain

E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

Serial loop

```
for (int y = 0; y < H; ++y) {
  for (int x = 0; x < W; ++x)
{   process_pixel(x, y);
  }
}
```

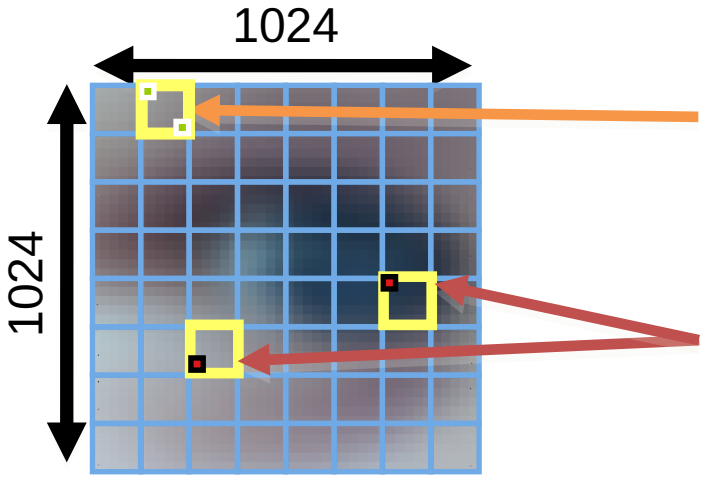
Data Parallel OpenCL

```
// Kernel: runs once per (x,y) work-item
__kernel void process_pixel(__global uchar4*
image,                      const uint    width)
{
    // each work-item computes its own coords
    uint x   = get_global_id(0);
    uint y   = get_global_id(1);
    uint idx = y * width + x;

    uchar4 px = image[idx];
    // …do your per-pixel work…
    image[idx] = px;
}
```

An N-dimensional domain of work-items

- Global Dimensions :
  - e.g 1024x1024 (whole problem space)
  - **Choose the dimensions that are specfic to your algorithim**
- Local Dimensions:
  - e.g 64x64 (**work-group**, executes together)



Synchronization between work-items possible only within work-groups:
barriers and memory fences

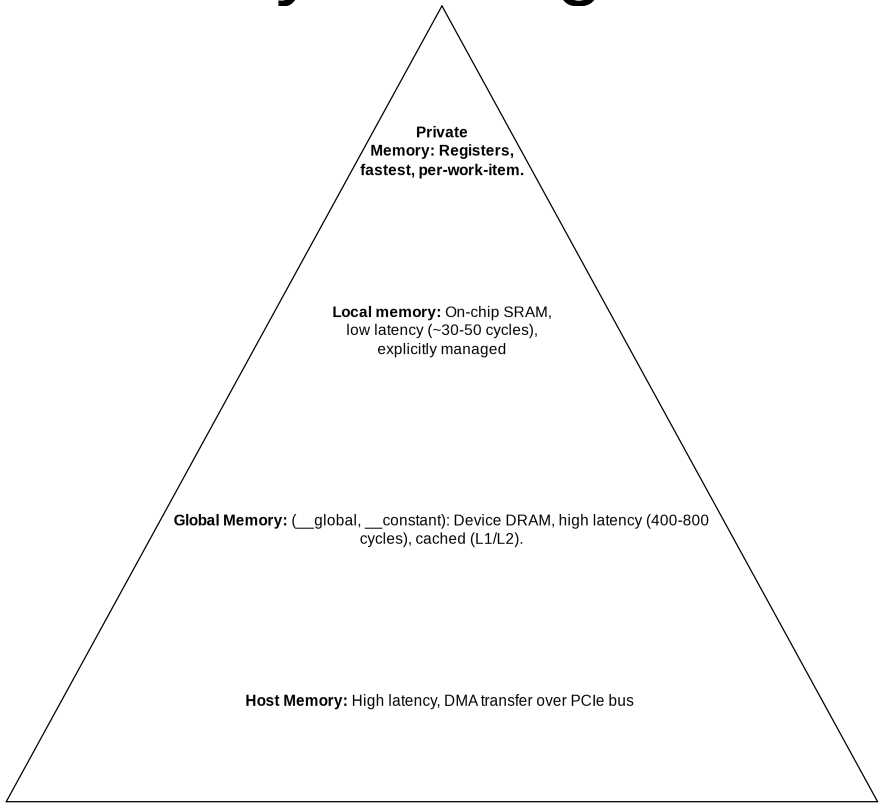
Cannot synchronize between work-groups within a kernel

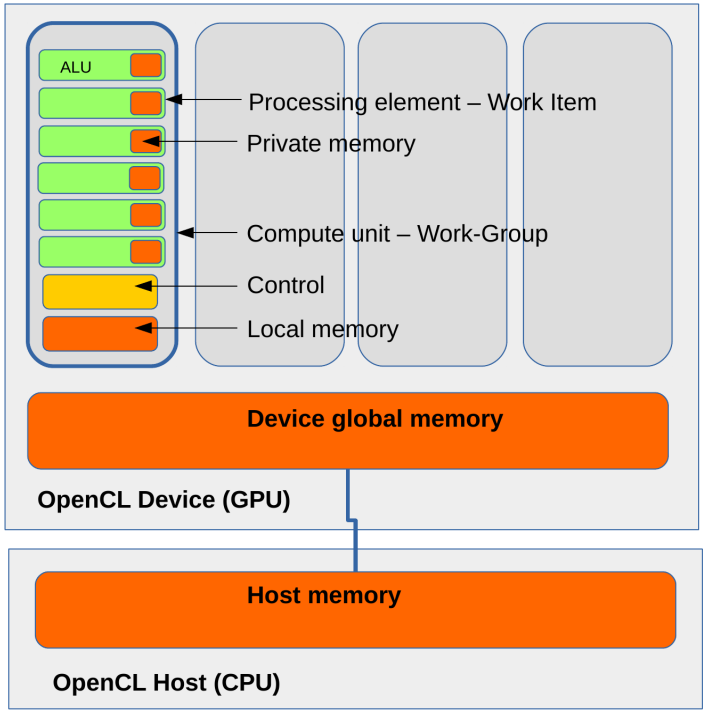
Organising the Work: The N-Dimensional Range

- **The NDRange is an N-dimensional index space (where N is 1, 2, or 3) that defines the total domain of computation. Each unique point in this space is a Work-Item.**
  - **This global index space is partitioned into uniform, contiguous blocks called Work-Groups.**
    - **Global Work Size (global_work_size)**
    - **Local Work Size (local_work_size)**
- **All work-items within a single work-group are guaranteed to execute concurrently on a single Compute Unit (CU).**
  - **This concurrency enables**
    - **Fast Data Sharing via on-chip __local memory.**
    - **Low-latency Synchronization using the barrier() primitive.**
- You specify **local_work_size** when enqueuing a kernel. Passing **NULL** lets the runtime decide, but this is **almost never optimal**

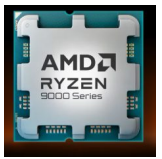
OpenCL Memory Model

- Memory management is explict





The basic platform and runtime APIs in OpenCL (using C)



