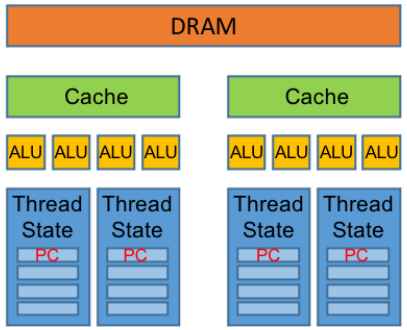
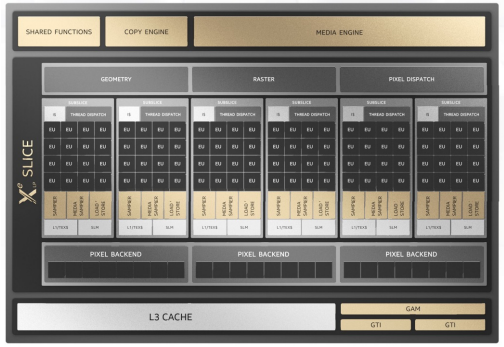
Lecture 21: Heterogeneous Parallel Programming

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture (Recap)

- Multicore processors are latency oriented, whereas GPUs are throughput oriented

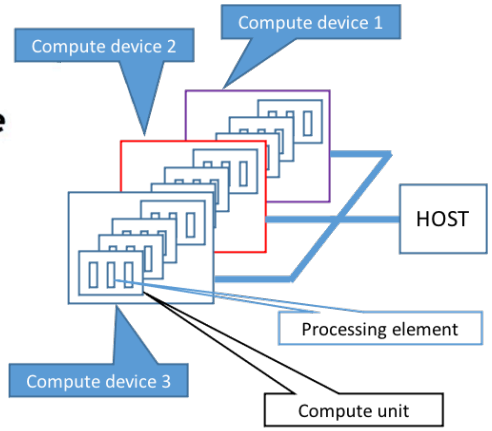





```
int* a = new int[size];
int* b = new int[size];
// create 'a' vector on the GPU
compute::vector<int> vector_a(size, context);
// create 'b' vector on the GPU
// copy data from the host to the device
compute::future<void> future_a = compute::copy_asyn
    a, a+size, vector_a.begin(), queue
);
// wait for copy to finish
future_a.wait();
// Create function defining the body of kernel
BOOST_COMPUTE_FUNCTION(int, vector_sum, (int x), {
    return x * 1.9 ;
});
// Launch the computation on the GPU using
// the command queue created above
compute::transform(
    vector_a.begin(),
    vector_a.end(),
    vector_b.begin(),
    vector_sum
);
// transfer results back to the host array 'c'
compute::copy(vector_b.begin(), vector_b.end(), b);
```

Executing OpenCL Programs

1. Query host for OpenCL devices
2. Create a context to associate OpenCL devices
3. Create programs for execution on one or more associated devices
4. Select kernels to execute from the programs
5. Create memory objects accessible from the host and/or the device
6. Copy memory data to the device as needed
7. Provide kernels to command queue for execution
8. Copy results from the device to the host



Today's Class

- Amdahl's law revisited
- Integrated CPU-GPU architectures
- Runtime solution for hybrid CPU-GPU parallelism

Amdahl's Law Revisited

- Recall, maximum possible speedup for a program having fraction “f” of its total execution “W” parallelizable on N processing resources
  - Speedup(f, N) = $T_{Sequential}$ / ($T_{Sequential}$ + $T_{Parallel}$) = W / {(1-f)W + fW/N}
  - Speedup(f, N) = 1 / { (1-f) + f/N }
- The assumption is that there is uniform scalability of processing resources when using more processors

