CS 133 Notes

Lecture Notes

1/7 - Week 1

- World is getting to be parallel, distributed, and heterogeneous.
- In a CPU, you'll almost never find a single core.
- 20 years ago, in computers, you used to have a single processor on a chip that only supported traditional sequential programming.
- Course will teach how to program multi-core CPUs.
- Resources
    - The CS 133 server has 16 cores and 32 threads.
    - The AWS server has 4 cores and 8 threads.
    - GPU will have 2048 cores.
    - FPGA's can have 2.5 million logic cells.
- In this class, the computer used in this class is lnxsrv08 and it is an 8 core CPU. It has two processors. The AWS computer is an 18 core CPU, and in the instance we have, we have access to 4 of the cores. The AWS GPU is a Tesla M60.
- Transistor size shrinks by 70% every generation and this is equivalent to saying that the number of transistors doubles every generation.
- FPGA allows you to program your own logic which is the differentiating component between these and CPUs and GPUs. You can implement any k input functions by using a lookup table that has 2^k functions.
    - FPGA chips are made of programmable logic blocks, programmable interconnects, an programmable I/Os.
- Cloud computing is making the use of supercomputers and compute power a utility that you can pay for on demand rather than you having to build your own computers.
    - You can also pay for heterogeneous computing, so you have options if you have different needs in terms of compute, storage, memory, etc.
- Moore's Law says that every 18 months the number of transistors on a chip doubles and because of the increased number of transistors, you get better single processor performance. Those extra transistors go towards more efficient memory organization, deeper pipelines, and better instruction schedule support to exploit ILP. The same sequential code for that program you wrote would get faster as the number of transistors increased.
    - The reason you can't have a single processor is that Moore's Law will stop eventually and you'll generate too much heat if you try to increase the clock speed for that processor in an attempt to increase the speed.
- Dennard scaling says that you want a constant electric field. It shows how as transistors get smaller, you get benefits in other areas besides just being able to put more transistors onto a chip.

    - If you shrink the transistor length by a factor of s, the area and power decrease by s^2 and the delay decreases by s as well and the energy decreases by s^3.
        - For example, if the size of the transistors gets 30% smaller, then area and power decrease by 50% and energy improves by 66%.
    - Stopped because we can't scale voltage exponentially without a limit. If voltage and frequency stays the same, then in P = c * v^2 * f, capacitance will still go down since area is getting smaller, and thus the only way to increase speed is to add more cores
        - Leakage power caused a lot of issues.
    - The stoppage of Dennard scaling meant that the frequency has reached an upper limit of about 4 Ghz.
- Hw Question: What is Dennard Scaling and why did it break down.
    - Because Moore's Law stated that every 18 months, the number of transistors on a chip will double. In effect, this means that the individual transistors on each chip are getting smaller (about 30% every generation). Dennard Scaling says that as transistors get smaller, in order to keep the electric field constant, voltage and current need to be scaled down in relation to the size of the transistor. Since voltage decreased, the power needed by the transistors also decreased, allowing manufacturers to raise clock speed without affecting the overall power usage. Dennard Scaling ended up breaking down because current leakage as a result of supply voltage and threshold voltage being too small, and thus we have to eventually stop making transistors smaller and we have to stop scaling voltage down. And so if we aren't decreasing voltage, any attempt to increase frequency will cause a rise in the power usage. The frequency scaling that used to give performance now cause power and heat generation to increase rapidly, thus giving us an approximate threshold for 4 GHz for clock speed.
- The big problem with trying to increase clock speed (frequency) is that there is too much power required.
    - Instead of frequency scaling and trying to extract gains from ILP and speculation with really complex single core architectures, the solution is to use the additional transistors to create more processors and simpler processors to the chip.
- Parallelism can help because you're keeping voltage and frequency constant and capacitance is scaling down as you're decreasing the size of each processor. This reduces the power (P = c * v^2 * f) that is consumed by each core, and allows you to increase the number of cores used so that you keep a constant power budget.
    - Specifically, to keep a constant power budget we can increase the number of cores by 1/(s - s^2).
- We can see today's Moore's law are more of the number of cores per chip doubling every two years, clock speed not increasing, and dealing with concurrent threads, inter-chip parallelism, and core heterogeneity.
- All the supercomputers have roughly the same frequency, but they differ more in the number of cores.
- Challenges to creating programs in parallel

    - Think parallel
    - Finding parallelism (Amdahl's Law - Basically, you have to find out which parts of your programs even have the ability to be parallelized)
    - Workload balancing - Dividing work evenly
    - Synchronization
    - Performance measures
- Amdahl's Law says that there if there is enough parallelism to make use of the multi-core machines, the speedup should be T1 / Tn (T1 = 1-thread execution time, Tn = n-thread execution time)
    - You can normalize the time to 1 for single core execution. Speedup thus equals 1 / (1 - p) + (p/n)
        - The 1-p refers to the sequential part that can't be sped up.
        - P is the parallel fraction and n is the number of threads.
    - If you have 60% concurrent execution and 40% sequential and you have 10 cores, then you get speedup of 1 / (.4) + (.6 / 10) = 2x speedup with 10 cores instead of the expected 10x.
    - If you have 80% concurrent execution and 20% sequential and you have 10 cores, then you get speedup of 1 / (.2) + (.8 / 10) = 3.75x speedup with 10 cores instead of the expected 10x.
    - Basically, the main point with this law is that if only part of the application can be parallelized (which is the case 99% of the time), then focus on parallelizing the part of the program that has the highest execution time.
- The biggest barrier to parallelism is the overhead that gets created when you try to parallelize. Aka these are the reasons that you don't get the expected speedups when you're dealing with spreading the work across multiple cores.
    - Cost of starting a thread or a process
    - Cost of communication of shared data
    - Cost of synchronizing
    - Cost of extra and redundant computation.
- Load imbalance is also another problem where some processors are idle while others are backed up with work. This could be caused by unequal size tasks and insufficient parallelism.
- In this class, we will use
    - OpenMP which works with shared memory processors.
    - MPI: Allows you to program supercomputers
    - OpenCL for heterogeneous computing
    - CUDA is for Nvidia GPUs

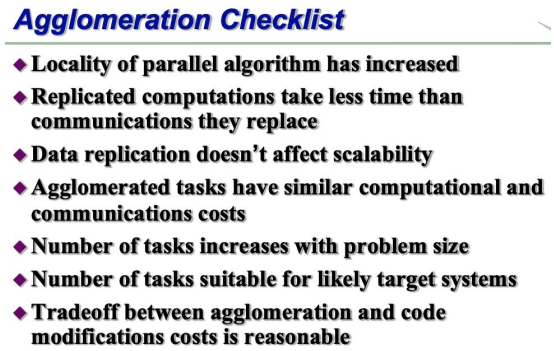
1/9 - Week 1

- Main reasoning we went to parallel computing was that we couldn't scale the frequency above 3 or 4 GHz. We were able to put more transistors on a chip though.
- Parallelizing an existing program

