Data Processing on Modern Hardware

Tutorial 8

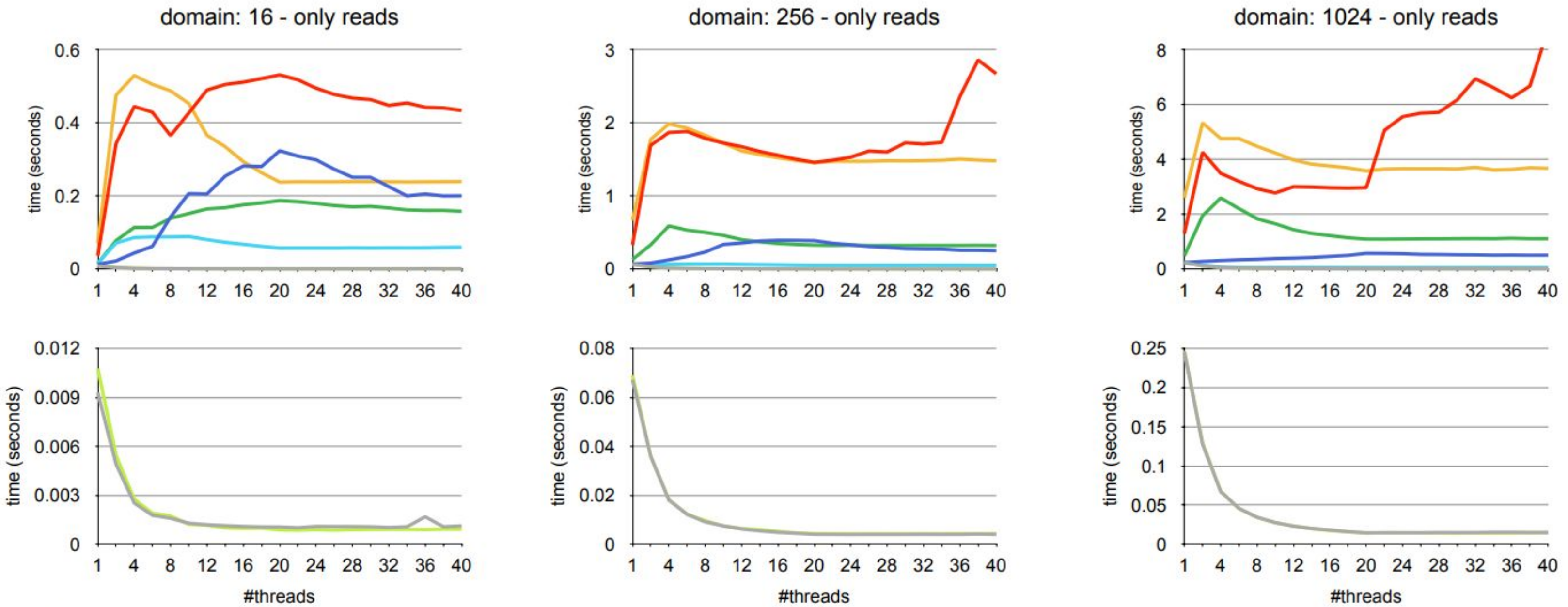
Ferdinand Gruber

Michalis Georgoulakis

We would be grateful if you take a few minutes to fill in the course evaluation you have received by email.