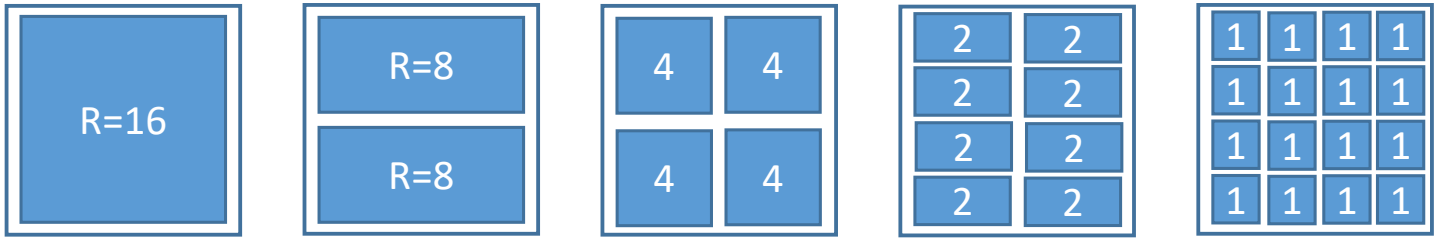
Simple Hardware Model

- Processor chip hardware roughly partitioned into
  - Multiple Cores (with L1 caches)
  - The Rest (L2/L3 cache banks, interconnect, etc.)
    - Uncore elements
  - Changing Core Size/Number does **NOT** change **The Rest**
- Resources for multiple cores are bounded
  - Bound of **N** resources per chip for cores
  - Due to area, power, cost ($$$), or multiple factors
- Single core performance can be improved by adding more of the bounded resources (**N**)
  - E.g., **N** could be total number of transistors on a processor
  - Also known as Base Core Elements (BCE)

Hill and Marty 2008

Symmetric Cores

- Examples of different possible multicore processors that are using the same number of total processing resources
  - Each processor has identical cores (**symmetric processor**)
  - Each processor is using the same number of resources (**N=16**)
  - Each core is consuming **R** number of these resources
    - Hence, cores per processor is **N/R**



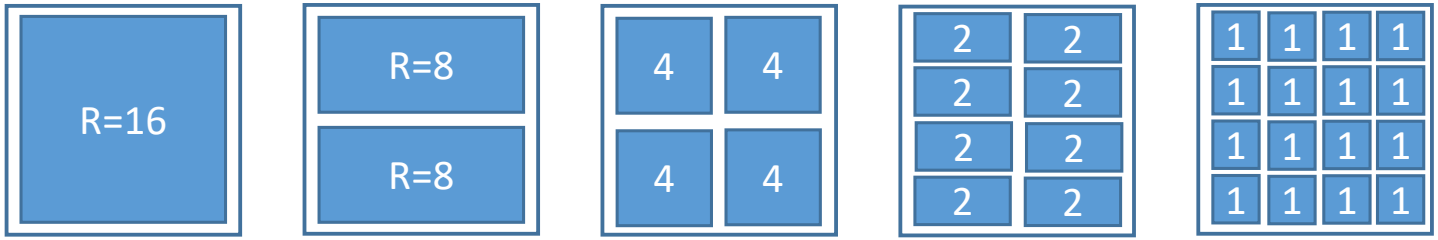
Hill and Marty 2008

Simple Hardware Model – Performance

- **$Perf_R$ <R**, where $Perf_R$ is performance of each core with R BCEs in a multicore processor with total N BCEs
  - **$Perf_R$ >=R** implies adding more cores will always enhance performance
    - Cannot happen due to the sharing of Uncore resources (The Rest)
- We assume $Perf_R$ =Square root of R *(avoiding cube root as it implies performance gains diminishes faster with increasing core count)*
  - For R=4, $Perf_R$ = 2 for each core
  - For R=9, $Perf_R$ = 3 for each core
  - For R=16, $Perf_R$ = 4 for each core

Amdahl's Law Revisited (Symmetric Cores)