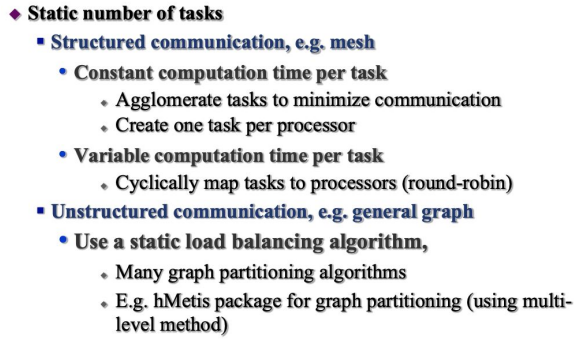
    - First try to see which is the longest part of the program when you're looking for what part to try to parallelize. This is because you'll get the biggest drop in execution time if you successfully parallelize that. Then, try to parallelize the next longest part of the program. This is an incremental parallelization.
    - Fork join model refers to when you are running a program, you start sequentially, then fork out a lot of threads to get parallelism for a certain part of the program, and then you join the threads so that you get back to sequential.
- Designing a new parallel program from scratch (The below 4 components are Foster's Design methodology.)
    - Partitioning - Dividing the computation and/or data into pieces.
        - Domain/Data Decomposition (Exploits data parallelism) - Splitting up the work based on the data. Gotta decide how the elements will be partitioned among the processors, have each of the processors do certain tasks, and then aggregate the results.
            - Ex) If you have a huge array and you have to compute the max of the array. You can just do it sequentially and keep a temp variable for max and update if needed. However, if we want to make it parallel, then we split up the array, have each core handle the local max, and then find the max of the local maxes.
                - The Map step is the splitting of the array and the Reduce step is finding the max from the local max.
        - Function/Task Decomposition (Exploits task parallelism) - Splitting up the work based on the jobs or functions or computation that we have to do. Gotta decide how the tasks will be partitioned among the processors, figure out what data/information those processors will need, and also figure out the order they processors need to work in, and finally the communication between all of them.
            - If we have a task graph that shows all the tasks that need to be done and the relationship between the tasks that need to go before others (through directed edges).
            - The maximum degree of concurrency depends on finding the maximal independent set which is NP hard.
                - An independent set is a set of vertices in a graph, no two of which are adjacent. A maximal independent set (MIS) is an independent set that is not a subset of any other independent set. In other words, there is no vertex outside the independent set that may join it because it is maximal with respect to the independent set property.
            - This graph is a partially ordered set (A poset consists of a set together with a binary relation indicating that, for certain pairs of elements in the set, one of the elements precedes the other in the ordering.)

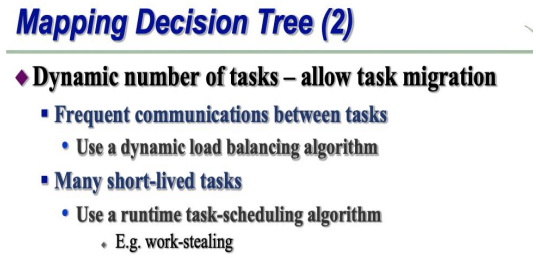
        - In these sets, the list of numbers is ordered but not really consecutive because there are some numbers missing?
        - Size of a maximum anti-chain = size of a minimum chain partition
    - Pipelining (Exploits pipeline parallelism) - This is a special kind of task decomposition. Make use of different parts of a program using different parts of the processor and finding computation that can be overlapped in time.
        - Let's say you're doing some graphics stuff and you have a series of steps, like model, project, clip, and restorization. Here there is a clear dependency between the steps so it seems like you cannot really parallelize it. Solution is to pipeline it where you can start different images on different stages. It works because the computation for the images are independent of one another.
        - Without pipelining, the latency is 4. And even with pipelining, the latency is still 4. However, the throughput increases. Without the pipelining, the throughput is ¼. With it, the throughput is approximately 1. Specifically, the throughput is N/N+3. The 3 comes from the number of steps to fill up the pipeline.
            - The speedup is 4N / N+3
        - The initiation interval is the number of time units it takes for you to add a new data point in (in other words, the number of cycles between the start of two loop iterations). This is not always possible depending on the availability of the resource and the possible data dependencies (loading before another component is supposed to store a new value).
            - If you have an II of 1, then the throughput is maximized.
        - For (i = 0; i < SIZE; i++){r += V[i]}
            - You're able to pipeline it because each loop will require a load and then an add. After the first two operations, the load of the second can come during the add of the first.
        - For (i = 0; i < SIZE; i++){r += V[i]; r *= V2[i]}
            - Here we have 4 ops, the 2 loads, and then the add and then the multiply. After the first data point, you have to wait an additional time step since we only have one ALU and we can't do a multiply and an add at the same time.
        - For (i = 0; i < SIZE; i++){d[i] = d[i-1] * v[i]}
            - The full operations involve 2 loads, 1 mult, and then a store.
- Communication/Optimization
    - Local communication - The current task needs information from a small number of other tasks

        - Global communication - Significant number of tasks needing to contribute data for one task to be able to do its job.
    - Agglomeration - Want to minimize the amount of communication that needs to take place. We need to do some smart grouping of tasks into larger tasks and we want to improve locality. Basically, if you combine two tasks into a larger one, then you remove the overhead associated with the communication between those two tasks.



    - Mapping - Physical Configuration and process of assigning tasks to processors. This can be done by the OS (centralized multiprocessor) or by the user (distributed memory system). The goal is to maximize the processor utilization and the minimize the amount of communication that needs to be done between the different processors.
        - Finding the optimal mapping is NP hard, and thus we have to rely on heuristics.
        - Finding the mapping is hard because you have to take into account task dependence.





- Graph partitioning is where you minimize the number of edges cut such that each are of roughly equal size.

1/14 - Week 2

- OpenMP = Open multiprocessor
    - Designed for shared memory machines.

    - Some of the features are that you have compiler directives, runtime functions, and environmental variables.
- OpenMP is incremental because you can take any program and pick and choose the places where you can try to include some parallelism.
- Pros of OpenMP: Incremental, easy to get started, and good for data partitioning
- Weaknesses of OpenMP: Not great for functional partitioning, be sensitive about race conditions.

1/16 - Week 2

- OpenMP has pragmas, runtime functions, environment variables
    - Important pragmas are parallel, parallel for, parallel section, parallel single, reduction, etc
    - Runtime functions are get_thread_num, get_nthread_num,
- Today, talk about performance optimization in terms of loops. This will deal with scheduling, loop transformations, and rule of thumb.
- The metrics for the performance optimization are
    - Speedup: time for sequential processor / time for parallel processor
        - In order words, T_1 / T_N
    - Efficiency: Speedup / Number of cores
        - Will be less than 1 most of the time since we can't utilize all the cores all the time most likely.
- Prediction
    - Something = 1 / (S + ((1 - S) / P))
- Example
    - Painting a house with 300 pickets. Every picket takes a minute for 1 person. You also have to add 30 minutes after for preparation and 30 more minutes for cleanup. These two have to be done sequentially.
    - If you have 1 person working, then it will take 360 minutes, speedup is 1x, efficiency = 100%
    - If you have 2 people working, then it will take 60 + 300/2 = 210 minutes, speedup is 1.7x, efficiency = 85%
    - If you have 10 people working, then it will take 60 + 300/10 = 90 minutes, speedup is 4x, efficiency = 40%
    - If you have 100 people working, then it will take 60 + 300/100 = 63 minutes, speedup is 5.7x, efficiency = 5.7%
- As you increase the number of processors or people, the speedup increases, but the efficiency decreases. However, the speedup reaches a bit of a plateau.
- General formula of Amdahl's Law
    - S + P / (S + (P/p) + O)
    - S = sequential portion, P = parallelizable portion, p = processors, O = overhead from creating threads, load balancing, synchronization, etc.

