day / date:

OpenCL → Vector Addition Example #1

function declaration → __kernel void vector_add (__global const int* A, __global const int* B, __global const int* C)
memory pointer in global memory; read only vectors; write only

```
int id = get_global_id(0);   → loop ; current threads unique index.
C[id] = A[id] + B[id]        → addition.
```

```
int main() {
```

input vector:
```
std::vector<float> a {1.0f, 2.0f, 3.0f, 4.0f}
std::vector<float> b = {4.0f, 3.0f, 2.0f, 1.0f}
```

output vector:
```
std::vector<float> c (a.size());
```

```
std::vector<cl::Platform> platforms;     available opencl platforms. (NVIDIA, INTEL)
cl::Platform::get(&platforms)            populates vector platform.
cl::Platform platform = platforms[0];    chooses first available platform.
```

```
std::vector<cl::Device> devices;         gets all GPU devices available on selected platform.
platform.getDevices(CL_DEVICE_TYPE_GPU, &devices)   → GPU | CPU | ALL
cl::Device device = devices[0]           chooses first available device.
```

```
cl::Context context(device);             manages memory buffers and kernels for selected device.
```

```
cl::Program program(context, "kernel.cl", true)   loads file, builds it, if the build fails it throws an exception. true means read only.
program.build({device});
```

copied from host to device.

```
cl::buffer bufA(context, CL_MEM_READ_ONLY, a.size()*sizeof(float), a.data());
cl::buffer bufB(context, CL_MEM_READ_ONLY, b.size()*sizeof(float), b.data());
cl::buffer bufC(context, CL_MEM_READ_ONLY, c.size()*sizeof(float));
```

day / date:

cl::CommandQueue queue(context, device) — all commands such as memory transfer and kernel execution are done through this.

cl::Kernel kernel(program, "vecadd")) ← function name in kernel.cl

kernel.setArg(0, bufA)

kernel.setArg(1, bufB)

kernel.setArg(2, bufC)

queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(a.size()), cl::NullRange)
(global offset (starts at 0); local workgroup size.)
(global worksize (4 items))

queue.enqueueReadBuffer(bufC, CL_TRUE, 0, c.size() * sizeof(float), c.data());
reads result from device memory (bufC) to host memory (c)

```
for(size_t i=0; i<c.size(); i++)
    cout << a[i] << " + " << b[i] << " = " << c[i] << endl;
```

day / date:

CUDA → Vector Addition Example #2.

```
#define N (2048*2048)          total number of elements in vector
#define THREADS_PER_BLOCK 512  number of CUDA threads / block.

runs on device
__global__ void add(int *a, int *b, int *c)   can be called from host.
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
            (index of block in grid)  (threads index within its block)
  (index of element this thread will process)
  (no. of threads in each block)
  if (idx < N)
    c[idx] = a[idx] + b[idx]   checks if within vector bounds.

int main(){                    runs on CPU.
  int *a, *b, *c;              host copies
  int *d_a, *d_b, *d_c;        device copies.
  int size = N * sizeof(int)

  (takes a pointer to device pointer)
  cudaMalloc((void**)&d_a, size);   allocate memory on GPU device.
  cudaMalloc((void**)&d_b, size);
  cudaMalloc((void**)&d_c, size);

  a = (int*)malloc(size); random_ints   allocate memory on the host (CPU)
  b = (int*)malloc(size); random_ints   (to initializes vector with random integers.
  c = (int*)malloc(size)

  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);   copy data from host array to GPU device memory.
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
                                        (tells CUDA the direction of copy.

  add<<<N/THREADS_PER_BLOCK>>>(d_a, d_b, d_c);   Launch kernel.
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);   copy from device to host.
  free(a); free(b); free(c)                 frees host memory
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);   frees device memory.
```

day / date:

MPI → parallel sum of an array

1. get a chunk of the array.
2. compute local sum.
3. send local sum to root process.
4. root process will compute the final total sum.

1. int MPI_Bcast (void* buf, int count, MPI_Datatype dtype, int root, MPI_Comm comm)

addressof send/receive. — Co number of elements. — Co sender — CoWORLD

- contents are copied from a sender to all other processes.

2. int MPI_Scatter (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

element sent to each process — datatype — Co elements received. — Co 0/sender — Co WORLD.