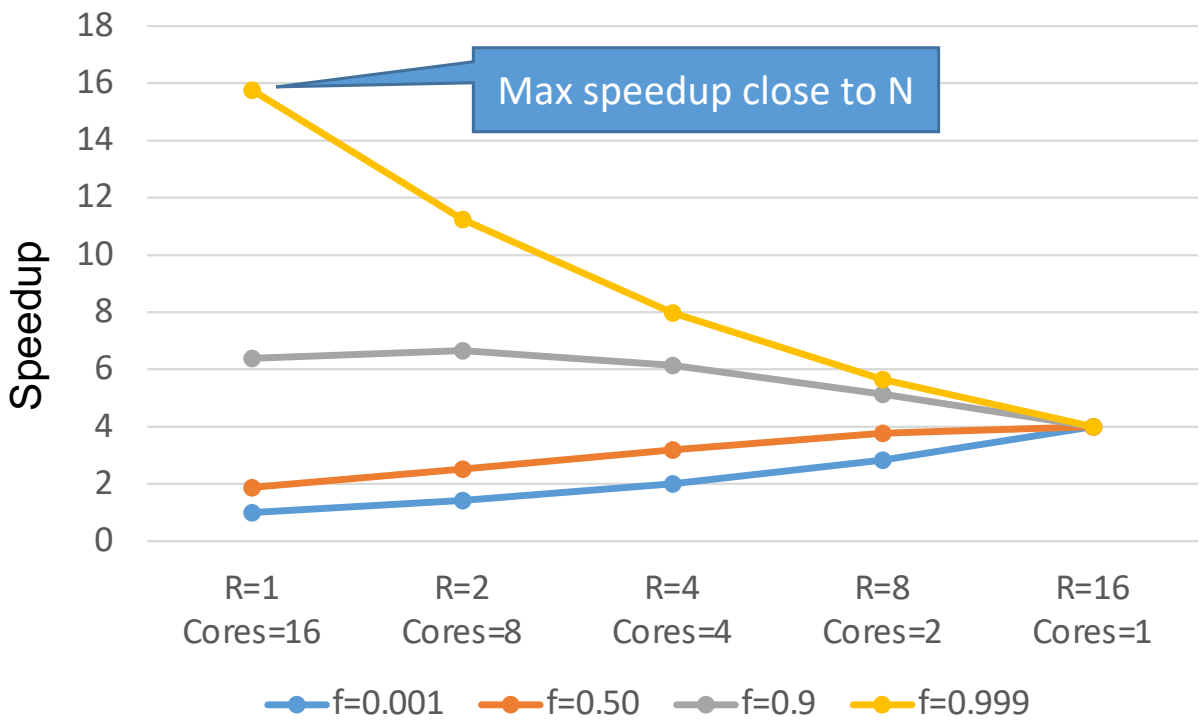
- Serial fraction of the program would now have a single core (sequential) performance on these combinations as:
  - $(1-f) / Perf_R$
    - $Perf_R$ is the performance of each of the single core of the processor type R shown below
- Parallel fraction use N/R cores at the rate $Perf_R$ each
  - $f / (Perf_R * (N/R)) = f*R / Perf_R *N$
- $Speedup(f, R, N) = 1 / (\{(1-f)/Perf_R\} + \{f*R/(Perf_R*N)\})$



Hill and Marty 2008

Amdahl's Law Revisited (Symmetric Cores)

- Symmetric multicore processor with total resources **N=16**
  - Speedup(f, R, 16) = 1 / ({(1-f)/$Perf_R$} + {f*R/($Perf_R$* 16)})

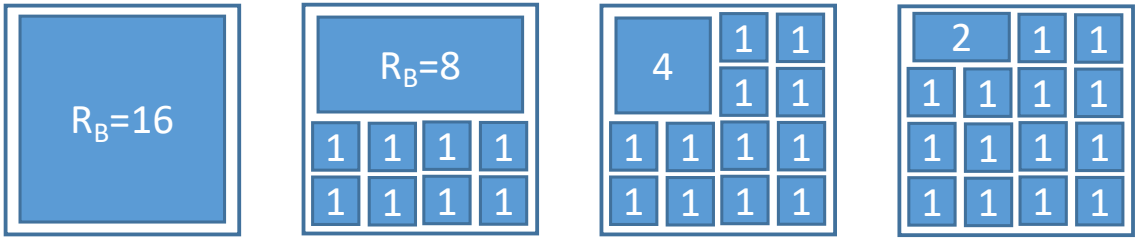


- At f=0.5, single core speedup better than 16 cores
  - Code should have parallelism for taking the benefit of multicores
- f=0.999 achieves far better speedup than f=0.9 using same 16 cores
  - "f" matters – Need to have as much parallelism as possible
- f=0.001 achieves 4x speedup in a single core processor v/s 1x speedup in a 16-core processor
  - Only a high performance core can improve the sequential performance
- How to design a processor that suits both sequential and parallel workloads?
  - Asymmetric cores!