- The hope when you do parallelization is that as you increase the number of processors, your speedup will be linear.
    - As the number of computations increase, more processors will result in you staying closer to the linear speedup line.
- Speedup does not equal speed.
- One rule of thumb is to start with the best sequential algorithm.
- Another is that you want to make the most out of locality especially when it comes to accessing values in an array and spreading array computation across the processors in an optimal way.
- OpenMP scheduling gives you a chance to partition the iterations of the for loop. Each thread will have a set number of iterations to execute.
- To add scheduling, as an addition to the pragma omp parallel for, you can add schedule (static, 2) where you doing the ordering explicitly.
    - The 2 refers to the size of the data that you want the processor to take on. I think the default would be (the number of iterations / number of processors).
    - If you replace static with dynamic, it becomes more first come first serve for the threads in terms of them being assigned work.
    - If you use guided, the first processor will get (iterations / processors) chunks, and then the next processor will get (the remaining number of iterations / number of processors).
        - This variant is constantly adjusting to the demand and to the supply of processors.
- Dependency Stuff - If R and then S
    - Def(S) = f(use(S))
        - Def refers to the variables to compute and use refers to variables to use.
    - use(S) intersect def(R) is flow dependency RAW
        - Most important since it refers to situation where a variable gets edited and then it gets read in a subsequent line.
    - use(R) intersect def(S) is anti dependency or WAR
    - def(R) intersect def(S) is output dependency or WAW
    - These are the Bernstein conditions.
        - http://meseec.ce.rit.edu/eecc756-spring2006/756-3-21-2006.pdf
- When you are parallelizing for loops, you can do
    - Loop permutation, which is where you switch up the order of loops to fix dependency problems or to take advantage of locality.
    - Loop distribution which is where you split up statements in the same for loop into 2 or more different for loops where those statements are split up. The tough part is to determine if those statements have dependencies.
    - Loop fusion is basically the opposite of ^. You're basically combining loops.
    - Loop shifting is where you change the bounds of a for loop. Helpful when you want to fuse two for loops together and you want to make sure that the bounds for both of the loops are the same.

1/23 - Week 3

- There are a lot of correctness issues in parallel programming. There are a bunch of worker threads doing work on a bunch of shared data. Nede to make sure they can work together.
- Synchronization is the process of managing shared resources so that reads and writes occur in the correct order regardless of how those threads are scheduled (they can be scheduled differently on every run).
- Barrier is a point where every member in a team of threads must arrive before any member can proceed onward.
    - This is a point of synchronization and is important for any data dependencies. Also helpful for if you want a certain part of your program to be the *last* thing to be run. Aka, you want the threads to wait when it gets to a certain point, and then proceed to the last statement(s).
    - You can add #pragma omp barrier
        - However, this is automatically added at the end of worksharing constructs like pragma omp for and pragma omp single.
    - Can be disabled with the nowait clause.
    - Make sure that all the threads in the thread pool can get to the barrier, otherwise you'll be waiting for a long time.
- Mutual exclusion is one kind of synchronization which allows only a single thread to have access to a shared resource.
    - Critical section is where only one thread at a time will execute the structured block within a critical section
- Race condition in non deterministic behaviour on a particular run caused by the times at which two or more threads access a shared variable.
- Avoiding race conditions can involve scoping variables to be private to threads and control access to shared variables with mutual exclusion.
- Locking mechanisms are commonly used where you have an atomic test-and-set operation and that helps you control access to shared resources.
- Pragma omp critical is a portion of the code that only thread at a time may execute it.
    - It removes the parallelism by forcing the threads to act sequentially in that section.
    - Helps avoid race conditions, but also slows things down.
- Important to lock data not code. Locking code means you want threads to go and not go into particular parts of code. Instead, lock the pieces of data so that threads don't access pieces of data (not code) at the same time.
- Atomic construct in openmp is a special case of a critical section. Atomic applies only to simple operations.
    - Difference between atomic and critical is that only the += and -= operations in an atomic section will be atomic, and you keep all the other computation being parallel.

1/28 - Week 4

- The main technique for synchronization is barriers. Basically things to make sure that all the threads come to the same point and wait before continuing.
- In order to get rid of race conditions, we use locks, critical sections, memory fence, and atomic operations. The key is that we want to do a test and set in one atomic operation.
- Loop transformation involves changing the order of nested loops.
    - Polyhedral model gives you a way to look at how to do loop transformations in a way that does not violate dependencies given some execution order (which you can represent with a polytope).
        - We want to maximize concurrency/parallelism and locality.
    - There are computer programs to extract a polytope
- Design patterns for writing parallel programs. Can organize the program based on the tasks, the data, or the flow.
    - Tasks can be done linearly or recursively. We can first create a graph where the nodes are the tasks and the edges are the dependencies (can be control or data dependencies). This allows us to see if there are any constraints we need to keep in mind when we are creating the schedule of our parallel program.
        - Linear tasks can be trivially parallel or they can have mild dependencies (such as in mapreduce tasks).
        - Tasks can also be done recursively. The most common algo that uses this is merge sort. It's basically a divide and conquer where you divide the tasks until you get a trivial task, do the computation, and then combine all the results. The bottleneck with these approaches is that at the end you have to combine two sorted lists. The complexity would be O(n + m). Can we get it to
            - Position (a_x) = rank(a_x, A) + rank(a_x, B). You already know the position of a_x in A and because B is sorted, you can do binary search with that part and get O(log n).
    - Data can also be handled linearly or recursively.
- In order to find the optimal parallel algorithm P and we know the T*(n) is the optimal sequential algorithm, then Work Total (P) = O(T*(n)) and T(P) is the best possible.
    - Work total refers to the computation done by all the processors in total. This can to be on the same order as the sequential algo.
        - I think the key is that you're not just doing a shit ton of work that becomes fast through parallelism. The overall efficiency has to be good.
    - T(P) being the best possible means that it is optimal in the strong sense which means that no other algorithm can beat it.
    - Something with PRAM model
- Divide the B array into log m pieces. Then the length of each piece is m / log m. But he said that is equivalent to the number of chunks. That is multiplied by log n which is the cost of the binary search.
    - (m / log m) * (log n) is the big O of splitting it up.

- Also saying that each of the merges is sequential.

1/30 - Week 4

- Let's look at an example that deals with the merging of lists in parallel.
  - Ex) A = [4,6,7,10,12,15,18,20] and B = [3,9,16,21]. Let's say we want to merge the lists. First step is to divide one of the lists into chunks, let's say B, and the number of chunks is log (size of list). You'll then have B_0 and B_1.Then pick the last item in the chunk and do binary search to find where it fits in list A. Once you have that index then split A into A_0 and A_1. Then, you can give the _0's to one processor, the _1's to another, etc. Basically the chunks are each given to different processors. Then, once all the sections of the list are merged independently by each processor and then combine all the pieces basically.
- Organizing a program based on data can be geometric or recursive.
  - Recursive examples are pointer jumping and balanced tree construction.
- The parallel prefix algorithm is not optimal because the amount of work required for the parallel algo is not on the same order as the work done in the linear algo. The caveat is that the parallel algo is ultimately faster because you have a bunch of processors working on it at the same time, even though there is more total work now.
- Prefix sum algorithm is the one that creates an array of running sums from another array.

input numbers	1	2	3	4	5	6	...
prefix sums	1	3	6	10	15	21	...