Context

Programs

```
__kernel void
dp_mul(global const float
*a,    global const float
*b,    global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

dp_mul
CPU program binary

dp_mul
GPU program binary

Kernels

dp_mul
arg[0] value
arg[1] value
arg[2] value

Memory Objects

Buffers

Command Queues

In Order Queue

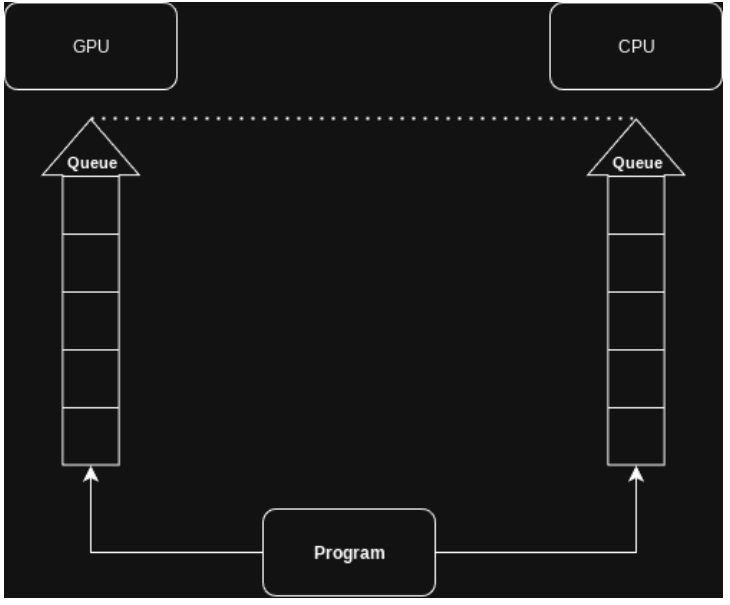
Out of Order Queue

Compute Device

Compile code → Create data & arguments → Send to execution

Command-Queues

- A **command queue** is a host-side object that accepts commands and submits them in order to a specific device.
- **Asynchronous by Default**: Command submission is non-blocking. The host can queue up a sequence of operations and continue with its own work.
- **Execution Models**
  - *In-Order Queue*: They excute in order they are enqueued
  - *Out-of-Order Queue*: The device driver is free to reorder commands



Buffers and Memory objects

- **Abstracting Device Memory**: A `cl_mem` object is a handle to a reference-counted region of memory on the device (`__global memory`).
- Primary Type: **Buffer Objects**
  - Represents a simple, linear block of **bytes**—essentially a "C array on the device."
  - Naming Convention: A common and highly recommended practice is to prefix host arrays with h_ and device buffers with d_.

```
// On the Host (CPU RAM)
float* h_input = (float*)malloc(bytes);

// On the Device (GPU VRAM) - a handle
cl_mem d_input = clCreateBuffer(context, flags, bytes, NULL,
&err);
```

Creating Buffers: Defining Access Intent

- The `flags` argument in `clCreateBuffer` is critical. It tells the driver how the memory will be used, allowing for significant optimization.
- **Kernel Access:**
  - `CL_MEM_READ_ONLY`: Kernel will only read from this buffer.
  - `CL_MEM_WRITE_ONLY`: Kernel will only write to this buffer.
  - `CL_MEM_READ_WRITE`: Kernel can read and write.
- **Host Initialization:**
  - `CL_MEM_COPY_HOST_PTR`: Allocate device memory and copy data from the provided host pointer during creation.

Data Transfer

- After creation, you move data using explicit commands:

```
// Copy FROM host TO device (blocking call)
clEnqueueWriteBuffer(queue, d_input, CL_TRUE, 0, bytes, h_input, 0, NULL, NULL);

// Copy FROM device TO host (blocking call)
clEnqueueReadBuffer(queue, d_output, CL_TRUE, 0, bytes, h_output, 0, NULL,
NULL);
```

Kernel Programming

- **C syntax (C99)**
- __**kernel**: The Entry Point
- __global / __constant: Pointers passed as arguments must have an address space qualifier. This explicitly defines them as living in device memory.
- Work-Item
  Functions: get_global_id(dim), get_local_id(dim), get_group_id(dim), get_global_size(dim).
- These are the intrinsic functions that connect your code to its position in the N-dimensional problem grid, enabling data-parallel computati
- Core Restriction: **No Recursion**
  - GPU hardware lacks the stack management for deep recursion
  - Algorithms must be re-formulated iteratively, often using a "work queue" pattern.
- **No Function Pointers** (as kernel arguments)

Explicitly Managing Data Locality

- **Address Spaces: YOU EXPLICITLY CONTROL DATA PLACEMENT**
  - **`__global`**: Main device memory (DRAM). High-latency, high-bandwidth.
  - **`__local`**: On-chip SRAM. Low-latency, shared within a work-group.
  - **`__constant`**: Read-only region of global memory, often with a dedicated cache.
  - **`__private`**: Per-work-item registers. Fastest access.

Exploiting Hardware SIMD/SIMT

- Vector Data Types in OpenCL
  - Types like `float4`, `char16` map directly to wide hardware execution units.
  - **Performance Rule**: Always prefer vector operations (`float4 a = b * c;`) over scalar loops (`for(i=0; i<4) a[i]=b[i]*c[i];`).
  - Use `vloadN` and `vstoreN` for efficient, aligned reads/writes of vector types from global memory.
  - convert_type(): Efficient, hardware-accelerated type conversions (e.g., `convert_float4(my_int4)`).

LU Decompisition Example

- Transform a matrix into the product of a lower trianglar and upper triangular matrix. Used to solve a linear system of equations

1. The Problem: A System of Linear Equations

We start with a system of equations:

$$x_1 + 2x_2 + 3x_3 = 6$$
$$2x_1 + 5x_2 + 8x_3 = 15$$
$$3x_1 + 8x_2 + 14x_3 = 25$$

Matrix form:

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \\ 3 & 8 & 14 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 6 \\ 15 \\ 25 \end{bmatrix}}_{\mathbf{b}}$$

2. Goal: Decompose $\mathbf{A}$ into $\mathbf{L}$ and $\mathbf{U}$

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Elimination steps to find U:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \\ 3 & 8 & 14 \end{bmatrix} \xrightarrow{R_2 \to R_2 - \mathbf{2}R_1} \xrightarrow{R_3 \to R_3 - \mathbf{3}R_1}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 2 & 5 \end{bmatrix} \xrightarrow{R_3 \to R_3 - \mathbf{2}R_2} \underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{U}}$$

Multipliers become L:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \mathbf{2} & 1 & 0 \\ \mathbf{3} & \mathbf{2} & 1 \end{bmatrix}$$

Solving the System

We replace $\mathbf{A}$ with $\mathbf{LU}$:

$$\mathbf{L}(\mathbf{U}\mathbf{x}) = \mathbf{b}, \quad \mathbf{y} = \mathbf{U}\mathbf{x}$$

Step 3a: Solve $\mathbf{L}\mathbf{y} = \mathbf{b}$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 15 \\ 25 \end{bmatrix}$$

$$y_1 = 6, \quad y_2 = 3, \quad y_3 = 1$$

Step 3b: Solve $\mathbf{U}\mathbf{x} = \mathbf{y}$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \\ 1 \end{bmatrix}$$

$$x_3 = 1, \quad x_2 = 1, \quad x_1 = 1$$

Solution: $\mathbf{x} = [1, 1, 1]^T$

Sequential Code in C

- The C code focues exclusively on the **A → LU**, decomposition part
  - LU Decomposition: **$O(N^3)$**
  - Subsititution: **$O(N^2)$**
  - **$O(N^3) > O(N^2)$**
    - For N = 1024:
      - N^2 = 1,048,576
      - **N^3 = 1,073,741,824**

Any time spent in the $O(N^2)$ solvers is just "noise" in the measurement for large matrices.