Hill and Marty 2008

Asymmetric Cores

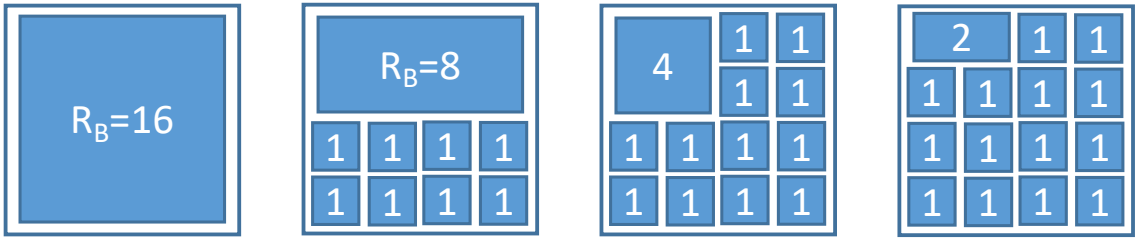
- Examples of different possible asymmetric multicore processors
  - **Symmetric** processor requires all cores to be equal, but **asymmetric** processor could have a mix of big and little cores
  - Each processor is using the same number of total resources (**N=16**)
    - One **B**ig core with R_B resources would leave $N-R_B$ resources for little cores
    - Assuming each little core has R=1, total number of little cores are $N-R_B$



Hill and Marty 2008

Amdahl's Law Revisited (Asymmetric Cores)

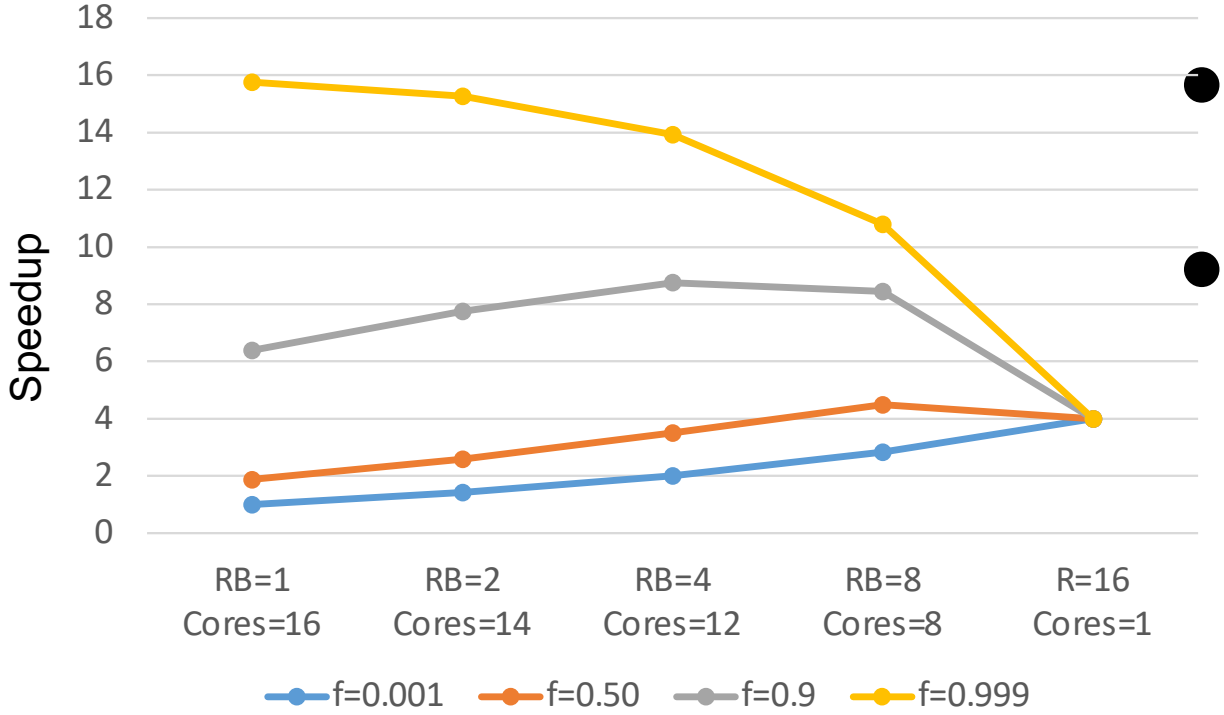
- Serial fraction would still be represented as in symmetric
  - $(1-f) / Perf(R_B)$
- Parallel fraction using one big core with $Perf(R_B)$ performance and $N-R_B$ little cores with $Perf(R_L)$ performance
  - $f / [Perf(R_B) + (N - R_B) \times Perf(R_L)]$
- When $R_L=1$, $Perf(1)=1$.
  - $f / (Perf(R_B) + N - R_B)$
- $Speedup(f, R_B, N) = 1 / (\{(1-f) / Perf(R_B)\} + \{f/(Perf(R_B) + N - R_B)\})$