  - In order to parallelize the sequential implementation, we need to create a balanced tree construction.
  - We can start by just try to pair up numbers in the input array and get those sums and we can do that in parallel since the pair computation is independent. Then you have an array y of length n/2 of pair sums. Then, you create a list z which is a prefix sum for the y's through recursion. Basically we saw how we want from x to y by computing pair sums and so we're going to get the list z through recursion and when the sums come up, we use that list to get the final sums for the original x. You can construct the prefix sum as [x0, z0, z0+x2, z1, z1+x4, etc]
  - Complexity T(n) = T(n/2) + a = T(n/4) + 2a = T(n/8) + 3a = a * log (n)
  - Total Workload = W(n/2) + b*n
    - The extra n factor is to account for the fact that all n of the processors did something.
  - Algorithm is optimal because the work done is on the same order as the sequential one and the time taken is faster. Don't know if optimal in the strong sense because no proof that lower bound is equal to that, or aka if there is an algorithm that can be better.

- We can also look at parallel programs in terms of flow. This often refers to pipelining. There is a concept of a compute time and communication time. The compute times are associated with the nodes, and the communication time represents the edges.
    - Assuming you have a 10 millisecond stage period (which is given as a constraint), you need to make sure none of the stages exceed that time. The stage time is a function of the communication and compute times.
    - Communication delay can disappear if it's all at the same processor. The communication delay between two stages goes on the receiving one.
    - The label of a node is the minimum number of stages up until that node. Basically a label for a node will be some number x if all the predecessors that have x added up are lower than the stage period. If not, then the stage is x+1.
- Task pipelining is example of coarse grain parallelism.

2/4 - Week 5

- OpenMP allows for the threads to see and modify a shared address space, but they also have the ability to have their own private address space.
- Different approach: If you don't have that shared address space (the threads have separate address spaces), you have to use a message passing paradigm where you use send/receive to communicate between threads. This model is loosely coupled. This approach is probably used when you have hundreds or thousands of machines, you can't use a shared address space because it will never be big enough and therefore, private address spaces should be used. It has a SPMD structure - single program multiple data
- When considering the send and receive commands, you have to pay attention to blocking and buffering.
- When a thread T wants to send info, then it will send a request R to send, and then the other threads will send an OK message saying it is okay to send, and then T will send the information.
    - However, if any of the other threads aren't ready to received, then it won't send the OK and thus thread T will be idle until it gets the OK. That idle time is classified as when the thread is blocked.
    - There is also a situation where T sends the request, gets the OK, but is not finished with the computation of the data. Therefore, the other thread is the one that is idle. They won't get the info until T computes and sends it.
- One of the solutions to the above idling problems is buffering where you put your send/receive messages into a buffer in order to be more efficient. You're putting the message into a buffer and then sending later. It's good because you're able to do work and write data even if you're not ready to send at the current moment.
- In a non-blocking situation with no buffer, it says that even if a thread gets an OK message (after a send request), the thread will keep doing computations and modify a data structure with that computation info.
    - If you do have a buffer, you can't overwrite the received data.

    - The biggest point with non-blocking is that you do computation without necessarily having to wait for a reply. This helps avoid deadlock and allows you to overlap computation.
- MPI refers to a Message Passing Interface
- Rsend/Rreceive allows you to do blocking while Isend/Ireceive allows you to do non-blocking.
- There are a lot of functions in MPI, the most important of which are init, finalize, communicator size (number of processes involved), communicator rank (process ID), send, and receive.
    - Send will take in a buffer, a count for the amount of data to send, a field for data type, the destination, a tag, and a communicator.
        - MPI-send(*buf, count, datatype, dest, tag, comm)
        - MPI-send(void, int, MPI-datatype, int, int, MPI-communicator)
    - Receive will take in a buffer, a count for the amount of data to receive, a field for data type of the expected messages, the source, a tag, and a communicator.
        - MPI-recv(*buf, count, datatype, source, tag, comm, status)
        - MPI-recv(void, int, MPI-datatype, int, int, MPI-communicator, MPI-status)
    - MPI-comm-rank is a function that returns you basically an ID for the process.
    - MPI-broadcast is a function that sends messages to others
    - MPI-reduce does a particular operation on all the data that the thread will receive.
    - MPI-barrier is a communicator itself that allows all the threads to synchronize.
    - MPI-scatter allows you to send different parts of a piece of data to different other threads.
        - The summation of recvcount should equal the sendcount.
    - MPI-gather is when the thread is expecting data from different sources, and the function will aggregate it.
        - The summation of sendcount should equal the recvcount.
- You can also divide the processes in different groups which can each different communication patterns.
    - You can have different communicators inside of the same group, and each one will have the same complex.

2/6 - Week 5

- Two models in parallel programming revolves around shared address space (OpenMP is an example) and a separate address space (MPI).
    - If we have the separate address space, then we communicate through send and receive.
- The Bsend function requires you to send a message with a required buffer.
- The Isend function is the blocking version TODO more info
    - Special hardware will allow you to continue with the computation but there are restrictions of what data you can pull out.

- 3 Different ways of doing matrix-vector multiplication where M = size of n x p and vector v = n x 1, and then the resulting vector c = p x 1.
    - 1 way is to split up the rows between the cores. This is row wise blocked striped. Each core will handle the computation of 1 c_i, and that is the dot product between the specific row of M and the vector v. Then, you'll need all-gather communication to gather all the c_i's to create the full vector c.
        - MPI-allgather takes in a send buffer, a send count, the type of thing being sent, a receive buffer, a receive count, a receive displacement (says where each element starts - array index is an example), the type of thing being received, and the communicator.
        - From a high level, the call to this function will take all the data created by each of the processors and combines them into one single data representation.
        - The complexity of the call is O(log p + n)
    - Another is to split up the columns between the rows. Each core will take a column and do an MPI-alltoall which does a transpose operation that turns the rows in M and turns them into columns, and then from there you can use MPI-allgather.
    - The last is to block the matrix M. If you have p processors then we can split M into a grid of size sqrt(p) x sqrt(p). We distribute the mini-blocks to the processors, do broadcast of the sections of vector b, do the computation, and then reduce the partial vectors of c along the rows.
- MPI-scatter and MPI-scatterv are other operations you can use to send/do computation on data to other processors.
- The efficiency of a program will depend on the data size. As the data increases, the curve will move to the right.
    - Traditionally, the curve slopes to the bottom right since as the number of processors increases the efficiency will go down.
- Scalability function tells you the memory @ f(p) / p.
    - Want the memory required to grow slowly.
    - Tells you if it is feasible in terms of memory to increase the number of processors.
- Total communication cost is c * p * n. This should be lower than n^2 which I'm assuming is the cost of a sequential program.
    - You can extend this to the multiplication program we're talking about earlier. In that checkerboard approach, we see that c * p * (n / sqrt(p)) * log (p) <= n^2. The left side shows you the communication cost and the right side shows the amount of computation that will be done in a sequential implementation. After simplifying, we see that n >= c * sqrt(p) * log (p). And then if you divide by p to get the scalability function, then SF = c^2 * log^2(p)
- With MPI, it is very important to minimize the amount of communication you're using. We have to undergo the pain of sending data to and from different processes.