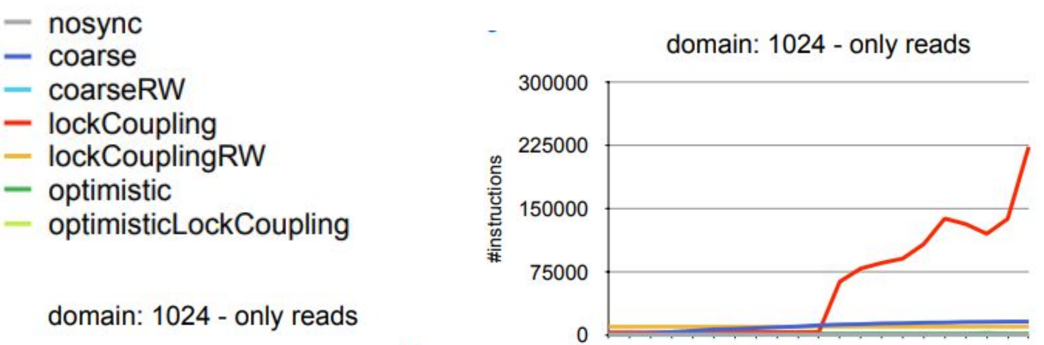
Assignment 5 - Synchronization

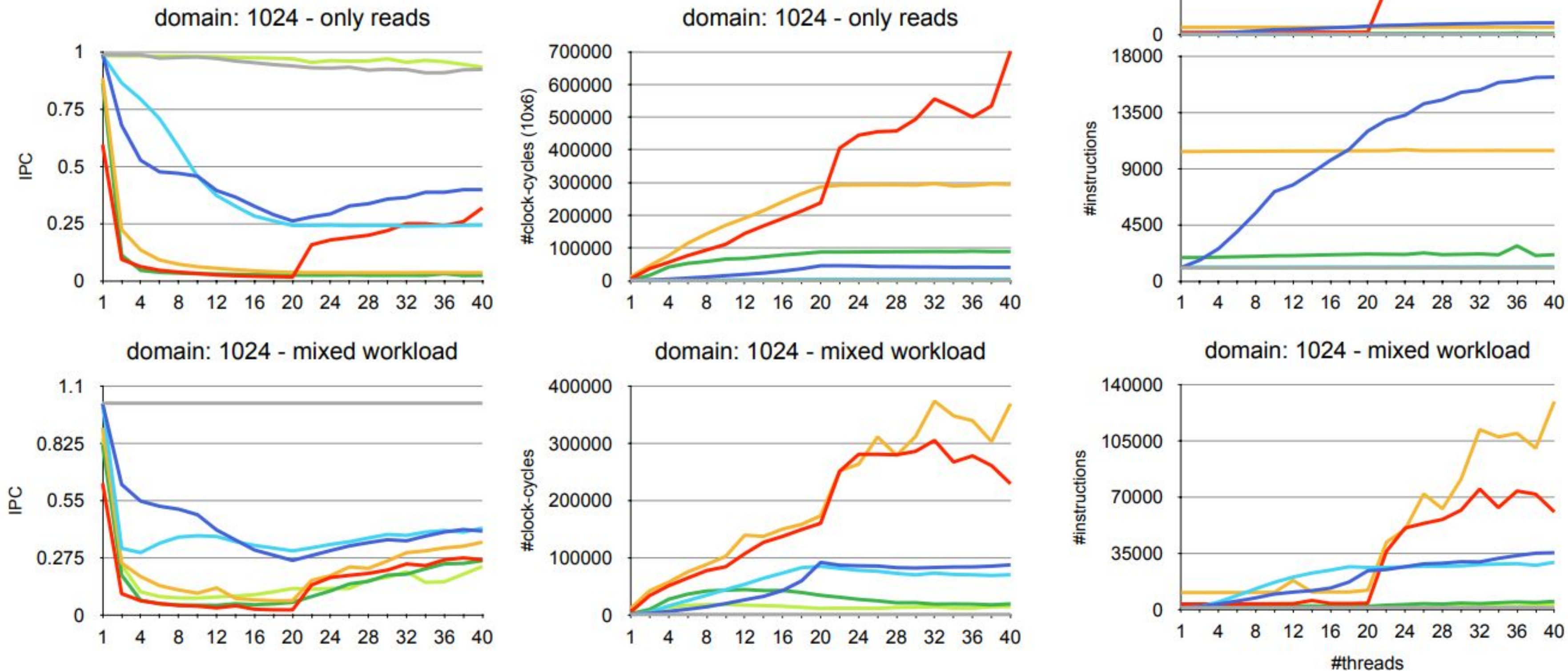
Latency Comparison - Read-Only Workloads



