SYCL GPU Best Practices

Ben Ashbaugh (Intel)

CC BY-SA 4.0 licensed presentation

Occupancy

Give the GPU enough work!

CC BY-SA 4.0 licensed presentation

GPU != CPU

- CPUs: Optimized for latency
  - Fewer fancier processors
- GPUs: Optimized for throughput
  - Many simpler processors

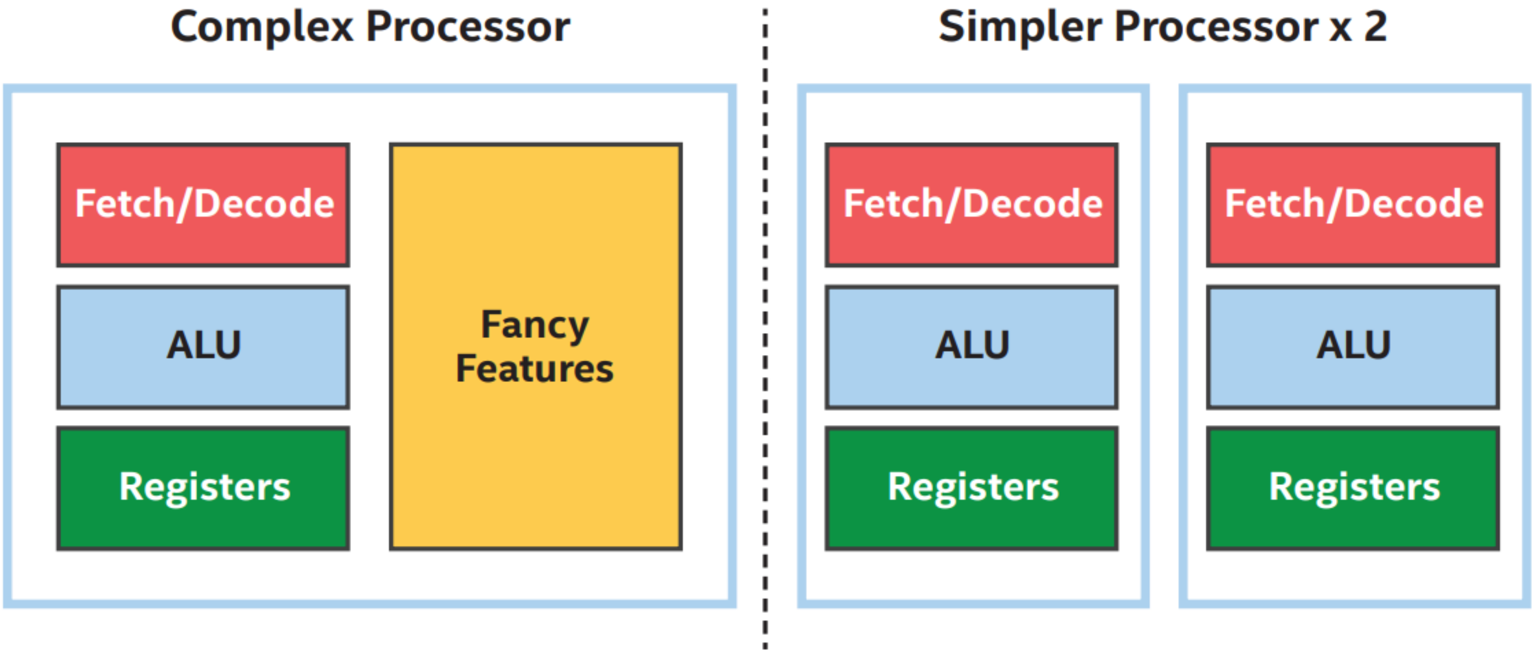

Figure 15-2. *GPU processors are simpler, but there are more of them*

CC BY-SA 4.0 licensed presentation

GPUs need lots of data!

- Graphics and games requires processing millions of pixels per second
- GPUs prefer many work-items for general-purpose computation
- Many work-items keeps GPU execution resources busy (occupied)!