2/11 - Week 6

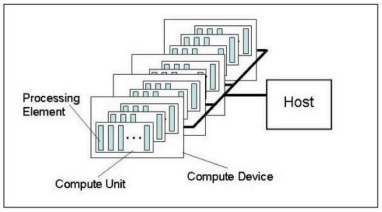
- Last time, we covered 3 algorithms for parallel matrix vector multiplication.
- In the isoefficiency calculations, it takes into account the overhead of parallel programs, in most cases we are referring to the communication costs.
- Data movement is often the biggest energy hog in a CPU. It's bigger that the energy spent on compute.
- If you have processors in a ring structure, how much time does it take for communication between a sender node and a receiver node. T_s is the time for setup and T_h is the per-hop (physical link speed) time and T_m is the per-word time.
    - For example, T_comm = T_s + (N * T_h) + (M * T_m). N is the number of hops and M is the number of words.
    - Often times, the (M * T_m) term will overshadow the others when you have a long message. And thus, we normally use T_comm = T_s + (M * T_m).
- If you have 8 processors in a ring (2 rows of 4 type of rectangle). If we want to do a one to all broadcast, then the trivial algo would be to have the first node send messages to all which is (p-1).
- A better algorithm would be to have 0 communicate with 4 and say that 0 will handle 4 of the nodes (lower nodes), and 4 will handle the other 4 nodes (upper nodes). 3 talks to 1, 4 talks to 6, and then 1 and 6 send to the neighbors.
    - Complexity is (T_s + (M * T_m)) * log (P).
    - The big theme here is that you have the first node send to the midpoint, and then that midpoint will send to its neighbors.
- For the case of a 2D mesh of processors, then you basically do the same thing as 1-D except you send messages to the midpoints in both directions.
- For the case of a hypercube, then you have same thing except need to send messages to the nodes at different heights.
    - In terms of MPI code, then you do send/receive commands based on what point you are at.
    - When you are at other points in the cube not at the origin, you want to do the broadcast operation as if you are at the origin. You XOR the point with itself, XOR that point with the other points and relabel them, then broadcast, and apply XOR again to get back to the original points.
    - The complexity of the scatter is (T_s * log (P)) + (T_m * M * P)
- For an all to all broadcast, the concepts are similar for rings, 2D mesh, hypercubes. This is a situation where each node will have a different message to send to all the other nodes.
    - The trivial solution is p * (p-1) where each processor sends to each of the other processors.
    - Other algo would be to send counter-clockwise if we had a ring. After the first step, every node will have their message plus the message to the left. The 2nd round (and every round until the p-1 round) will involve each node passing on

what you've just received. At this point, the nodes are getting messages from nodes at distance 2 from it.
        - Complexity is (T_s + (M * T_m)) * P
    - For the same algo if you had a 2-D, it's really just applying the ring algo in parallel across all of the rows and imagining the each node at the end of each row connects back to the first node at the beginning of each row. Then, you repeat with the columns instead.
        - Complexity is (T_s + (M * T_m)) * sqrt(P) + (T_s + (M * T_m * sqrt(P))) * sqrt(P)

2/20 - Week 7

- CUDA influenced OpenCl.
- Tesla M60 GPU Info
    - 2048 x 2 Cuda Cores, 8 GB x 2 Memory size, and 160 GB/s x 2 memory bandwidth.
- OpenCL (Open Computing Language) is big because it is a framework for writing programs that execute across heterogeneous platforms like CPUs, GPUs, FPGAs.
    - Allows us to not have to rewrite a program 3 times. However, getting the same amount of performance is not always possible.
- GPU kernels are written using the SPMD programming model where you execute multiple instances of the **same program** independently, where each program works on a **different portion of the data**.
    - Message Passing Interface (MPI) is used to run SPMD on a distributed cluster
    - POSIX threads (pthreads) are used to run SPMD on a shared memory system
    - Kernels run SPMD within a GPU
- One key to note with the above programming model is that in CPU, the overhead of creating threads is so high that the chunks need to be large so that you don't have to create as many threads and each thread can be busy doing its own work.
    - In GPU programming, there is low overhead for thread creation.
- Heterogeneous computing has been a lot bigger than homogenous computing.
    - Of all the top supercomputer, there are a lot of cores and a lot of the systems are heterogeneous.
- Every device (GPU, FPGA, etc) has compute units and within each unit, there are a lot of processing elements (PEs). This is a two level hierarchy. 2 Level hierarchy composed of
    - Work group - compute unit
    - Work item - processing element. It's basically an instance of a kernel or a thread.
- In OpenCL, each instance of a kernel is called a work-item and those are organized as work groups that are independent from one another.
    - Those kernels can be ID'd by global ID, work-group ID, and local ID within the workgroup.
- Memory Model in OpenCL

    - Every device also has a global memory that is shared by all the work-items.
    - There is also a constant memory that is part of the global memory, but the values in it can't be changed.
    - Local memory allows elements within a workgroup to share a memory.
    - Private memory is private to each work-item.
- Memory management in OpenCL is explicit. Must move data from host memory to device global memory, from global memory to local memory, and back
- There is a host code and a kernel code in OpenCL.
    - Host sets up environment for OpenCL and creates and manages the kernels.
        - Define the platform … platform = devices+context+queues. Create and Build the program (dynamic library for kernels). Setup memory objects. Define kernel (attach arguments to kernel function). Submit commands … transfer memory objects and execute kernels
            - Different companies can have their own implementations of OpenCL and each will define a platform that enable the host system to interact with OpenCL-capable devices.
            - Installable client driver (ICD) model is where you let different vendors create platforms and implementations that coexist.
    - Kernel code is the code that the work-items run.
- Platform model slide
    - Host is what the OpenCL library runs on (what type of CPU), devices are processors that the library can talk to (CPUs, GPUs, etc)
        - **The model consists of a host connected to one or more OpenCL devices**
        - **A device is divided into one or more compute units**
        - **Compute units are divided into one or more processing elements**
            - **Each processing element maintains its own program counter**



- First need to select the platform and then that creates a list of devices that the platform knows how to interact with.
- Processing elements within each compute unit can synchronize but there is no synchronization of PEs across compute units.
- There is loose synchronization between kernels.
- GPUs can start and suspend threads very quickly so that
- Some OpenCl functions
    - Get_group_id(dim)
    - Get_num_groups(dim)
    - Get_local_id(dim)

    - Get_local_size(dim)
    - Get_global_id(dim)
    - Get_global_size(dim)
- Need to think about mapping the thread to the data
- When coding with OpenCl, we need to specify a context so that the host can communicate with the device. Inside of the context, we have the following components.
    - There is also a command queue that creates a relationship between a context and the device. It is the mechanism for the host to request that an action be performed by the device. We need to create it using a function. Once we do that, we can enqueue it with data and be able to do execution. We also need buffers to communicate.
        - Separate command queue is required for each device.
    - You also have to store the actual program to run on the device.
    - Need to use createWithSource, then BuildProgram, then createKernel, and then we have to execute the kernel using enqueueNDRangeKernel. Then, when we're done, we do a enqueueReadBuffer.
    - The kernel is the part that runs on the devices.
- Memory objects are OpenCL data that can be moved on and off devices
    - The objects can be buffers (contiguous chunks of memory) or they could also be images.
    - Transferring data to and from the devices involves clEnqueue. Sending data from host to device is considered a write, while the other way around is considered a read. The data will both be part of the context on the host as well as physically on the device.
- Program objects are basically collections of OpenCL kernels. A program object is created and compiled by providing source code or a binary file and selecting which devices to target.
    - **clCreateProgramWithSource** is the function call.
- Then, once you create that program object, you have to call **clBuildProgram** which will compile and link an executable from the program object for each device in the context.
    - If there is a compilation failure then you need to look at the information in CL_PROGRAM_BUILD_STATUS.
- There is a bunch of overhead with compiling programs and creating kernels, but that operation only has to be done once and then the program can continue with the work-items executing the kernel.
- Specifically, a kernel is a function that is executed on an OpenCL device. Kernel objects are created from a program object through the call **clCreateKernel** by specifying the name of the kernel function.
    - Memory objects and individual data values can be set as kernel arguments
- Etc