Latency Comparison - Mixed Workloads

Perf Metrics





Conclusions

- Assignment Workload difficult to tune with synchronization as:
  - list operations are cheap
  - compared to the operations, locking is indeed expensive

- Fine-grained locking approaches produce a high overhead
  - especially lock coupling (unexpectedly), probably due to the number of locks - low IPC
  - coarse is decent for low domain: locking period is short
  - coarse-grained waiting times add up with multiple threads

- Optimistic Methods deliver the best performance among the synchronized methods
  - Optimistic Lock Coupling can even outperform single-threaded no sync approach
  - It also is the only method with quite low CPU cycles - minimal waiting overhead

Optimistic Lock Coupling

“The first mode is similar to a traditional mutex and excludes other threads by physically acquiring the underlying lock.“

“In the second mode, reads can proceed optimistically by validating a version counter that is embedded in the lock (similar to optimistic concurrency control). “

The second mode can not be achieved by a conventional mutex. What we expected to see was the first mode, as exemplified in the previous tutorial’s slides.

Optimistic Lock Coupling: contains()

```
struct Entry {
  T key;
  Entry *next;
  uint64_t version = 0; // Version counter for the lock coupling
  M mutex;              // Mutex for each element in the list
};

Entry staticHead;
Entry staticTail;
```

```
bool contains(T k) {
  start:
  Entry *pred;
  Entry *curr = &staticHead;
  uint64_t lastVersion = curr->version;

  while (curr->key < k) {
    pred = curr;                          // Make current the predecessor
    curr = curr->next;                    // Go to the next element
    if (pred->version != lastVersion) // Validate using version
      goto start;
    lastVersion = curr->version; // Update for the new element
  }
  return (curr->key == k);
}
```

Optimistic Lock Coupling: insert()

```
struct Entry {
  T key;
  Entry *next;
  uint64_t version = 0; // Version counter for the lock coupling
  M mutex;              // Mutex for each element in the list
};

Entry staticHead;
Entry staticTail;
```

```
void insert(T k) {
  start:
  Entry *pred;
  Entry *curr = &staticHead;
  uint64_t predVersion;
  uint64_t currVersion = curr->version;

  while (curr->key < k) {
    pred = curr;                       // Make current the predecessor
    curr = curr->next;                 // Go to the next element
    predVersion = currVersion;         // Make currVersion the predVersion
    currVersion = curr->version;       // Update for the new element
    if (pred->version != predVersion) // Validate using version
      goto start;
  }

  pred->mutex.lock(); // Lock the predecessor of the relevant element
  curr->mutex.lock(); // Lock the relevant element

  if (pred->version != predVersion) {
    pred->mutex.unlock();
    curr->mutex.unlock();
    goto start;
  }

  if (curr->key != k) {              // Is the element in the list?
    auto *n = new Entry{k, curr}; // Create the element
    pred->next = n;                  // And insert it
    pred->version++;                 // Update the version due to changes
  }
  pred->mutex.unlock();
  curr->mutex.unlock();
}
```

Optimistic Lock Coupling

remove()

```
struct Entry {
  T key;
  Entry *next;
  uint64_t version = 0; // Version counter for the lock coupling
  M mutex;              // Mutex for each element in the list
};

Entry staticHead;
Entry staticTail;
```

```
void remove(T k) {
  start:
  Entry *pred;
  Entry *curr = &staticHead;
  uint64_t predVersion;
  uint64_t currVersion = curr->version;

  while (curr->key < k) {
    pred = curr;                      // Make current the predecessor
    curr = curr->next;                // Go to the next element
    predVersion = currVersion;        // Make currVersion the predVersion
    currVersion = curr->version;      // Update for the new element
    if (pred->version != predVersion) // Validate using version
      goto start;
  }

  pred->mutex.lock(); // Lock the predecessor of the relevant element
  curr->mutex.lock(); // Lock the relevant element

  if (pred->version != predVersion) {
    pred->mutex.unlock();
    curr->mutex.unlock();
    goto start;
  }

  if (curr->key == k) {        // Is the element in the list?
    pred->next = curr->next;   // Then, remove it.
    pred->version++;           // Update the version due to changes
    curr->version++;           // Update the version due to changes
  }
  pred->mutex.unlock();
  curr->mutex.unlock();
}
```

