Thanks for the feedback. Here’s my revised proposal:

Topic: Parallel Computing task delegation methods overview

Throughline: Why computers multitask better than you. THIS IS SIMPLER AND POTENTIALLY INTERESTING. YOU WILL TALK ABOUT CS IDEAS.

Audience: Laypeople including students of NUS High School and also the NUSH teacher named Eric Francis Tinsay Valles.

They are expected to be laypeople who don't know the slightest bit about parallel computing.

They are interested in increasing their work productivity.

They care about this perspective because many people know computers are highly efficient, but I think I can make a case for how some methods used (e.g. transform the cuda threading system into the explanation "divide your work into standalone tasks, then solve each of them" or perhaps job dispatching with multiprocessors could be applied to real life prioritizing of tasks) can be applied to real life productivity.

Am I passionate about this topic? Yes

Does it inspire Curiosity? Yes

Will it make a difference to the audience to have this knowledge? They will learn to multitask like a computer, and people interested in compsci will pick up some knowledge on why their application multithreaded keep is breaking.

Can I explain the topic within the time limit: Yep time summary below

What are the 15 words that encapsulate my talk? Throughline above

Would the 15 words be persuasive: What do you think?

Talk Segments

15s Introduction: (Tense music) (Many many piles of work stacking up, a cartoonish face cries cartoonishly in cartoonish despair)

30s Context: (Music Cuts) (Picture a poorly animated clip art wizard) STOP! You have probably faced many unmeetable deadlines and undue amounts of stress. I, Dr MicroProcessor, can relate with you! But I've got good news, there's some nifty tricks up my sleeves you can use to work faster like me!

2m 15s Main concepts & Examples: The concept of parallel processing spans quite a vast area of topics. I personally am already familiar with the CUDA architecture, but the reader does not want this, they want multitasking methods. I will first define the multitasking problem using diagrams so I can work off that. I will then cherry pick specific innovations from current parallel processing technology that can be translated into human productivity methods. They will be unique ideas that should follow a "cover all the bases" strategy, each idea addressing typical problems faced by overworked individuals. THIS SHOULD BE MORE DETAILED, ESPECIALLY THE CS ELEMENTS. THIS IS APPROVED.

45s Practical Implications: Quick explanation of how our simplified box diagrams get used in actual technology, should be done in a dramatic reveal style (this productivity method? ChatGPT uses it to optimize response time!!?!! no way), and display some graphics to emphasise the increase in efficiency obtained.

15s Conclusion: And that's my talk on how you should multitask. After all, if we strive to be like computers, we might even surpass them one day!

Story of CUDATale.

We feature the protagonist Sugar Rush, embarking on a quest to save themselves from dying.

Sugar Rush is a children’s edutainment mascot. They appear as an anthropomorphic cloud of glitter, often compared to the “magic” emoji that is commonly associated with artificial intelligence in the modern age. This has led fans to draw comparisons between Sugar Rush and fictional Ais, such as Rambley, Monika, etc, lending an existential horror aspect to the otherwise innocent portrayal. Their physical state allows them to morph into recognizable shapes, and will otherwise remain in the default “3 stars” pose, occasionally changing the configuration of the stars to imitate human expression. Critics have often slammed the simple artstyle of Sugar Rush as cheapening the experience, and a means to save money.

The plot of the Sugar Rush animated web series revolves around the maintenance of their “server”, bringing real life stories about challenges faced by the actual developers when maintaining the real-life Sugar Rush servers to the realm of children’s entertainment. (there’s something here about how the animations were AI generated, and the Sugar Rush developers didn’t bother hiring actual artists, so they made up the story themselves).

We enter arc 136 season 1 episode 3 (Sugar Rush effectively reboots the story each arc (5302 episodes so far), so there is no lore established yet.