CC BY-SA 4.0 licensed presentation

SYCL Academy

Bad: Not Enough Parallelism

```
h.single_task([=]() {
  for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n++) {
      T sum = 0;
      for (int k = 0; k < K; k++)
        sum += matrixA[m * K + k] * matrixB[k * N + n];
      matrixC[m * N + n] = sum;
    }
  }
});
```

Figure 15-3. *A single task matrix multiplication looks a lot like CPU host code*

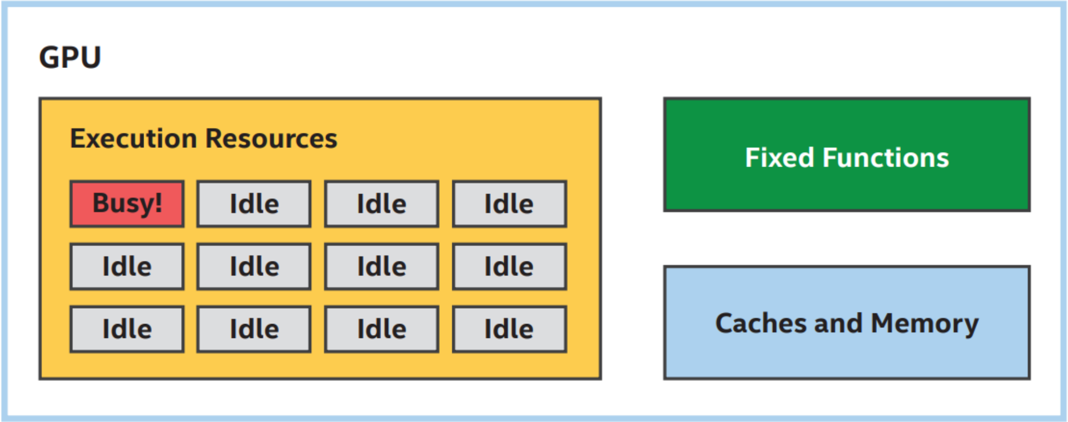

Figure 15-4. *A single task kernel on a GPU leaves many execution resources idle*

CC BY-SA 4.0 licensed presentation

Better: More Parallelism

```
h.parallel_for(range{M}, [=](id<1> idx) {
  int m = idx[0];

  for (int n = 0; n < N; n++) {
    T sum = 0;
    for (int k = 0; k < K; k++)
      sum += matrixA[m * K + k] * matrixB[k * N + n];
    matrixC[m * N + n] = sum;
  }
});
```

Figure 15-5. *Somewhat-parallel matrix multiplication*

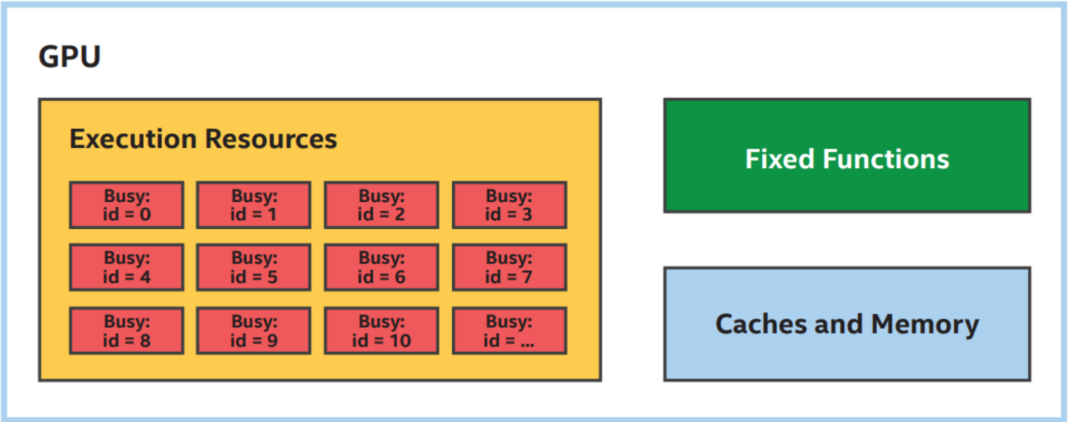

Figure 15-6. *Somewhat-parallel kernel keeps more processor resources busy*

CC BY-SA 4.0 licensed presentation

More Parallelism is Even Better!

```
h.parallel_for(range{M, N}, [=](id<2> idx) {
  int m = idx[0];
  int n = idx[1];
  T sum = 0;
  for (int k = 0; k < K; k++)
    sum += matrixA[m * K + k] * matrixB[k * N + n];
  matrixC[m * N + n] = sum;
});
```

***Figure 15-7.** Even more parallel matrix multiplication*

CC BY-SA 4.0 licensed presentation

Why is so much parallelism important?