```c
int lu_decompose(float A[N][N]) {
    for (int k = 0; k < N; k++) {
        for (int j = k; j < N; j++) {
            float sum = 0.0f;
            for (int p = 0; p < k; p++) {
                sum += A[k][p] * A[p][j];
            }
            A[k][j] = A[k][j] - sum;
        }

        if (fabs(A[k][k]) < 1e-9) { // Use a small epsilon for floating
point
            fprintf(stderr, "Matrix is singular!\n");
            return -1;
        }

        float pivot_inv = 1.0f / A[k][k];
        for (int i = k + 1; i < N; i++) {
            float sum = 0.0f;
            for (int p = 0; p < k; p++) {
                sum += A[i][p] * A[p][k];
            }
            A[i][k] = (A[i][k] - sum) * pivot_inv;
        }
    }
    return 0;
}
```

Benchmark Single Core

Case	**MFLOPS**	
	*CPU	GPU
Sequential C (not OpenCL)	714.8	N/A

MFLOPS = ((2/3) * N^3) / (Time_in_seconds * 10^6)

*Ryzen 9 3900X (12 core 4.2GHz)

Decomposing the Sequential Loops

```
int lu_decompose(float A[N][N]) {
    for (int k = 0; k < N; k++) {
        for (int j = k; j < N; j++) {
            float sum = 0.0f;
            for (int p = 0; p < k; p++) {
                sum += A[k][p] * A[p][j];
            }
            A[k][j] = A[k][j] - sum;
        }

        if (fabs(A[k][k]) < 1e-9) { // Use a small epsilon for floating
point
            fprintf(stderr, "Matrix is singular!\n");
            return -1;
        }

        float pivot_inv = 1.0f / A[k][k];
        for (int i = k + 1; i < N; i++) {
            float sum = 0.0f;
            for (int p = 0; p < k; p++) {
                sum += A[i][p] * A[p][k];
            }
            A[i][k] = (A[i][k] - sum) * pivot_inv;
        }
    }
    return 0;
}
```

Parallelizable Work

Remove outer loops and set work-item co-ordinates

OpenCL Kernel (1/2)

```
__kernel void lu_row_kernel(__global float* A, const int k, const int n)
{   // Each work-item computes one element 'j' in the k-th row of U
    int j = k + get_global_id(0);

    float sum = 0.0f;
    for (int p = 0; p < k; p++) {
        sum += A[k * n + p] * A[p * n + j];
    }
    A[k * n + j] = A[k * n + j] - sum;
}
```

Maps work-item ID to a column index

Each work-item writes one result

OpenCL Kernel (2/2)

```
__kernel void lu_column_kernel(__global float* A, const int k, const int n)
{   // Each work-item computes one element 'i' in the k-th column of L
    int i = k + 1 + get_global_id(0);

    // This kernel can ONLY run AFTER the k-th row kernel is complete
    // because it needs the updated pivot value A[k*n + k].
    float pivot_inv = 1.0f / A[k * n + k];

    float sum = 0.0f;
    for (int p = 0; p < k; p++) {
        sum += A[i * n + p] * A[p * n + k];
    }
    A[i * n + k] = (A[i * n + k] - sum) * pivot_inv;
}
```

EVERY work-item reads this!

Host Side (C++) 1: Setup the program

```
// 1. Discover available platforms (e.g., NVIDIA, Intel)
clGetPlatformIDs(...);
cl_platform_id platform = ...; // User selects a platform

// 2. Discover devices on that platform (e.g., a specific GPU)
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, ...);
cl_device_id device = ...; // User selects a device

// 3. Create a context to manage all OpenCL objects
cl_context context = clCreateContext(NULL, 1, &device, ...);

// 4. Create a command queue to send work to the device
cl_command_queue queue = clCreateCommandQueueWithProperties(context, device,
...);
```

Host Side Part 2: Preparing Kernels and Data

```
// 1. Load the kernel source code from the .cl file
std::string kernel_source = read_kernel_file("lu_kernels.cl");
const char* source_ptr = kernel_source.c_str();

// 2. Create and build the OpenCL program
cl_program program = clCreateProgramWithSource(context, 1, &source_ptr,
err); clBuildProgram(program, ...); // JIT compilation for the target GPU!
// ... (Error checking clGetProgramBuildInfo is critical here) ...

// 3. Extract the individual kernel functions from the program
cl_kernel row_kernel = clCreateKernel(program, "lu_row_kernel", ...);
cl_kernel col_kernel = clCreateKernel(program, "lu_column_kernel", ...);

// 4. Allocate memory buffer on the GPU device
cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_WRITE,
                            sizeof(float) * N * N, NULL, ...);

// 5. Copy our input matrix from host RAM to GPU VRAM
clEnqueueWriteBuffer(queue, d_A, CL_TRUE, 0,
                     sizeof(float) * N * N, h_A_working, ...);
```

Host Side Part 3: The Execution Loop

```
// Set the constant arguments for the kernels (device buffer and
clSetKernelArg(row_kernel, 0, sizeof(cl_mem), &d_A);
clSetKernelArg(row_kernel, 2, sizeof(int), &matrix_size);
// (repeat for col_kernel...)
```

```
for (int k = 0; k < N; k++) {
    cl_event row_event; // Used to chain commands

    // Set the changing argument 'k' for this iteration
    clSetKernelArg(row_kernel, 1, sizeof(int), &k);
    size_t global_work_size_row = N - k;
    if (global_work_size_row > 0) {
        clEnqueueNDRangeKernel(queue, row_kernel, 1, NULL, &global_work_size_row,
                               NULL, 0, NULL, &row_event); // Launch kernel
    }

    // Set 'k' for the column kernel and define the dependency
    clSetKernelArg(col_kernel, 1, sizeof(int), &k);
    size_t global_work_size_col = N - k - 1;
    if (global_work_size_col > 0) {
        clEnqueueNDRangeKernel(queue, col_kernel, 1, NULL, &global_work_size_col,
                               NULL, 1, &row_event, NULL); // Launch AFTER
row_event
    }
}
```

Host Side Part 4: Retrieving Results and Cleaning

```
// 1. Block the host thread until all queued commands on the GPU have
finished.This includes all 2*N kernel launches and their dependencies.
clFinish(queue);
clock_t end_ocl = clock(); // Stop the timer

// 2. Create a buffer in host memory for the result
float* h_Result = new float[N * N];

// 3. Copy the final decomposed matrix from GPU VRAM back to host RAM
err = clEnqueueReadBuffer(queue, d_A, CL_TRUE, 0,
                          sizeof(float) * N * N, h_Result, ...);

// 4. Now that the data is on the host, we can verify its correctness
if (h_Result) {
    verify_lu_decomposition(h_A_original, h_Result);
}

// 5. Clean up all OpenCL objects (clReleaseMemObject, clReleaseKernel, etc.)
// ..
```

Host programs can be “ugly”