Hill and Marty 2008

Amdahl's Law Revisited (Asymmetric Cores)

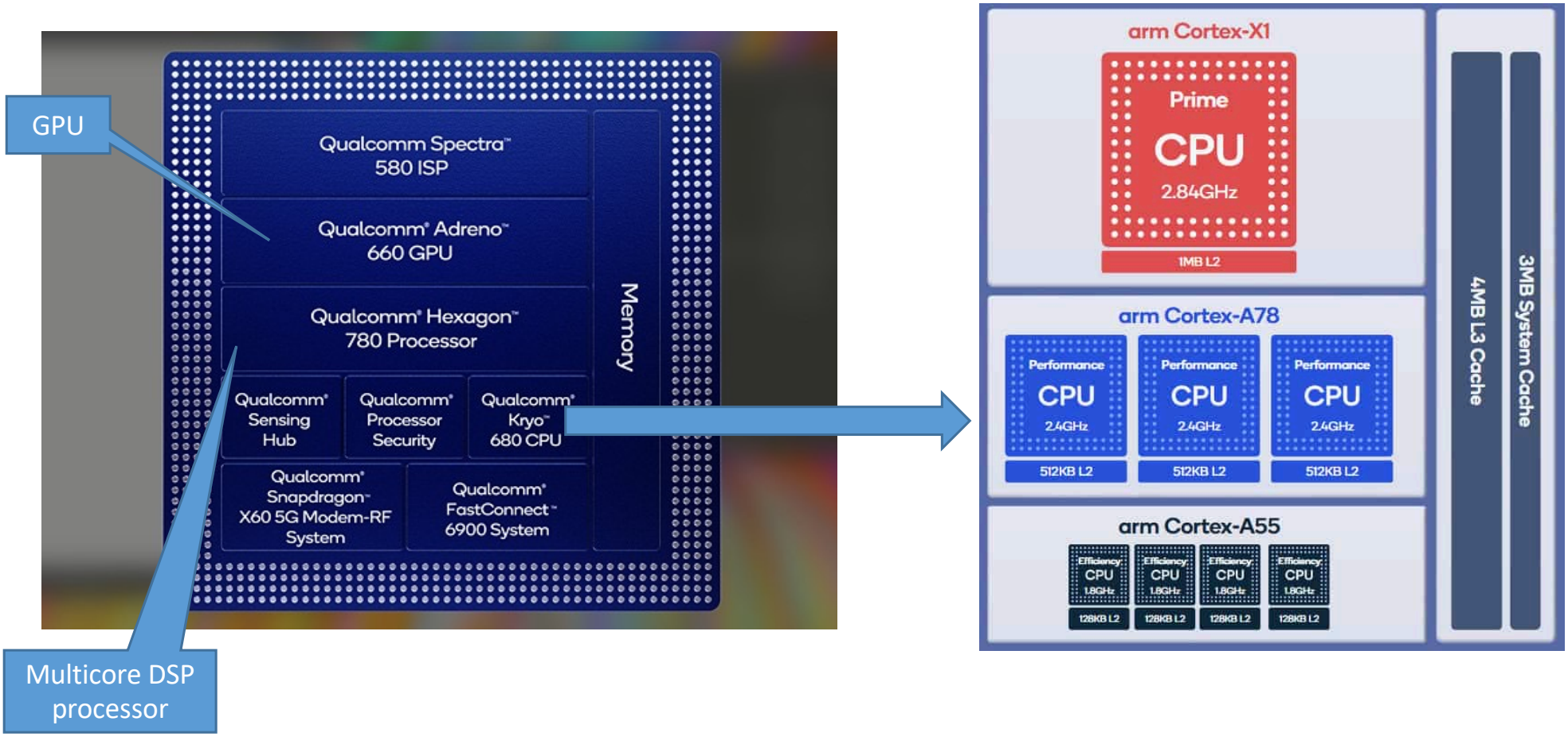
- Asymmetric multicore processor with total resources **N=16** (Perf(R_B) modeled as square root of R_B)
  - Speedup(f, R_B, 16) = 1 / ({(1-f) / Perf(R_B)} + {f/(Perf(R_B) + 16 –R_B)})