- In addition to keeping processing elements busy…

- Many GPU processors operate on multiple data elements simultaneously:

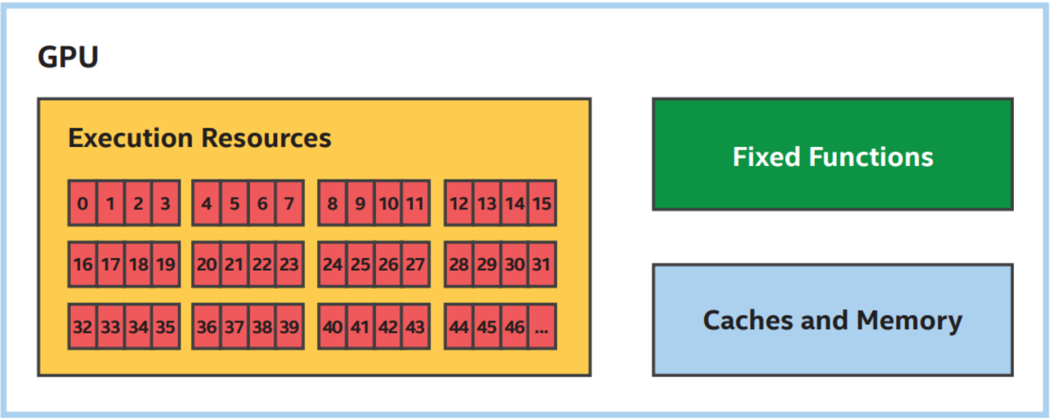

***Figure 15-9.** Executing a somewhat-parallel kernel on SIMD processors*

- Many GPU processors use multiple threads to hide latency:

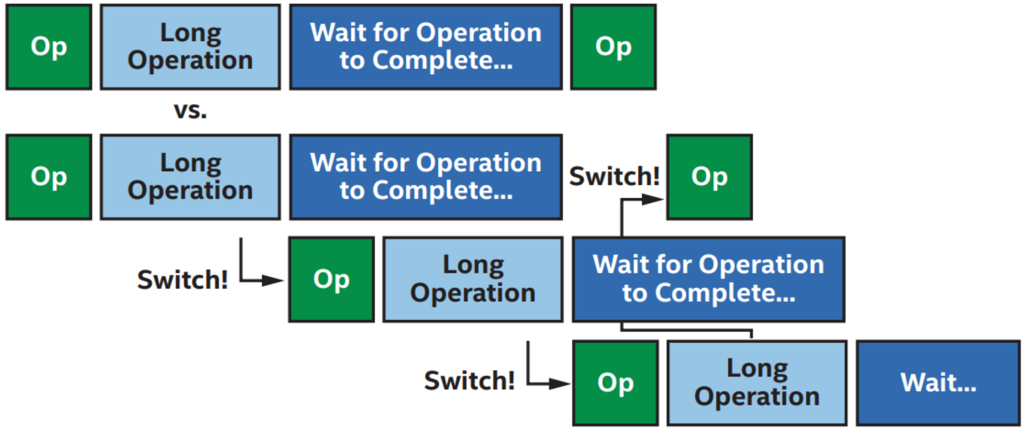

***Figure 15-13.** Switching instruction streams to hide latency*

CC BY-SA 4.0 licensed presentation

SYCL Academy

Memory

Keep the GPU fed and happy!

CC BY-SA 4.0 licensed presentation

Beware the Costs of Offload

- Moving data to or from a GPU is not free
- Moving an algorithm to or from a GPU is not always profitable