```
int main() {
    srand(time(NULL));

    std::cout << "Setting up host data for a " << N << "x" << N << " matrix..." << std::endl;
    float* h_A_original = new float[N * N];
    initialize_matrix(h_A_original);
    float* h_A_working = new float[N * N];
    memcpy(h_A_working, h_A_original, sizeof(float) * N * N);

    cl_int err;

    cl_uint num_platforms;
    clGetPlatformIDs(0, NULL, &num_platforms);
    std::vector<cl_platform_id> platforms(num_platforms);
    clGetPlatformIDs(num_platforms, platforms.data(), NULL);

    std::cout << "Available Platforms:" << std::endl;
    for (cl_uint i = 0; i < num_platforms; ++i) {
        char platform_name[128];
        clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, sizeof(platform_name), platform_name, NULL);
        std::cout << "  " << i << ": " << platform_name << std::endl;
    }
    std::cout << "Select a platform: ";
    unsigned int plat_idx;
    std::cin >> plat_idx;
    cl_platform_id platform = platforms[plat_idx];

    cl_uint num_devices;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
    std::vector<cl_device_id> devices(num_devices);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, devices.data(), NULL);

    std::cout << "Available Devices:" << std::endl;
    for (cl_uint i = 0; i < num_devices; ++i) {
        char device_name[128];
        clGetDeviceInfo(devices[i], CL_DEVICE_NAME, sizeof(device_name), device_name, NULL);
        std::cout << "  " << i << ": " << device_name << std::endl;
    }
    std::cout << "Select a device: ";
    unsigned int dev_idx;
    std::cin >> dev_idx;
    cl_device_id device = devices[dev_idx];

    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &err); CL_CHECK(err);

    cl_command_queue queue = clCreateCommandQueueWithProperties(context, device, 0, &err); CL_CHECK(err);

    std::string kernel_source = read_kernel_file("lu_kernels.cl");
    const char* source_ptr = kernel_source.c_str();
    cl_program program = clCreateProgramWithSource(context, 1, &source_ptr, NULL, &err); CL_CHECK(err);
    err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
    if (err != CL_SUCCESS) {
        size_t log_size;
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
        std::vector<char> log(log_size);
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size, log.data(), NULL);
        std::cerr << "Build Log:\n" << log.data() << std::endl;
        exit(1);
    }

    cl_kernel row_kernel = clCreateKernel(program, "lu_row_kernel", &err); CL_CHECK(err);
    cl_kernel col_kernel = clCreateKernel(program, "lu_column_kernel", &err); CL_CHECK(err);
    cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * N * N, NULL, &err);
CL_CHECK(err);

```

A LOT OF
BOILERPLATE CODE!

```
    std::cout << "\n--- Running Naive OpenCL Implementation ---" << std::endl;

    err = clEnqueueWriteBuffer(queue, d_A, CL_TRUE, 0, sizeof(float) * N * N, h_A_working, 0, NULL, NULL);
CL_CHECK(err);
    int matrix_size = N;
    clSetKernelArg(row_kernel, 0, sizeof(cl_mem), &d_A);
    clSetKernelArg(row_kernel, 2, sizeof(int), &matrix_size);
    clSetKernelArg(col_kernel, 0, sizeof(cl_mem), &d_A);
    clSetKernelArg(col_kernel, 2, sizeof(int), &matrix_size);


    clock_t start_ocl = clock();

    for (int k = 0; k < N; k++) {
        cl_event row_event; // Event to track completion of the row kernel

        // Set arguments for the row kernel
        clSetKernelArg(row_kernel, 1, sizeof(int), &k);
        size_t global_work_size_row = N - k;
        if (global_work_size_row > 0) {
            // Enqueue the row kernel and associate it with row_event
            err = clEnqueueNDRangeKernel(queue, row_kernel, 1, NULL, &global_work_size_row, NULL, 0, NULL, &row_event);
            CL_CHECK(err);
        }


        // Set arguments for the column kernel
        clSetKernelArg(col_kernel, 1, sizeof(int), &k);
        size_t global_work_size_col = N - k - 1;

        if (global_work_size_col > 0) {
            // Enqueue the column kernel.
            // The '1, &row_event' part tells this kernel to wait for row_event to be signaled.
            err = clEnqueueNDRangeKernel(queue, col_kernel, 1, NULL, &global_work_size_col, NULL, 1, &row_event, NULL);
            CL_CHECK(err);
        }

        // It's good practice to release events when they are no longer needed in the wait list
        // to avoid resource leaks, especially in more complex scenarios.
        if (global_work_size_row > 0) {
            clReleaseEvent(row_event);
        }
    }

    // Wait for all enqueued commands to finish before stopping the timer.
    clFinish(queue);

    clock_t end_ocl = clock();

    double time_ocl = ((double)(end_ocl - start_ocl)) / CLOCKS_PER_SEC;
    double flops = (2.0 / 3.0) * pow(N, 3);
    double mflops_ocl = (flops / time_ocl) / 1e6;
    printf("Naive OpenCL Time: %f seconds\n", time_ocl);
    printf("Naive OpenCL Performance: %.2f MFLOPS\n", mflops_ocl);

    // READ RESULT BACK
    std::cout << "Reading result from GPU for verification..." << std::endl;
    float* h_Result = new float[N * N];
    err = clEnqueueReadBuffer(queue, d_A, CL_TRUE, 0, sizeof(float) * N * N, h_Result, 0, NULL, NULL); CL_CHECK(err);

    // Verify Results
    if (h_Result) {
        verify_lu_decomposition(h_A_original, h_Result);
    }

    // MUST DO CLEANUP REMEMBER...!!!!
    clReleaseMemObject(d_A);
    clReleaseKernel(row_kernel);
    clReleaseKernel(col_kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    delete[] h_A_original;
    delete[] h_A_working;
    if (h_Result) delete[] h_Result;