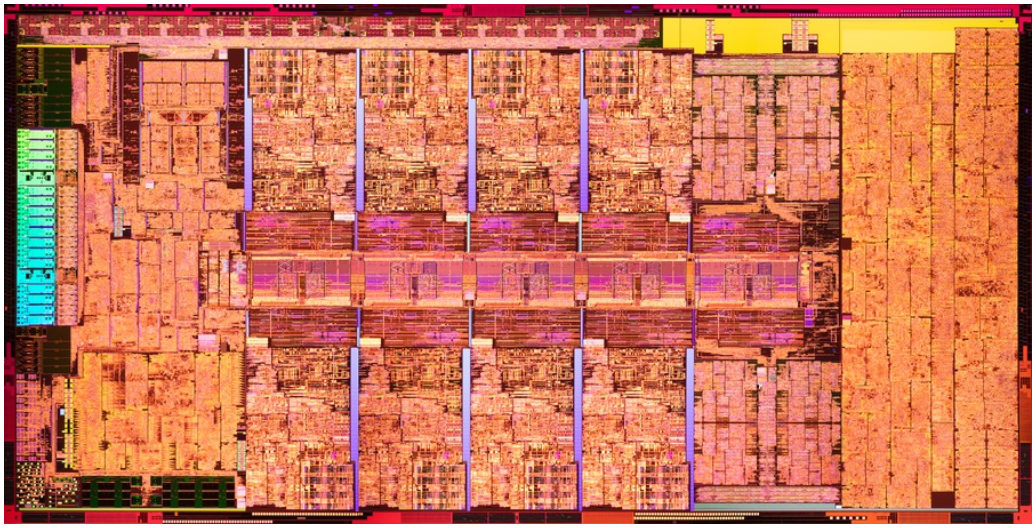
- Most real world applications are a mix of sequential and parallel code
- Providing a heterogeneous processor with different types of cores can help in improving the performance of both the sequential and parallel portion
  - Sequential part runs on the big core
  - Parallel part runs on a bunch of little cores

Hill and Marty 2008

Qualcomm Snapdragon Mobile SoC



Intel Heterogeneous SoC Architecture



- Alder Lake S (2021)
  - 8 **P**erformance and 8 **E**fficient cores
    - P cores max frequency 5Gz
    - E cores max frequency 3.8GHz
    - Up to 24 threads supported
  - Integrated GPU with 32 Execution Units
  - Shared memory across CPU and GPU

Picture source: https://en.wikichip.org/wiki/intel/microarchitectures/alder_lake

Software Issues with Asymmetric Multicore

- When to use the big core v/s little cores?
  - Or, how to use them simultaneously if such application arrives
- Managing the locality
- Achieving energy efficiency execution

Today's Class

- Amdahl's law revisited
- Integrated CPU-GPU architectures
- Runtime solution for hybrid CPU-GPU parallelism

Programming Integrated CPU-GPU SoC

1. **Setup inputs on the host CPU**
2. **Allocate memory on the host CPU**
3. ~~Allocate memory on the GPU~~
4. ~~Copy inputs from the host to GPU~~
5. **Start GPU kernel**
6. ~~Copy output from the GPU to host~~

- Intel integrated CPU-GPU architecture
  - Host and the device share the same physical DRAM unlike the discrete GPUs
    - Enables using the same copy of memory between the CPU and GPU
      - Avoids explicit memory transfers for GPU kernel execution
- Supported by OpenCL 2.0 Shared Virtual Memory (SVM) feature

OpenCL 2.0 Shared Virtual Memory (SVM)

- Allows sharing pointers across host and device
  - Coherency in the shared data modified across host and device on integrated CPU-GPU architecture
- Supports memory consistency models
  - Allows usage of atomics across host and device like two distinct CPU cores
- Supported by Boost.compute using simple APIs

Source: https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html

Using Boost.Compute SVM on Intel SoC

- Demo of programs available in course GitHub repository
  - GPU-only vector addition
    - https://github.com/hipec/cse513/blob/main/lec21/vecadd.cpp
  - Hybrid CPU-GPU matrix multiplication
    - https://github.com/hipec/cse513/blob/main/lec21/matmul.cpp