Landscape of Parallel Computing APIs

- **OpenCL**
  - Open standard by Khronos Group for heterogeneous computing
  - Portable across CPUs, GPUs, DSPs, FPGAs
- **CUDA**
  - NVIDIA-proprietary API for NVIDIA GPUs
  - Rich ecosystem (Thrust, cuDNN, etc.) and mature tooling
- **Vulkan Compute**
  - Explicit low-overhead graphics & compute API
  - Fine-grained control over memory and execution on GPUs
- **Other Approaches**
  - **SYCL:** Higher-level C++ wrapper over OpenCL
  - **DirectCompute:** Microsoft's compute API for Windows
- **When to Choose?**
  - Portability / Learning Curve: OpenCL/SYCL
  - Performance & Ecosystem: CUDA (on NVIDIA hardware)
  - Graphics + Compute Integration: Vulkan Compute

OpenCL Execution Model

Open Standard by Khronos Group

- Vendor-neutral API for parallel programming
- Widely adopted across hardware manufacturers (Gimp, Affinity, Sony Vegas, legacy Photoshop)

Host vs. Device Code

- **Host:** C/C++ (or bindings) running on CPU
- **Device:** Kernels (C-like) compiled for GPU/accelerator

Portability Across Architectures

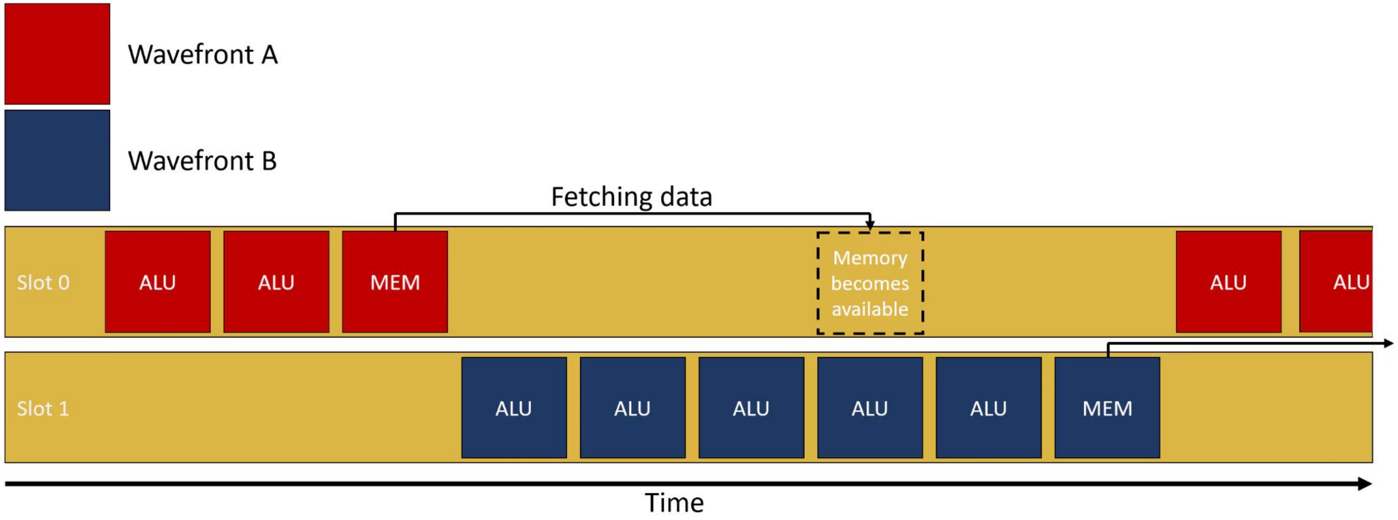
- Write once, run on CPUs, GPUs, DSPs, FPGAs
- Single source for heterogeneous platforms