    return 0;
```

Benchmark Naive OpenCL Kernels

Case	**MFLOPS**	
	CPU	*GPU
Sequential C (not OpenCL)	714.8	N/A
Naive Kernels (OpenCL)	376.62	20,991.25

Maximum Absolute Error: 1.19e-07

*** Nvidia 4090**

The Next Step: Optimizing the Kernels

Our first OpenCL implementation was 28.5x faster than the CPU, which is a great start. But it's far from the theoretical peak performance of the GPU.

- **Problem 1: Kernel Launch Overhead**
  - For an N=2048 matrix, the host launches 2 * 2048 = 4096 kernels.
  - Each kernel launch has a small but significant CPU and driver overhead. This adds up.
- **Problem 2: Memory Bandwidth Saturation**
  - The kernels are memory-bound. They spend most of their time reading from and writing to slow global VRAM.
  - There is almost no data reuse. A value is read from global memory, used in one calculation, and then discarded. The key to GPU performance is to read data once into fast memory and reuse it many times.

Algorithim Change: Block LU Decomposition

- Partition a large matrix M into four blocks.
- Goal: factor M into block lower L and block upper U.

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

Main Advantage: For large matrices, computers can work on the different blocks at the same time

Target factorization:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}.$$

Assumption

Top-left block A is invertible.

Step 2: Solving for the Matrix Blocks

Multiply the block factors and match blocks:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}.$$

1. From $A = L_{11}U_{11}$: LU-decompose A to get L_{11}, U_{11}.
2. From $B = L_{11}U_{12}$: $U_{12} = L_{11}^{-1}B$.
3. From $C = L_{21}U_{11}$: $L_{21} = C\,U_{11}^{-1}$.

Step 3: The Schur Complement

From the bottom-right block,

$$D = L_{21}U_{12} + L_{22}U_{22} \quad \Rightarrow \quad L_{22}U_{22} = D - L_{21}U_{12}.$$

Substitute L_{21} and U_{12}:

$$L_{22}U_{22} = D - (CU_{11}^{-1})(L_{11}^{-1}B) = D - C(L_{11}U_{11})^{-1}B.$$

Schur Complement

Define $S := \boxed{D - CA^{-1}B}$ (since $A = L_{11}U_{11}$ invertible).
LU-decompose S to obtain L_{22}, U_{22}.

A Hybrid CPU+GPU Block Algorithm

- Step 1: Decompose Diagonal Panel *(A → LU)*
  - This is a small LU decomposition (32x32). It's serial and complex.
  - Best suited for the CPU. We'll *copy the panel from GPU to CPU*, decompose it, and *copy it back*.
- Step 2: Update Row/Column Panels
  - These are triangular matrix solves (**TRSM**). They are highly parallel.
- Step 3: Update Trailing Matrix *(D - $CA^{-1}B$)*
  - This is a large General Matrix-Matrix Multiplication (**GEMM**). This is the most computationally expensive part **($O(N^3)$)**.
- The Big Idea: This approach trades thousands of small, inefficient operations for a few, large, highly optimized ones (TRSM and GEMM) that can saturate the GPU's computational resources.

Synchronization

On a GPU, we have two levels of synchronization:

- **Within a Work-Group (Easy & Fast):**
  - We can use a `barrier()` to create a rendezvous point.
  - When a work-item hits a `barrier()`, it pauses until every other work-item in its group also reaches that barrier.
  - This is essential for safely sharing data through `__local memory`. A common pattern is:
    - All work-items cooperatively load data into `__local memory`.
    - `barrier(CLK_LOCAL_MEM_FENCE);` // Wait for all loads to complete.
    - All work-items can now safely read the shared data for computation.
- **Across Work-Groups (Hard & Explicit):**
  - There is **NO** direct way for different work-groups to communicate or synchronize within a single kernel launch.
  - The hardware scheduler can execute work-groups in any order, concurrently or serially.
- Solution: If you need global synchronization, you must end the current kernel and launch a new one. The host API (using events) becomes your global synchronization tool.

Be careful with barriers

- A `barrier()` must be encountered by all work-items in a group, or none at all. Placing a barrier inside a branch is a common and dangerous source of bugs.

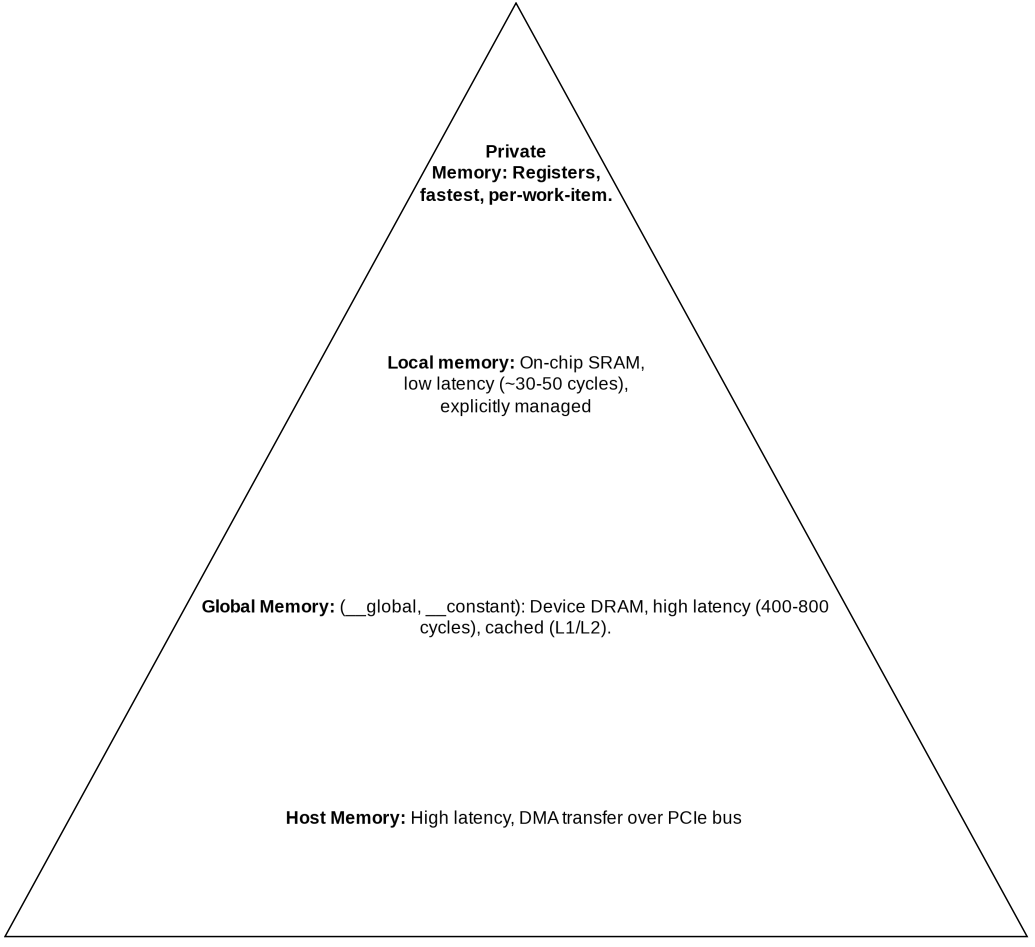
This will cause a deadlock and hang your GPU:

```
1 // WORK-GROUP HANGS!
2 if (get_local_id(0) == 0) {
3     // Only ONE work-item enters this branch...
4     // ...but it will wait FOREVER for the others to
5     //arrive.
6     barrier(CLK_LOCAL_MEM_FENCE);
7 }
```

This is correct:

```
1 int some_value = ...;
2 // ALL work-items check the condition.
3 if (condition_is_uniform_for_the_group)
4 {   // ... do work ...
5     barrier(CLK_LOCAL_MEM_FENCE);
6     // ... do more work ...
7 }
```

Memory Model Again :)

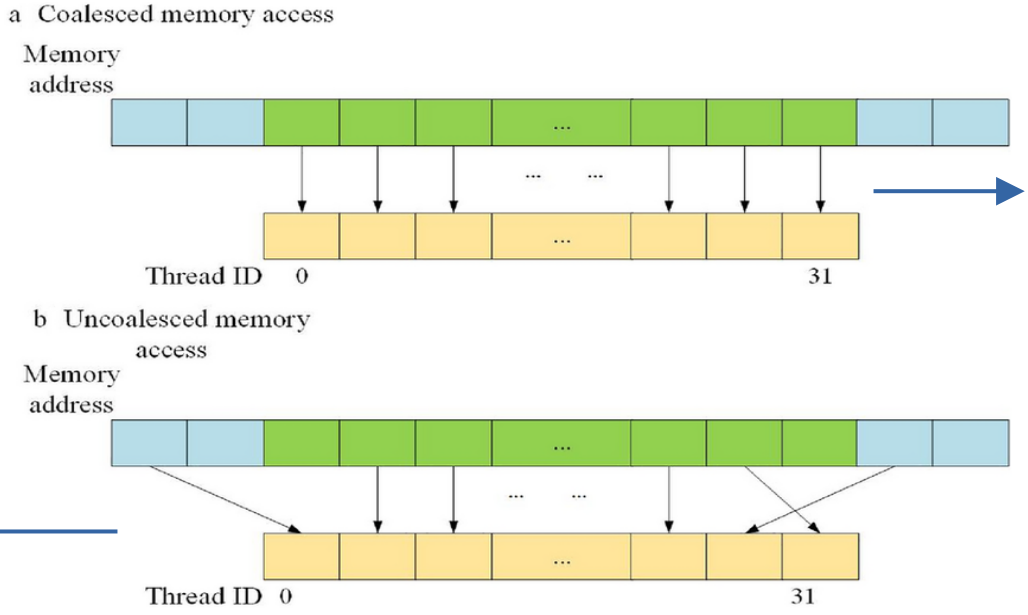


Caching Panels in __local Memory

- **The Problem**: The naive kernels were memory-bound. They spent all their time waiting for data to arrive from slow global VRAM.
- USE the __local memory. The strategy is:
  - Work-items in a group **cooperatively load** a block of data from global to local memory.
  - They **synchronize** to ensure the load is complete.
  - They perform all calculations using the **fast** local memory copy.

Coalesced Loads for Maximum Bandwidth

- **The Problem**: Just *using* __local memory isn't enough. We must fill it from global memory as fast as possible. The GPU's memory bus is extremely wide, and we need to use all of it.
- **The Solution:** Memory Coalescing. When adjacent work-items (e.g., IDs 0, 1, 2, 3...) access adjacent memory locations, the hardware combines these individual reads into a single, wide, and highly efficient transaction.



Work-items read from a contiguous block

Work-iteams read from random locations

Registers & The Strategic Uncoalesced Read

- **The Problem:** Even __local memory has some latency. For data that is used extremely heavily by a single work-item, we want it in the absolute fastest storage: **private registers**.
- **The Solution:** The OpenCL compiler is smart. A small, compile-time-sized array declared inside a kernel will often be mapped directly to registers.

```
// 1. This small array will be stored in ultra-fast private registers
float x[BLOCK_SIZE];

// 2. Load the column data from global memory into registers
for(int i = 0; i < BLOCK_SIZE; i++) {
    // This access is strided by N. IT IS NOT COALESCED and looks very slow!
    x[i] = A[(k + i) * n + global_j];
}

// 3. The Payoff: Perform the entire substitution using only registers and
//local memory.
for (int i = 0; i < BLOCK_SIZE; i++) {
    float sum = 0.0f;
    for (int p = 0; p < i; p++) {
        // x[p] is read from a register. L_panel is from local memory. ALL
FAST!   sum += L_panel[i][p] * x[p];
    }
    x[i] = x[i] - sum; // Write to a register
}
```

BLOCK_SIZE is typically something like 32

Sizing Work for High Occupancy

- The goal is to keep the Compute Units Busy!!!
  - Have more active **Subgroups** than the GPU can execute at once
    - If a singular subgroup stalls (while waiting for memory)
      - Schedular instantly swaps in another that is ready to compute.
- **Occupancy**: the ratio of active Subgroups on a CU to the maximum Subgroups that CU can support.
- The Trade-off: There is a finite amount of resources (registers, __local memory) per CU.
  - Large Work-Groups:
    - Pro: Can enable more data sharing in __local memory.
    - Con: Consume more resources
  - Small Work-Groups:
    - Pro: Consume fewer resources,
    - Con: May not be able to cache enough data in __local memory to be effective.

TEST AND PROFILE !!!

Work-Item Divergence

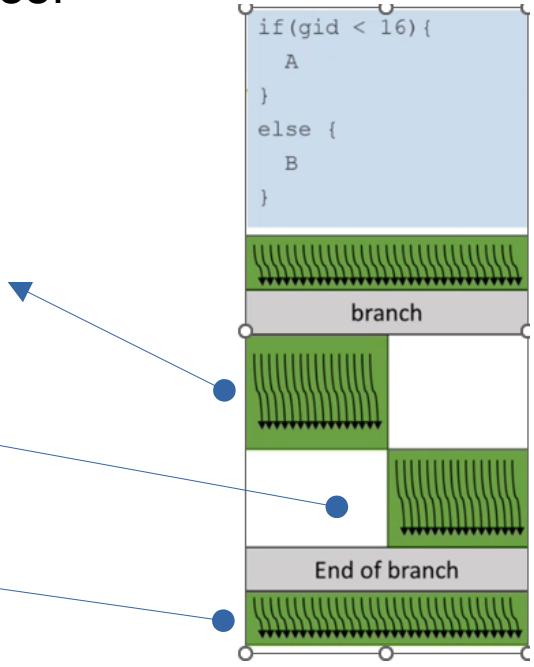
The Concept: All work-items in a Subgroup must execute the same instruction at the same time. What happens when the code has an if/else statement?

- If some work-items in a Subgroup need to enter the if block and others need to enter the else block, this is called **Branch Divergence**.

First execute the if path, disabling all the work items that didn't take it

Then, do the else path, disabling the work-items that took the if path.

The total time taken is the sum of both paths.



TSRM Kernels

Using __local Memory as a User-Managed Cache

Synchronize

Using Private Memory

```
__kernel void lu_row_update_kernel(__global float* A, const int k, const int n)
{
    __local float L_panel[BLOCK_SIZE][BLOCK_SIZE];

    int local_id = get_local_id(0);

    // Cooperative load of the L_panel. Every work-item in a group helps.
    if (local_id < BLOCK_SIZE) {
        for (int j = 0; j < BLOCK_SIZE; j++) {
            L_panel[local_id][j] = A[(k + local_id) * n + (k + j)];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    // Each work-item is responsible for one column of the output panel.
    int global_j = k + BLOCK_SIZE + get_global_id(0);
    if (global_j >= n) return;

    // Load column into private memory
    float x[BLOCK_SIZE];
    for(int i = 0; i < BLOCK_SIZE; i++) {
        x[i] = A[(k + i) * n + global_j];
    }

    // Perform forward substitution on private data
    for (int i = 0; i < BLOCK_SIZE; i++) {
        float sum = 0.0f;
        for (int p = 0; p < i; p++) {
            sum += L_panel[i][p] * x[p];
        }
        // Unit diagonal for L is assumed
        x[i] = x[i] - sum;
    }

    // Write result back to global memory
    for(int i = 0; i < BLOCK_SIZE; i++) {
        A[(k + i) * n + global_j] = x[i];
    }
}
```

GEMM Kernel

Filling the Cache Efficiently

Checks for matrices whose size isn't a perfect multiple of BLOCK_SIZE. This is a necessary source of divergence.