- sends chunks of array to each process

3. int MPI_Gather (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

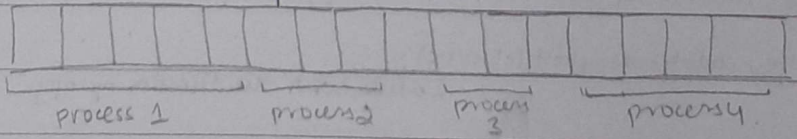
element sent to root process. — Co elements received by root

- root process receives data from all processes from send buffers.
- data is received in order of the process number.

4. int MPI_Scatterv (const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype dtype, void* recvbuf, int recvcount, MPI_Datatype dtype, int root, MPI_Comm comm)

address of send buffer. — array, number of element sent to each — array, index of which point data is going to get transferred.

- sendcounts = (5, 3, 2, 4) displs = (0, 5, 9, 12)



day / date:

5. int MPI_Gatherv (const void *sendbuf, int sendcount, MPI_Datatype dtype,
void* recvbuf, (elements from each process) const int recvcounts[], (which index to place data) const int displs[], MPI_Datatype dtype,
int root, int MPI_Comm comm)

6. int MPI_Allgather (const void* sendbuf, (number of element to send to root) int sendcount, MPI_Datatype dtype,
void* recvbuf, int recvcount, MPI_Datatype, MPI_Comm comm)
(number of element received from any process)

7. int MPI_Allgatherv (const void *sendbuf, (number of elements to send) int sendcount, MPI_Datatype dtype,
void* recvbuf, const int recvcounts[], (array, # of elements received from each process) const int displs[], (array, index of place incoming data) MPI_Datatype dtype,
MPI_Comm comm)

8. int MPI_Alltoall (const void* sendbuf, (elements to each process) int sendcount, MPI_Datatype dtype,
void* recvbuf, int recvcount, MPI_Datatype dtype, MPI_Comm comm);
(to receive from each process)

• each process sends equal-size data to all other processes and receives from all others.

9. int MPI_AlltoAllv (const void* sendbuf, const int sendcounts[], (displacement into send buffer) const int sdispls[], MPI_Datatype dtype,
void* recvbuf, (receive buffer) const int recvcounts[], (from each process) const int rdispls[], (displacement into receive buffer) MPI_Datatype dtype,
MPI_Comm comm);

5. int MPI_Reduce (void* sendbuf, void* recvbuf, (number of elements in send buf) int count, MPI_Datatype dtype,
MPI_Op op, (operation (+, -)) int root, MPI_Comm comm)

• count, op, root have to be equal in all processes.

int MPI_Allreduce (void* sbuf, void* rbuf, int count, MPI_Datatype dtype, MPI_Op op, MPI_Comm comm)

• returns result value to all processes.

day / date:

12. int MPI_Init (int *argc, char ***argv) — initialization in main

13. int MPI_Comm_size (MPI_Comm_WORLD, &size) — · returns size of communicator · number of processes.

14. int MPI_Comm_rank (MPI_Comm_WORLD, &rank) — · rank of a process in a communicator starting from 0

15. int MPI_Finalize()

B

16. MPI_Send (void *data, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
(count: number of elements to be sent; dest: rank; tag: identifier of sender)

17. MPI_Recv (void *data, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status)
(source: identifier; tag: to receive tag.)

NB

18. MPI_Isend (buf, count, datatype, dest, tag, comm, request)

19. MPI_Irecv (" " " " " " ")

20. MPI_Wait (request, status) — · blocks until send/receive with the desired request is done.

21. MPI_Test (request, flag, status) — · to see if communication is completed.

22. MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)

23. int MPI_Barrier (MPI_Comm_WORLD) — · synchronizes all processes.

day / date:

OpenMP

1. #pragma omp parallel for
2. omp_set_num_threads (4) — 4 thread based parallel region.
3. #pragma omp parallel if (scalar_expression) — if expression is true
4. #pragma omp parallel private ( ) — each thread has its own copy, by default is shared.
5. #pragma omp parallel firstprivate (list) — initialized with the value the variable had before entering the construct.
6. #pragma omp parallel for lastprivate — thread that executes the final iteration of the loop updates the value
7. #pragma omp parallel default (private) shared (list)
8. #pragma omp parallel default (shared) private (list)
9. omp_get_thread_num ( )
10. #pragma omp for schedule ( )
    - static: chunks of specified size
    - dynamic: chunks are assigned when thread finishes previous chunks.
11. #pragma omp critical — if a thread is executing in CS, any other thread attempting to enter will be blocked.
12. #pragma omp atomic — only applies to the update of a memory location.
13. #pragma omp master
    - executed only by the master thread of the team.
    - all other threads skip this.
14. #pragma omp barrier
    - a thread will wait, until all other threads have reached this point.
15. #pragma omp operator: list
    - a private copy for each thread is initialized.
16. #pragma omp parallel reduction ( )

day / date:

- What is PDC?
  - → use of multiple processors or machines working together on a common task.
- Bit-level parallelism
  - → based on increasing processor word size, reduces the number of instructions the processor must execute.
- Instruction-level parallelism
  - → simultaneous execution of multiple instructions from a program.
- Technology Push
  - → Moore's law states that number of transistors doubled every 2 years., meaning more parallelism.
  - → single core processors had overheating issues, huge power consumption.
  - → computes are no longer faster, they just have more parallelism.
  - → data-parallel computing is the most scalable solution.
  - → Multi-core CPU chip : the cores fit on a single processor socket.
- Application Pull
  - → atmosphere, earth, science, etc.
- Parallelizing Applications
  - → Traditionally, code to be run on a single compute having a single CPU, instruction executed one after another.
  - → parallel execution now is to be run using multiple CPUs, a problem broken into discrete parts that can be solved concurrently
- Parallelization strategy.
  1. Problem Understanding
     - → check for code dependencies, communication, synchronization.
     - → know the hotspots and ignore where there is less CPU usage.
     - → solutions: restructure the program, different algorithm, overlap communication with computation.
  2. Partitioning / Decomposition.
     - → divide work into chunks, tasks executed concurrently.
     - → fine-grain : large number of small task.
     - → coarse-grain : small number of large task.

day / date:

→ Domain Decomposition: each parallel task works on a portion of data.

1D: block is same sized large blocks, cyclic is small sized many blocks.

2D: rows and columns

→ Functional Decomposition: computation that is to be performed, problem decomposed according to the work, can be split into different tasks.

3. Assignment

→ composing fine-grained into processes or coarse-grained tasks.

→ check for load balance, uniform communication, ease of synchronization.

4. Orchestration / Mapping

→ inter task communication, synchronization among tasks, data locality aspect.

- Embarrasingly Parallel
  - → so straightforward
  - → very little or no inter task communication
- communications can bring waiting time for synchronization.
- inter task communication implies overhead.
- latency is the time taken to send a message from point A to point B.
- bandwith is the amount of data that can be communicated per unit time.
- efficient way: package small messages into large message.
- in distributed, comunications are explicit and vice versa for shared.
- synchronous communication
  - → blocking communication.
- asynchronous communication
  - → non-blocking communication
- synchronization
  - → barrier: each task comes to halt.
  - → lock/semaphore: only one task uses lock and releases as well.
  - → synchronous communication.

day / date:

- Flynn's Taxonomy.
  - → SISD: single processor, single instruction stream, data stored in single memory.
  - → SIND: parallel processor, all processing unit execute same instructions, but on different data element.
  - → NISD: each processor executes different instruction sequence, using same data.
  - → MIMD: simultaneously execute different instructions. e.g. grid, cloud, clusters.
    - to shared memory: SMP, NUMA
    - to distributed memory: ~~NUMA~~ clusters
- Symmetric Multiprocessor (SMP)
  - → processors share memory.
  - → communicate via bus.
  - → same memory access time.
  - → work can be done in parallel, failure of single processor does not halt the system, adding additional processors enhance performance.
- Non-Uniform memory access (NUMA)
  - → access times of different regions of memory differs.
  - → all processes have access to all parts of memory.
  - → different processors access different regions of memory at different speeds.
  - → SMP limits number of processors.
  - → effective performance at higher level of parallelism.
- Distributed memory
  - → each processor has access to its memory only.
  - → data transfer between processors is explicit.
  - → clusters are loosely coupled.
  - → high performance, high availability.
  - → scalability
  - → grid computing are heterogenous computer over the whole world providing CPU power
  - → geographically distributed services.
  - → cloud computing is a network based computing that takes place over the internet.
  - → a group of integrated and networked hardware called a platform.
  - → they are on demand services

day / date:

→ supercomputers that leads the world in terms of processing capacity, speed of calculation

→ computer speed measured in FLOPs.