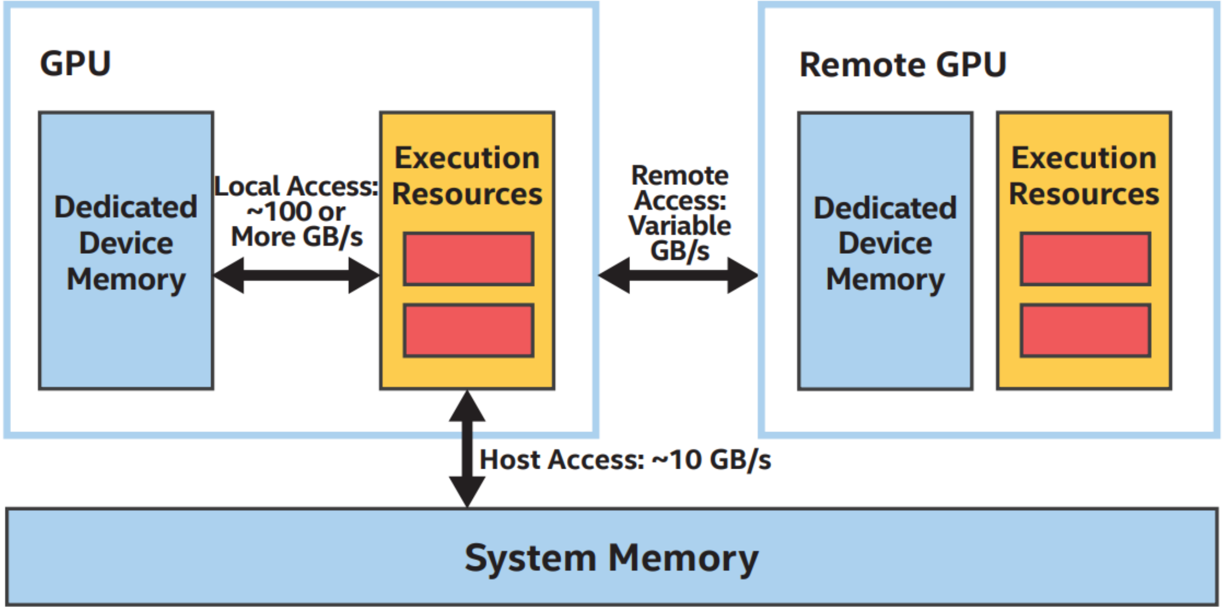

Figure 15-16. *Typical differences between device memory, remote memory, and host memory*

CC BY-SA 4.0 licensed presentation

Accessing Memory in Kernels

- Maximize performance by maximizing locality
- Typically means organizing data structures efficiently
- Or, choosing the right dimension to parallelize
- Not sure which dimension is best? Try both and profile!

CC BY-SA 4.0 licensed presentation

Bad: Strided Memory Accesses

```
h.parallel_for(range{M}, [=](id<1> idx) {
  int m = idx[0];

  for (int n = 0; n < N; n++) {
    T sum = 0;
    for (int k = 0; k < K; k++)
      sum += matrixA[m * K + k] * matrixB[k * N + n];
    matrixC[m * N + n] = sum;
  }
});
```

***Figure 15-5.** Somewhat-parallel matrix multiplication*

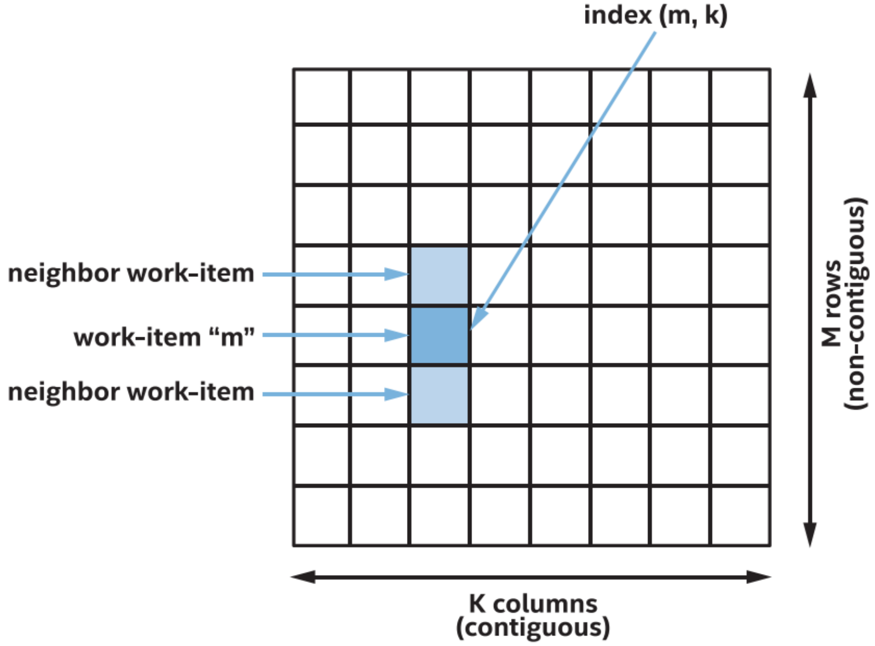

***Figure 15-17.** Accesses to `matrixA` are highly strided and inefficient*

CC BY-SA 4.0 licensed presentation

SYCL Academy

Good: Contiguous Memory Accesses

```
// This kernel processes columns of the result matrix in parallel.
h.parallel_for(N, [=](item<1> idx) {
  int n = idx[0]

  for (int m = 0; m < M; m++) {
    T sum = 0;
    for (int k = 0; k < K; k++)
      sum += matrixA[m * K + k] * matrixB[k * N + n];
    matrixC[m * N + n] = sum;
  }
});
```