OpenCL Execution Model

- **Platform → Device → Context → Command Queue**
  - **Platform:** Implementation vendor (e.g., NVIDIA, AMD)
  - **Device:** Compute unit (GPU, CPU core)
  - **Context:** Manages objects (buffers, programs) across devices
  - **Command Queue:** Orders work for a specific device
- **Work-Items & Work-Groups**
  - **Work-Item:** Single execution instance of a kernel
    1. Execute completely independently (aside from any explicit synchronization) and map directly to the hardware's SIMD lanes or scalar threads.
  - **Work-Group:** Group of work-items sharing local memory and synchronization
    1. Execute on the same compute unit (CU) or multiprocessor
    2. Share a region of local memory (__local)
    3. Can synchronize with each other via barrier(CLK_LOCAL_MEM_FENCE)
- **NDRange**
  - Defines the grid ("global size") and grouping ("local size")

Wavefront (AMD) / Warp (NVIDIA) Width

- GPUs don't really run each work-item entirely independently—they bundle them into fixed-size SIMD "lanes" that execute in lock-step:
- AMD calls this a **wavefront**, typically 64 work-items wide.
- NVIDIA calls it a **warp**, typically 32 threads wide.



OpenCL Memory Model

Global Memory

- Accessible by all work-items across all work-groups
- **Characteristics:** High latency (100s of cycles), large capacity (GBs)
- **Best Practice:**
  - Read large chunks coalesced (e.g., `float4` loads)
  - Minimize random accesses

Constant Memory

- Read-only region of global memory, cached on device for fast lookups
- **Characteristics:** Low-latency on repeated reads, limited size (tens of KB)
- **Example Use:** Lookup tables (e.g., color conversion coefficients)

Local Memory

- Shared among work-items in the **same** work-group
- **Characteristics:** Low latency (close to registers), limited size (tens of KB)

Private Memory

- Per-work-item registers or stack space
- **Characteristics:** Fastest access, very limited size
- **Example Use:** Temporary accumulators or loop indices

Data Transfer and Access Patterns

Data-Transfer Patterns

- **Host ↔ Global Memory** via explicit enqueues
  `clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0, size, hostA, 0, NULL, NULL);`
  `clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0, size, hostC, 0, NULL, NULL);`

- **Optimization Tips:**
  - **Batch transfers** (fewer, larger copies rather than many small ones)
  - **Asynchronous transfers** with events to overlap compute and copy
  - **Use pinned (page-locked) host memory** for higher PCIe bandwidth

Access Pattern Examples

- **Coalesced Access:** consecutive `get_global_id(0)` threads load contiguous floats → full bus utilization
- **Strided Access (Anti-pattern):** `data[i * stride]` with large `stride` → many small transactions, low throughput

CUDA Syntax – Kernels and Memory

Defined by `__global__ <signature>`

Invoked with `<kernel_name><<<X,Y>>>`

Memory allocation

- Static allocation `__device__ int X`
- Dynamic allocation `cudaMalloc(&X, size)`

Memory copying (Host to Device, Device to Host)

- `cudaMemcpy`
- `cudaMemcpyManaged`

```
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__ void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride){
    y[i] = y[i] + x[i];
  }
}

// Run kernel on 1GB elements on  on the GPU
add<<<1, 256>>>(N, gx, gy);
```

```
// Initialize pointers and allocate memory
float *gx, *gy;
cudaMalloc((void**)&gx, N*sizeof(float));
cudaMalloc((void**)&gy, N*sizeof(float));
```

```
// Move data from CPU to GPU
cudaMemcpy(gx, hx, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(gy, hy, N*sizeof(float), cudaMemcpyHostToDevice);

// Move data from GPU to CPU
cudaMemcpy(hy, gy, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Questions?