    - **__global – memory allocated from global address space**
    - **__constant – a special type of read-only memory**
    - **__local – memory shared by a work-group**
    - **__private – private per work-item memory**
    - **__read_only/__write_only – used for images**
    - **Kernel arguments that are memory objects must be global, local, or constant**

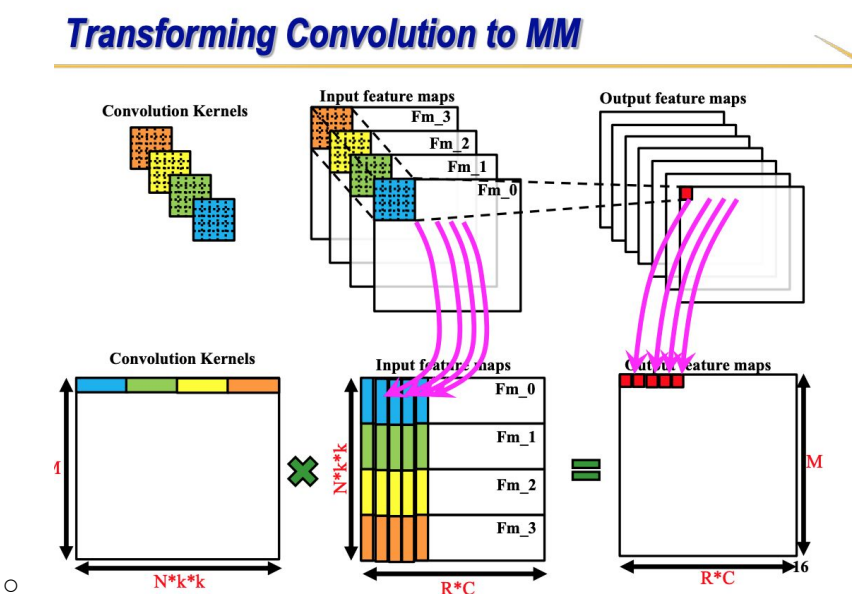
- Kernels execute asynchronously from the host - clEnqueueNDRangeKernel just adds it to the queue, but doesn't guarantee that it will start executing.
- **clEnqueueNDRangeKernel** is what tells the device associated with a command queue to
  begin executing the specified kernel and this is where the global and local sizes are specified.
- Last step is to copy the data back from the device to the host with **clEnqueueReadBuffer**.

2/25 - Week 8

- Command queue allows you to enqueue data as well as the kernels.
- Imagenet was a competition with 1.2 million training images and 1000 categories.
- Convolution operations in CNNs take most of the computation time, but the fully connected layers are the ones that require the most storage capacity since the number of parameters is a lot larger.

	CONV	POOL	ReLU	Fully connect
Computation Complexity(10^6)	30600	6.12	13.5	122.7
Percentage	99.5%	0.0%	0.1%	0.4%
Storage Complexity (MB)	113	0	0	471.6
Percentage	19.3%	0.0%	0.0%	80.6%
Time% in pure software	96.3%	0.0%	0.0%	3.7%

- Throughput is important because we want to process lots of images, and latency is important because we want to get the outputs of the network very quickly.
- Try to convert turn the filters and the matrices into vectors that we can take the dot product of.



2/27 - Week 8

- A memory system has to worry about latency (_ ns) and bandwidth (GB/s)
    - Latency is the time from the issue of a memory request to the time the data is available at the processor.
    - Bandwidth is the rate at which data can be pumped to the processor by the memory system.
- CPU performance goes up by around 60% each year.
    - However, latency decreases at 4-8 percent
    - Bandwidth increases at 20 percent.
    - Both still have less improvement than CPU performance. The big idea is that memory and accessing data is not as fast as CPU computation.
- Need to make use of caching in order to make memory performance better.
- The cache is located very close to the CPU, and it also has pointers to a 2nd cache, and then you have pointer to the main memory and then a pointer to disk.
    - First cache: Latency of < 1 ns, size is 1 Kb
    - Second cache: Latency of 10 ns, size is 1 Mb
    - Main memory: Latency of 100 ns, size is 1 Gb
    - SSD: Somewhere in between main memory and disk
    - Disk: Latency of 10 ms, size is 1 Tb
    - Tape: Latency of 10 sec, size is 1 Pb
- If you have a system with 2 MACs and you're running at 1 GHz, you get 4 GFlops/sec which is the peak. This is because each MAC does one multiply and one accumulate, which is 2 operations.
    - Then, let's say you have main memory and the time to access to 100 ns. You want to do a dot product between 2 4K vectors.
        - In 200 ns, you can do 2 operations, which means you get 10 MFlops/sec.

        - The reason for this being less than the peak is the time to get the elements from the arrays from memory.
    - Let's say we keep that dot product task and add a 32 Kb cache.
        - Transfer A and B from main memory into cache (4K vector x 2 vectors x 100 ns = 800 microseconds)
        - The performance would be 8K operations (multiply and the add for each index) / 800 microseconds = 10 MFlops/sec.
    - Now, if you add a 32 Kb cache and you have the task of multiplying 2 32x32 matrices (each of these matrices is 1 K words which is 4 Kb).
        - Transfer A and B from main memory into cache (2K words x 100 ns = 200 microseconds)
        - Then, the number of ops is $2n^3 = 2 * 32^3 = 64K$ operations, which takes 16 microseconds.
        - In total, we've done 64K operations in 216 microseconds, which is 303 MFlops.
    - Now, let's say we have the main memory situation, and we can read 4 words at a time instead of reading one at a time. This will improve the performance from 10 MFlops to 40.
- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units. Memory bandwidth can be improved by increasing the size of memory blocks.
- Another way to improve performance (besides caching) would be prefetching (predicting or computing the value before you get it).
- Another is to use multi-threading to hide the memory latency, but it requires high bandwidth.
    - So if given the choice between low latency and high bandwidth, choose bandwidth because
- In general, the time required $T = f * t_f + m * t+m$ where f is the number of operations, t_f is the arithmetic time, m is the amount of memory, and t_m is the access time.
- In a matrix * vector problem, $m = n^2$, $f = 2n^2$ and thus $q = f/m = 2$.
- In a matrix * matrix problem, $m = n^3 + 3n^2$, $f = 2n^3$ and thus $q = f/m = 2$.

3/4 - Week 9

- To improve the speed for accessing and modifying memory, we can use caches, multithreading, and prefetching.
    - Caches help with both latency and bandwidth. Temporal locality helps with bandwidth and spatial helps with latency.
    - Multithreading and prefetching helps to hide the latency. There needs to be a minimum level of bandwidth necessary in order to hide it.
- GPUs have lots of cores and they rely on fast context switching.