***Figure 15-18.** Computing columns of the result matrix in parallel, not rows*

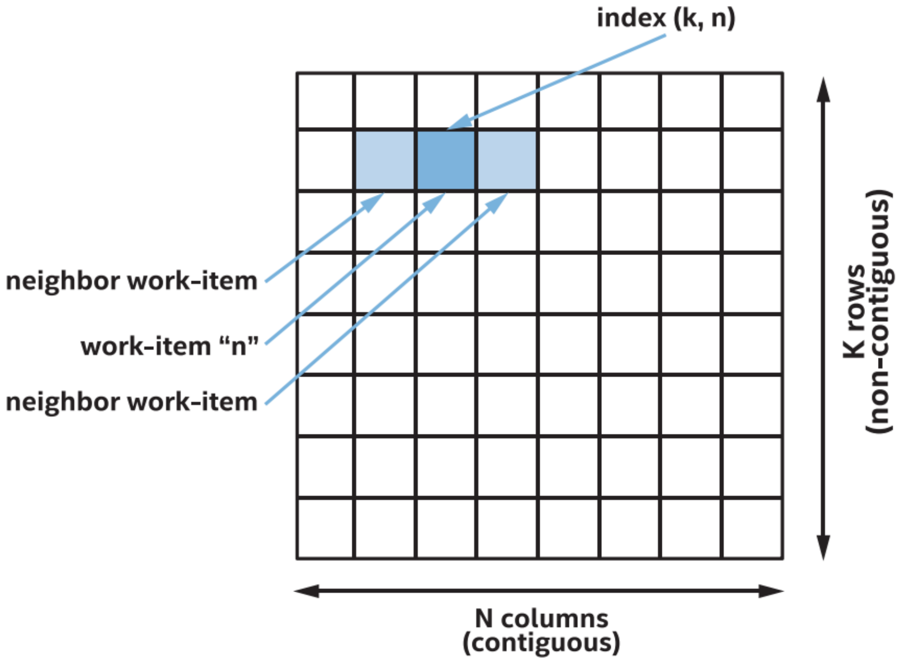

***Figure 15-19.** Accesses to `matrixB` are consecutive and efficient*

CC BY-SA 4.0 licensed presentation

Local Memory

- Beyond the scope of this talk
- Know that local memory supports more access patterns efficiently than global memory!

	Global Memory:	Local Memory:
`ptr[ id ]`	Full Performance!	Full Performance!
`ptr[ id + 1 ]`	Lower Performance	Full Performance!
`ptr[ id * 2 ]`	Lower Performance	Lower Performance
`ptr[ id * N ]`	Worst Performance	Worst Performance
`ptr[ id * M ]`	Worst Performance	Full Performance!

Figure 15-20. *Possible performance for different access patterns, for global and local memory*

CC BY-SA 4.0 licensed presentation

Compute

Cater to GPU strengths!

CC BY-SA 4.0 licensed presentation

Optimizing for GPU Compute

- GPUs are traditionally optimized for graphics tasks
- Typically, 32-bit floating-point math on vertices or pixels

CC BY-SA 4.0 licensed presentation

GPU Compute Best Practices

- Prefer smaller data types, 32-bits or less
- Some 64-bit data types may be slow or unsupported (e.g., double)
- Watch for 64-bit address arithmetic, use of 64-bit size_t
- 16-bit or smaller data types may be faster (e.g., half-precision fp16)
- Use MAD or FMA, don't disable contractions
- Trade precision for performance using sycl::native math functions
- Trade portability for performance using vendor-specific extensions

- Special operations found in various devices, including GPUs: **MAD** = multiply-add; **FMA** = Fused multiply add
  There use can change the results slightly (typically more accurate, but also different);
  they are generally are much faster than using the individual instructions.

CC BY-SA 4.0 licensed presentation

SYCL Academy

Summary and Recap

CC BY-SA 4.0 licensed presentation

SYCL Academy

Summary

- GPUs are massively parallel throughput devices
  - Important to give a GPU lots of work, thousands or millions of work-items
- Optimize Memory First
  - Monitor and minimize data transfer costs to or from the GPU
  - Improve performance by rearrange data or computation for locality
- Optimize Compute Next
  - Prefer smaller data types, trade precision for performance

- Thank you!

ben.ashbaugh@intel.com, @bashbaug

CC BY-SA 4.0 licensed presentation