```
__kernel void gemm_update_kernel(__global float* A, const int k, const int n) {
    int local_row = get_local_id(1);
    int local_col = get_local_id(0);
    int group_row = get_group_id(1);
    int group_col = get_group_id(0);

    // Top-left corner of the destination tile this work-group is responsible for
    int dest_tile_row_start = k + BLOCK_SIZE + group_row * BLOCK_SIZE;
    int dest_tile_col_start = k + BLOCK_SIZE + group_col * BLOCK_SIZE;

    __local float L_tile[BLOCK_SIZE][BLOCK_SIZE];
    __local float U_tile[BLOCK_SIZE][BLOCK_SIZE];

    // Load the tile from the L panel (A_ik)
    L_tile[local_row][local_col] = A[(dest_tile_row_start + local_row) * n + (k +
local_col)];

    // Load the tile from the U panel (A_kj)
    U_tile[local_row][local_col] = A[(k + local_row) * n + (dest_tile_col_start +
local_col)];

    barrier(CLK_LOCAL_MEM_FENCE);

    // Standard local memory GEMM computation
    float accumulator = 0.0f;
    for (int p = 0; p < BLOCK_SIZE; ++p) {
        accumulator += L_tile[local_row][p] * U_tile[p][local_col];
    }

    // Calculate the final global memory address to write to
    int final_global_row = dest_tile_row_start + local_row;
    int final_global_col = dest_tile_col_start + local_col;

    // Boundary check and write result
    if (final_global_row < n && final_global_col < n) {
        A[final_global_row * n + final_global_col] -= accumulator;
    }
}
```

Host Code 1: The Hybrid Main Loop

The main loop now advances in large BLOCK_SIZE steps.

We use clEnqueueRead/WriteBufferRect to transfer only the small diagonal panel, minimizing slow PCIe traffic.

clFinish() is called only once

```
// The loop now iterates over BLOCKS, not elements
for (int k = 0; k < N; k += BLOCK_SIZE) {
    // === CPU WORK PHASE ===

    // 1. Fetch the diagonal panel from the GPU where it currently
    //lives
    clEnqueueReadBufferRect(queue, d_A, CL_TRUE, /* src_origin */,
                            /* dst_origin */, ..., h_panel, ...);

    // 2. Perform the small, serial LU decomposition on the CPU
    cpu_panel_decompose(h_panel, BLOCK_SIZE);

    // 3. Send the factorized panel back to the GPU
    clEnqueueWriteBufferRect(queue, d_A, CL_TRUE, /* dst_origin */,
                            /* src_origin */, ..., h_panel, ...);

    // === GPU WORK PHASE (details on next slide) ===

    // 4. Enqueue all the parallel GPU work for this step
    enqueue_gpu_kernels_for_step(k);
}

// 5. After the loop queues EVERYTHING, block the host and wait for the
//GPU to finish.
clFinish(queue);

```

Host Code 2: Asynchronous Kernels with Events

- Defining a Dependency Graph
- Maximizing Concurrency
- Non-Blocking Host

```
1 // Inside the main loop, for a given 'k':
2 // An array to hold our synchronization objects
3 cl_event update_events[2];
4
5 // 1. Launch the row panel solver. It can start immediately.
6 //    The completion of this kernel will signal update_events[0].
7 err = clEnqueueNDRangeKernel(queue, row_update_kernel, ...,
8 &update_events[0]);
9 // 2. Launch the column panel solver. It can also start immediately
10 //    and run CONCURRENTLY with the row kernel.
11 //    Its completion will signal update_events[1].
12 err = clEnqueueNDRangeKernel(queue, col_update_kernel, ...,
13 &update_events[1]);
14 // 3. Launch the massive GEMM update kernel.
15 //    This is the crucial dependency: we tell this kernel it must WAIT for
16 //    the TWO events in our update_events array before it can start.
17 err = clEnqueueNDRangeKernel(queue, gemm_kernel, ..., 2, update_events, NULL);
18
19
```

Benchmark Optimised OpenCL Kernels

Case	**MFLOPS**	
	CPU	*GPU
Sequential C (not OpenCL)	714.8	N/A
Naive Kernels (OpenCL)	376.62	20,991.25
Block Hybrid (Synchronous)	-	1,339,874.37

Maximum Absolute Error (|A - L*U|): 1.708984e-03

Approaching Peak Performance

Our block-hybrid model is already impressive, achieving over 1.3 TFLOPS. But there are still two remaining bottlenecks:

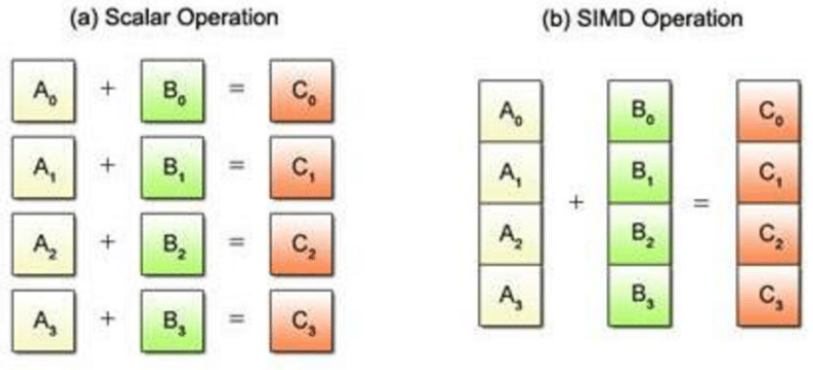
- Inside the Kernel: Our gemm_kernel still has a relatively low arithmetic intensity. Each work-item reads a lot from __local memory relative to the amount of math it does.
- On the Host: Our main loop is synchronous. The GPU waits for the CPU to process a panel, and the CPU waits for the GPU to finish its work. There is still "dead time" where one processor is idle.

To achieve the next level of performance, we must attack both of these issues simultaneously.

GPU: Vectorization (SIMD)

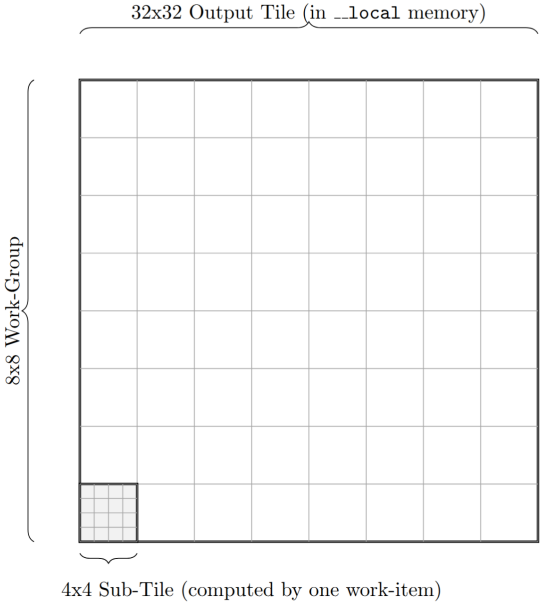
At their core, GPUs are SIMD (Single Instruction, Multiple Data) processors. Their hardware is not designed to operate on single numbers (scalars) but on short vectors of numbers all at once.

- Scalar Operation (Inefficient):
  - float a = b * c; -> 1 Instruction, 1 Multiplication
- Vector Operation (Efficient):
  - In OpenCL, we can use built-in vector types like float4, float8, or float16.
  - float4 a = b * c; -> 1 Instruction, 4 Multiplications!



Optimisation 2: Register Blocking

- **The Old Way:** In our previous gemm_kernel, each work-item in a 32x32 work-group computed one (i, j) output element.
- **The New Idea:** What if we make each thread responsible for a small 2D tile of the output? For a 32x32 output tile, we can use an 8x8 work-group where each of the 64 threads computes its own private 4x4 sub-tile.



GEMM Kernel 2

Unroll the loop, compiler does this

```
#define TILE_DIM 4      // Each work-item handles a 4x4 tile
#define WGS 8           // The Work-Group is 8x8 = 64 items

__kernel void gemm_update_kernel_register_blocked(...) {
    // 1. POINTER VECTORIZATION: Treat global memory as a grid of
    //float4's
    __global float4* A_vec = (__global float4*)A;