GPU-only vector addition

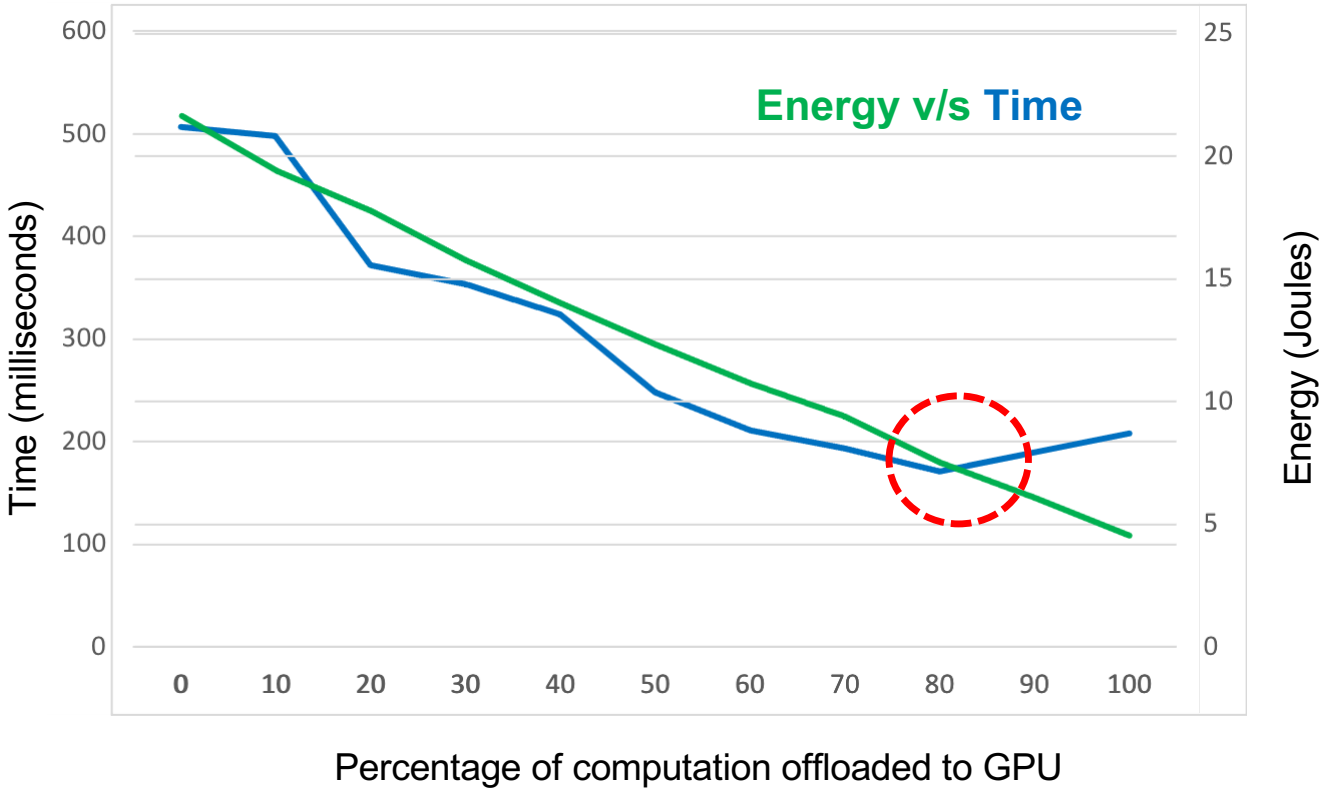
```
const int vector_width = argc>2?atoi(argv[2]):8;
double time_gpu=0;
// lookup default compute device
auto gpu = compute::system::default_device();
// create opencl context for the device
auto context = compute::context(gpu);
// create command queue for the device
compute::command_queue queue(context, gpu);
// print device name
std::cout << "device = " << gpu.name() << std::endl;
int* a = my_svm::alloc<int>(context, size);
int* b = my_svm::alloc<int>(context, size);
int* c = my_svm::alloc<int>(context, size);
std::fill(a, a+size, 1);
std::fill(b, b+size, 2);
std::fill(c, c+size, 0);
for(int i=0; i<size; i++) assert(a[i] == 1);
for(int i=0; i<size; i++) assert(b[i] == 2);
```

```
// source code for the add kernel
const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
    __kernel void add(__global const int *a,
                      __global const int *b,
                      __global int *c)
    {
        const uint i = get_global_id(0);
        c[i] = a[i] + b[i];
    }
);
// create the program with the source
compute::program program = compute::program::build_with_source(source, context, "-cl-std=CL2.0");
// create the kernel
compute::kernel kernel(program, "add");
// set the kernel arguments
kernel.set_arg_svm_ptr(0, a);
kernel.set_arg_svm_ptr(1, b);
kernel.set_arg_svm_ptr(2, c);
// run the add kernel
timer::kernel("GPU kernel", [&]() {
  queue.enqueue_1d_range_kernel(kernel, 0, size, vector_width);
  queue.finish();
});
time_gpu+=timer::duration();
std::cout<<"Total GPU time = "<<1000*time_gpu<<"ms"<<std::endl;
//verify the computation
for(int i=0; i<size; i++) assert(c[i] == 3);
std::cout<<"Test Passed at GPU\n";
timer::kernel("CPU kernel", [=]() {
  for(int i=0; i<size; i++) c[i] = a[i] + b[i];
});
std::cout<<"Speedup of GPU over CPU= "<<timer::duration()/time_gpu<<std::endl;
//cleanup
my_svm::free(context, a);
my_svm::free(context, b);
my_svm::free(context, c);
```