- Normally, in CPU computation, we have an array of threads and each thread runs some kernel code (the same code for each thread - SPMD) and then depending on what ID the thread has, it will do computation on a different piece of the data.
    - With a GPU, instead of a thread array, we have thread blocks and each block has an array of threads inside that can communicate with each other (can synchronize and have a low latency shared memory).
        - So now, in GPUs, kernels will be executed by a grid of these thread blocks.
        - In CPU, a thread just had to get its ID, but now, a thread needs to get the ID of the block it is a part of, and then an ID for its location within that block.
- The thread blocks are equivalent to work groups in OpenCl.
    - Those thread blocks will get divided into threads (equivalent to work items in OpenCl)
    - In CUDA, the thread blocks are divided into WARPs which is a group of 32 threads.
        - You can switch from one WARP to another in just one clock cycle.
        - We will have more WARPs than cores.
- Older model of GPU - G80
    - Composed of group of streaming multiprocessors (SM) which are composed of streaming processors (SP).
    - A thread block maps to an SM. (thread block = software, SM = hardware because the SMs execute the blocks). You could also have more than one block map to a particular SM. Each SM launches WARPs of threads.
    - We care about # SP / # SM as well as the number of SMs.
        - For G80, # SP / # SM = 8 and #SM = 16 and thus #SP = 128
- So like we mentioned the SM is hardware that launches 32 threads at a time in groups called WARPs. The SM is the one that implements WARP scheduling.
    - Warps whose next instruction has its operands ready for consumption are eligible for execution.
    - All threads in a Warp execute the same instruction when selected.
    - When scheduling these warps, we fetch one instruction, issue one warp instruction/cycle and have the SM broadcast that instruction to the 32 threads of a warp.
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
    - Reason: single issue, one warp has 32 threads, one SM has 8 processors. If one global memory access is needed for every 5 instructions, a minimal of 10 Warps are needed to fully tolerate 200-cycle memory latency.
- Dividing program into WARPs is device independent.
- Inside each core, you have a single shared memory (called local) for the k different threads. Each of the threads also has its own private memory and they have their register files.

    - Thus, threads within the same block can communicate by modifying the shared memory.
    - There is also the concept of a global memory which all the threads in all the thread blocks can write/read to. That global memory is your DRAM.
        - This is the way to communicate between host and device.
    - TI;dr Each thread can R/W per-thread registers, R/W per-thread local memory, R/W per-block shared memory, R/W per-grid global memory, Read only per-grid constant memory, Read only per-grid texture memory.
- In terms of the lab, have to decide what data should be kept in global memory, shared memory, and local memory.
- For the GPU for the lab, it has the following stats:
    - #SP/#SM = 128, #SM = 16, #cores = 2048, #blocks/SM = 32, # Warps/SM = 64, Max # threads/SM = 2048.
    - Each SM can have a max of 64 WARPs (2048 threads) and something about 32 blocks. In each SM, we only have 128 SPs.
- Scenarios with that GPU
    - A workgroup with 64 x 64 work items
        - Won't work because you have about 4K threads/work items, and our SM has a max of 2048 threads.
    - A workgroup with 32 x 32 work items and each uses 70 registers.
        - No. Each work item can have 255 registers, and each block has a max of 64K registers, in this situation you have 70K registers.
    - A workgroup with 4 x 4 work items and each uses 1000 registers.
        - No. Each work item or thread can only have 255 registers, so that constraint is violated.

3/6 - Week 9

- When you're writing GPU programs, you will write code that a kernel will execute. The kernel will be composed of work groups and those will have work items.
    - This will get mapped to the physical GPU. It'll get mapped to SMs and SPs within those.
        - Multiple work groups can map to one SM
    - 128 SPs in an SM means that 4 WARPs will get mapped from the workgroup to the hardware.
- So with this GPU, we have a 8 x 128 grid of SPs. The 8 refers to the number of SMs and in each SM we have 128 SPs.
- For getting the best performance, we want to:
    - Keep everyone busy
    - Do memory optimization for latency and bandwidth. In other words, want to make sure that we're loading in the right amount of data in the right order.
    - Optimize for a given GPU.
    - Instruction throughput.

        - Want single precision computation if we can.
        - Strength reduction in that you replace x2 with shift operations.
        - Minimize divergence.
    - Memory coalesce in order to grab the largest pieces of memory we can.
    - Memory interleaving in order to have each thread grab specific parts of the data.
- The shared memory for our GPU is 96 KB.
    - In your A * B = C calculation, you have 3x^2 <= 96/4 or 19 kiloWARPs and x^2 <= sqrt(8 kiloWARPs). This means 89 x 89 threads inside your block.
    - 3 since you have 3 matrices to look at.
- FPGAs allow you to do customized computation that will have good performance on your own unique workload.
- It has customizable logic and interconnect.
    - Logic: One example is implementing a K bit input boolean function, where for each possible input you have, you have a corresponding boolean output value which is shown in the truth table.
    - Interconnect: Have memory cells that will tell you whether to have pass or stop signal at a transistor. Path transistors are used to direct signals to different parts on the chip.
- Lot of fixed logic cells, digital signal processing units (can do multiplication and accumulation), and BRAM (76 MB in total, each is 36 Kb) are put into the FPGAs.
    - There is also a URAM (270 MB) which is memory for the whole FPGA.
    - There is also a DRAM (16 GB)
    - The FPGA is also connected to a host via PCI-e.

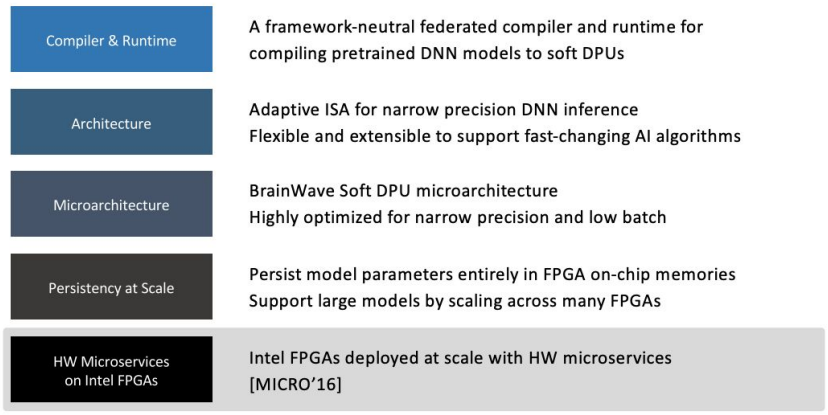
3/11 - Week 10

- CPUs and GPUs have fixed architectures and the program is adapted to the architecture. You cannot change the silicon and cannot change the connections. The compiler takes the program and converts it into instructions that will fit with the architecture given.
    - FPGAs on the other hand, have a customizable architecture and the architecture is adapted to the program. They have programmable logic blocks, logic gates (Use transistors to implement combinational and sequential logic), programmable interconnects (Wires to connect inputs and outputs to logic blocks), and programmable I/Os (Special blocks at periphery for external connections).
- An FPGA contains:
    - Lookup tables: Aka memory.
    - Sequential elements:
    - Dedicated on-chip memory: BRAM, URAM: These are sprinkled all around the chip.
    - Arithmetic: DSP engine.
- It is harder to program FPGAs because it involves learning Verilog and stuff like that.