    // 2. REGISTER BLOCKING using a vector type.
    //    'accum' is an array of FOUR float4's, holding all 16 results in
    //registers.
    float4 accum[TILE_DIM];
    for(int i = 0; i < TILE_DIM; ++i) { accum[i] = (float4)(0.0f); }

    // 3. VECTORIZED LOAD into __local memory for maximum bandwidth.
    //    Each work-item loads a float4 vector.
    *(__local float4*)&L_tile[...] = A_vec[...];
    barrier(CLK_LOCAL_MEM_FENCE);

    // 4. VECTORIZED COMPUTATION in the main loop
#pragma unroll
    for (int p = 0; p < BLOCK_SIZE; ++p) {
        // Fetch a full float4 vector from the U tile
        float4 u_vec = *(__local float4*)&U_tile[p][local_col *
TILE_DIM];
        // Fetch 4 L values for the 4 rows this thread is responsible for
        float l_vals[TILE_DIM];
        for(int i=0; i<TILE_DIM; ++i) { l_vals[i] = L_tile[...][p]; }

        // The PAYOFF: Fused-Multiply-Add on 4 floats at once.
        // The scalar l_vals[i] is broadcast to a vector before the FMA.
        for(int i=0; i<TILE_DIM; ++i) {
            accum[i] = fma((float4)l_vals[i], u_vec, accum[i]);
        }
    }
    // 5. VECTORIZED WRITE from registers back to global memory
    A_vec[...] -= accum[i];
}

```

Host Code 1: Enabling True Asynchronicity with Pinned Memory

The Problem: Standard C++ new float[...] memory is "pageable." For the GPU to access it, the driver must first copy it to a "pinned," non-pageable staging buffer, and then DMA it to the GPU. This is an implicit, blocking copy.

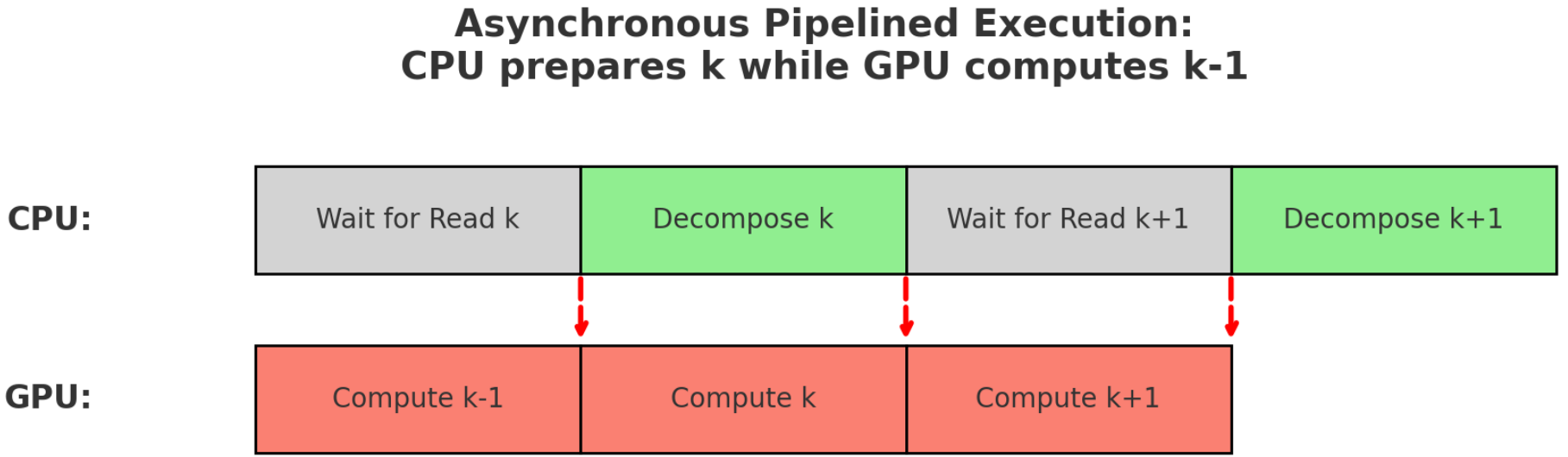
- The Solution: Pinned (or Page-Locked) Memory.

  We can ask the OpenCL driver to allocate host memory that the GPU's DMA engine can access directly. This eliminates the staging copy and is the key to enabling non-blocking data transfers.

```
1 // Allocate a buffer on the host, but flag it so the GPU can see it directly
2 cl_mem pinned_buffer = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR, size,
3 ...);
4 // Get a normal C++ pointer to this pinned memory region
5 float* h_panel = (float*)clEnqueueMapBuffer(queue, pinned_buffer, ...);
6
7 // Now, transfers using this h_panel pointer can be truly asynchronous.
8 clEnqueueReadBufferRect(..., h_panel, ...); // Can return immediately!
9
```

Host Code 2: The Pipelined Main Loop

To hide all latency, we use two command queues (one for transfers, one for compute) and completely restructure the main loop into a pipeline. While the CPU processes panel k, the GPU can be busy computing the kernels for panel k-1.



Host Code 2: The Pipelined Main Loop (Continued)

```
// Create TWO queues: one for transfers, one for compute
cl_command_queue compute_queue = ...;
cl_command_queue transfer_queue = ...;

for (int k = 0; k < N; k += BLOCK_SIZE) {
    // 1. Asynchronously start reading panel 'k' from GPU to pinned memory.
    //    This is non-blocking and returns instantly.
    clEnqueueReadBufferRect(transfer_queue, ..., CL_FALSE, ..., &read_event);

    // 2. If this isn't the first loop, enqueue all the compute kernels for the
    //    PREVIOUS iteration (k-1). They will run while the CPU waits for step 3.
    if (k > 0) {
        // All these kernels are submitted to the COMPUTE queue and
        // depend on the WRITE event from the previous iteration.
        clEnqueueNDRangeKernel(compute_queue, ...);
    }

    // 3. This is the only host-side wait. The CPU pauses here until panel 'k'
    // has finished transferring from the GPU.
    clWaitForEvents(1, &read_event);

    // 4. While the GPU is now busy on step 2, the CPU does its work on panel'k'.
    cpu_panel_decompose(h_panels[...], BLOCK_SIZE);

    // 5. Asynchronously start writing the decomposed panel 'k' back to the GPU.
    //    This is also non-blocking. Save the 'write_event' for the next
    // iteration.
    clEnqueueWriteBufferRect(transfer_queue, ..., CL_FALSE, ..., &write_event);
    last_write_event = write_event;
}
clFinish(compute_queue); // Final sync
```

The Results of Full Optimization

Case	MFLOPS	
	CPU	*GPU
Sequential C (not OpenCL)	714.8	N/A
Naive Kernels (OpenCL)	376.62	20,991.25
Block Hybrid (Synchronous)	-	1,339,874.37
Pipelined & Register-Blocked		3,044,024

Maximum Absolute Error (|A - L*U|):1.870495e-01

Comparing GEMM Kernels

Case	**TFLOPS**	
	CPU	*GPU
Pipelined & Register-Blocked	N/A	18.8
CLBlast GEMM Kernel		41.3

Learning Resources

- SimpleOpenCLSamples
- C++ Concurrency in Action [Anthony Williams]
- https://github.com/ProjectPhysX/FluidX3D
- **Data Parallel C++**
- **https://sycl.tech/**
- **hands-on-opencl**