Today's Class

- Amdahl's law revisited
- Integrated CPU-GPU architectures
- Runtime solution for hybrid CPU-GPU parallelism

Hybrid CPU-GPU Matrix Multiplication



- Experiment using the hybrid CPU-GPU matrix multiplication inside the course GitHub repo (size=1024x1024)
- Intel i7-6700 processor
  - 4-core CPU @ 3.4GHz
  - 24-EU GPU @ 350MHz
- Optimal execution at 80% offload
  - Would vary depending on application, processor, and computation size

Software Challenges with Integrated CPU-GPU

- **Scheduling** – when to use the big core (or CPU) v/s little cores (or GPU)?
  - Heavily depends on the type of application
    - Offloading 80% computation on GPU was optimal in case of matrix multiplication, but it might be suboptimal in case of vector addition
      - Why?
    - Offloading percentage would be different for different kind of workloads (IO-bound, memory-bound, CPU-bound, etc.)
  - Depends on optimization criteria (time, or energy, or best of both, etc.)
- **Programming** – manually partitioning the workload would hurt programmer's productivity
- **Solution** – runtime assisted scheduling!

Energy-Aware Scheduler for Integrated CPU-GPU[1]

- How to do an online profiling in a running application?
  - Classify the work-load by running a few iterations of the for-loop on both CPU & GPU simultaneously
  - Compare the findings of the online profiling either with a pre-trained processor specific model (on-the-fly is also possible, and is discussed in several papers)
    - A one-time characterization of the processor's power usage by measuring the power used by different kinds of workloads by varying the value of **alpha** (**α** → percentage of tasks offloaded to GPU)
      - Measured once on each experimental machine

Energy-Aware Scheduler for Integrated CPU-GPU[1]

- Steps for energy aware runtime scheduling of parallel_for
  1. If α exists for this computation
     - Schedule iterations over CPU-GPU using this known α value
  2. If α does not exists for this computation then perform repetitive profiling to determine the minimum value of the target object function (objective could be energy, EDP, etc.)
     a) Divide fixed sized N_{Chunks} between CPU-GPU by changing α in the range 0…100% (where, $N_{Total}\%N_{Chunks}=0$)
        - If N_{total} is too small then exit this loop and launch entire computation on CPU only
     b) Profile the CPU and GPU executions (CPU & GPU throughputs, memory bandwidth, etc.)
     c) Characterize work-load and determine corresponding power curve
     d) Increment α in step of 0.1 and repeat the above steps until half of the N_{chunks} remaining

How to choose the value of N_{chunks}?

Recall, Intel GPUs have EUs akin to a CPU-core, and are also N-way SMT with wide vector units

[1] https://dl.acm.org/doi/abs/10.1145/2854038.2854052

Reading Materials

- Amdahl's law in the multicore era
  - https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/34400.pdf
- OpenCL 2.0 shared virtual memory overview
  - https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html
- A black-box approach to energy-aware scheduling on integrated CPU-GPU Systems
  - https://dl.acm.org/doi/abs/10.1145/2854038.2854052

Next Lecture

- Resilience in the exascale era
- **Quiz-4**
  - **Syllabus: Lectures 18-21**