- In a FPGA, you aren't stuck with a particular bit datapath and the number of caches, etc. You get custom bit-width, memory hierarchies, and custom datapaths.
- You can also have a deeply pipelined implementations, bit manipulations, wide datapath, and custom memory hierarchy.
- For the lab
    - Create multiple compute units to execute the same kernel.
    - Maximize the number of transfers and the average size.
    - Possibly divide the data into a wider array. Data partitioning stuff

3/13 - Week 10

- Using the Merlin compiler will allow you to create optimized code for FPGA acceleration.
- In CPUs, in order to add two numbers together, you need to run the following instructions: fetch, decode, rename, load A/B, schedule, wake-up, execute A+B, register file, store.
- In GPUs, you have SPMD meaning that you only have to do the fetch, decode and rename steps once and then you execute the same instruction on lots of data with lots of processors.
- In FPGAs, you don't have instructions per se, you create a program which creates a certain circuit and the program just runs. There is way more flexibility.
- As the amount of power for CPUs increases, parallelization was thought of as the solution, and now the next step is customization and specialization.
- Processor ALUs are a lot more expensive energy wise than dedicated units.
    - In a normal processor, 36% of energy is spent on non-compute, while it's only 11% in custom ASICs. The reason you get the efficiency is custom bit-width, custom memory architecture, and predictable communication patterns. The only problem is that it is too costly and time expensive to build an ASIC.
- The custom computing proposal focuses on having dedicated and composable accelerators that are more specialized and energy efficient.
- Customization can happen at different levels
    - Single-chip level - Require new processor designs, e.g. using fixed-function or composable accelerators
    - Server node level - Host CPU + FPGA via PCI-e or QPI connections
    - Data center level - Clusters of heterogeneous computing nodes
- FPGA helps with ML models because of:
    - Excellent inference performance at low batch sizes. Ultra-low latency serving on modern DNNs
    - Their flexibility when it comes to implementing models with lots of different operations.
    - Exploit sparsity, deep compression for larger, faster models
- BrainWave Stack



- Hardware utilization increases with batch size up to a plateau, but the latency increases exponentially. Want high utilization + low batch sizes.
-

Final Exam Prep Questions

1. Please review again all your lab designs after the mid-term exam, and make sure you (i) pay attention to the OpenCL syntax you used in your program; (ii) understand which parallelization or optimization techniques help you to achieve the most significant speedup; (iii) be ready to extend these techniques to other problems.
2. Please review and compare the CPU, GPU, and FPGA architectures, understand their similarities and differences, and implication to performance and energy efficiency of computation.

	CPU	FPGA	GPU
Overview	Traditional sequential processor for general-purpose applications	Flexible collection of logic elements and IP blocks that can be configured and changed in the field	Originally designed for graphics; now used in a wide range of computationally intensive applications
Processing	Single- and multi-core MCUs and MPUs, plus specialized blocks: FPU, etc.	Configured for application; SoCs include hard or soft IP cores (e.g., Arm)	Thousands of identical processor cores
Strengths	Versatility, multitasking, ease of programming	Configurable for specific application; configuration can be changed after installation; high	Massive processing power for target applications—video processing, image analysis, signal

		performance per watt; accommodates massively parallel operation; wide choice of features: DSPs, CPUs	processing
Weaknesses	OS capability adds high overhead; optimized for sequential processing with limited parallelism	Relatively difficult to program; second-longest development time; poor performance for sequential operations; not good for floating-point operations	High power consumption, not suited to some algorithms; problems must be reformulated to take advantage of parallelism, but API frameworks provide abstraction

FPGAs are good with low latency. They have good connectivity and FPGAs can directly be connected to inputs and can offer very high bandwith. The engineering cost is typically much higher than for instruction based architectures, so the advantages must really be worth it.

CPU is a general purpose processor. General Purpose in the sense that it is designed to perform a number of operations but the way these operations are performed may not be best for all applications. Graphics or Video Processing is one such example. Although a CPU can perform these tasks (which involve repeated additions/multiplications which may be performed in parallel) , the performance achieved is not good enough for modern applications.

GPU Graphics processing Unit or GPU is designed to accelerate creation of images for a computer display. A CPU consists of a few cores optimized for sequential serial processing while a GPU consists of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. They are designed to perform functions such as texture mapping, image rotation, translation, shading, etc. They may also support operations such as motion compensation, calculation of inverse DCT, etc. for accelerated video decoding.

FPGA (Field Programmable Gate Array) is entirely different from CPU, GPU, DSP, etc. in the sense that it is not a processor in itself i.e. it does not run a program stored in the program memory. In layman's terms, an FPGA is nothing but a bulk of reconfigurable digital logic suspended in a sea of programmable inter-connects. A typical FPGA may also have dedicated memory blocks, digital clock manager, IO banks and several other features which vary across different vendors and models. Since they can be configured after manufacturing at costumer's end, they can be used to implement any logic function (including but not limited to a processor core). This makes them ideal for re-configurable computing and application specific processing. Intel has recently announced a new range of Xeon Processors with integrated FPGA so that each chip can be configured at run type depending upon application needs. FPGA is massively

parallel – each FPGA includes millions of parallel system logic cells. It is flexible – no fixed instruction set, can implement wide or narrow datapaths. It is programmable using available, cloud-based FPGA development tools

3. Please review and compare OpenMP, MPI, and OpenCL programming languages that you have learned in this class, and understand their similarities and differences.

OpenMP vs MPI: OpenMP is a language-extension for expressing data-parallel operations (commonly arrays parallelized over loops). MPI is a library for message-passing between shared-nothing processes. OpenMP is a higher-level of abstraction, since its purpose is to expose the program's concurrency and dataflow to the compiler. By contrast, MPI concurrency is implicit (all processes are parallel), and the messages establish the dataflow structure of the computation. MPI focuses on using message passing paradigms which is generally shared nothing. OpenMP focuses on shared memory paradigms. Both support C,C++ and FORTRAN.

OpenMP is based on thread approach. It launches a single process which in turn can create n number of thread as desired. It is based on what is called "fork and join method" i.e depending on particular task it can launch desired number of thread as directed by user. OpenMP is well-suited for domain/data decompositions but not well-tailored for functional decompositions. Compilers also do not check for such errors as deadlocks and race conditions.

MPI is available from different vendor and can be compiled in desired platform with desired compiler. One can use any of MPI API i.e MPICH, OpenMPI or other. OpenMP are hooked with compiler so with gnu compiler and with Intel compiler one have specific implementation. User is at liberty with changing compiler but not with openmp implementation.

MPI target both distributed as well shared memory system while OpenMP target only shared memory system. MPI based on both process and thread based approach while OpenMP is just thread based. MPI has overheads associated with transferring message from one process to another while OpenMP can use shared variables.

OpenMP is a way to program on shared memory devices. This means that the parallelism occurs where every parallel thread has access to all of your data. You can think of it as: parallelism can happen during execution of a specific for loop by splitting up the loop among the different threads.

MPI is a way to program on distributed memory devices. This means that the parallelism occurs where every parallel process is working in its own memory space in isolation from the others. You can think of it as: every bit of code you've written is executed independently by every process. The parallelism occurs because you tell each process exactly which part of the global problem they should be working on based entirely on their process ID.

OpenCL is a low-level API for heterogeneous computing that runs on CUDA-powered GPUs. Using the OpenCL API, developers can launch compute kernels written using a limited subset of the C programming language on a GPU.

OpenCL will always have some extra overhead when compiling the kernel at runtime. OpenCL allows you to parallelize in any kind of device: CPU, GPU, FPGA, etc. that's what it was made for, is a little harder to learn but is more extensible and supports a lot of devices.OpenMP is mostly for "trivial" parallelization, it mostly breaks for loops into [p]threads and that's it, the code will compile even without it installed (but will not be parallelized).

OpenCL involves a lot of background work like memory allocation, kernel settings and loading, getting platform, device information, computing work-item sizes etc. All this adds overhead in OpenCL. However, we find that, in spite of this overhead, OpenCL gives very good performance. But OpenCL fails in application where it has less scope of work; this can be seen from the string reversal example.