Hhhh we need to make this dumb glitter cloud embark on a quest to optimize their inference time. Ok so

Content to teach impressionable curious kids:

- CUDA and how it interfaces with GPUs

- Haha so basically the cuda code first needs to compile right? We make it turn into ptx assembly, and then from there it goes into driver assembly, and gets compiled into a fatbinary. cuda runtime libraries link. The fatbinary is then accessed through jump instructions when we want to run device specific assembly. (moment of silence for the cut content)

So then the instructions go towards the gpu right? So for some number of instructions, we’ll assign a vacant SM to have some number of warps each containing 32 threads. Those warps will be filled with the necessary instructions, and then pipelining and scheduling will be used to optimize throughput of the SM processing of data. We need to talk about how data is processed by each If some code contains branching, all branches will be “executed”, but inactive execution paths will be blocked. Talk about what each core specializes in (this has to be done in the context of pipelining, since that’s what’s relevant to our topic of multitasking). And that’s the story of CUDA!

Scheduling + pipelining are the 2 big things I need to describe in detail.

…I might drop one of them to be honest, or find a better pitch for tying the 2 together. Because it’s probably confusing if my multitask talk is split between scheduling and pipelining.

Realistically there’s not much open knowledge about scheduling as compared to pipelining.

I can make pipelining a subidea of scheduling

Like for instance

If we say that our task is to allocate N tasks between J subprocessors, we can say that allocation in a specific way should consider pipelining when making the scheduling decision.

This should all follow a tree diagram, lemme go model that

- each streaming multiprocessor has a L1 cache

- Couple scheduling algorithms (gto, lrr)

Ugghh guess I’ll put in a generic explanation of an alu.

The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multipro-cessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread) that is described in SIMT Architecture. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading as detailed in Hardware Multithreading. Unlike CPU cores, they are issued in order and there is no branch prediction or speculative execution.

Since our throughline is that computers multitask better than you, and we as biology haters do not understand enough to make a case for this, we just need to make a case for computers being really good, and then say that it’s doing it better than a human! Real.

**[ 20s ]** *So you want to know how computers multitask?*

**[ 00s ]** *GPU architecture, etc etc, streaming multiprocessors. We use Round Robin Scheduling to delegate tasks among SMs, which is to execute the first instruction in all tasks first, then 2nd instruction, 3rd and so forth.*

**[ 50s ]** - GPU architecture

The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multipro-cessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread) that is described in SIMT Architecture. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading as detailed in Hardware Multithreading. Unlike CPU cores, they are issued in order and there is no branch prediction or speculative execution.

**[ 30s ]** - Round Robin Scheduling

Round-robin is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can be applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept. The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

**[ 10s ]** *This is useful in that when we execute 1 instruction, we can follow up with another of the same instruction to save time through pipelining.*

**[ 50s ]** - ALU Design and Clock Cycles

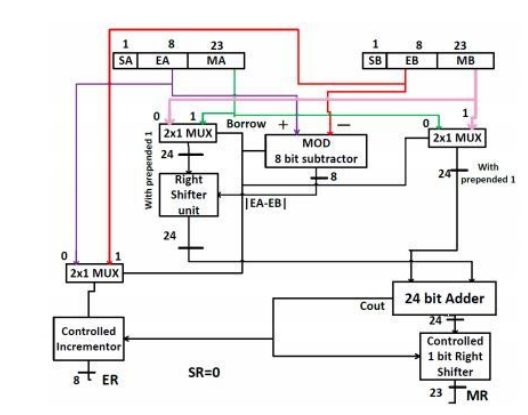
When we want to execute an instruction in the gpu, there are often multiple steps required to do so. For instance, maybe we’d want to execute a multiply instruction. For instance, in the first clock cycle, we may alter a circuit state to take in operands 1 and 2, then the next k clock cycles we will incrementally calculate the final result of a multiply operation.

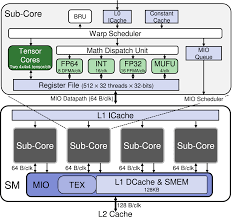
**[ 30s ]** - Idea that when you have a instruction executing in its 2nd stage, you can call another instruction in its first stage

If we want to do a 2nd multiply operation after the 1st, we can immediately start executing it after the 1st operation.

**[ 10s ]** *And since we’re pipelining, we can use specialized cores for optimized calculation of operations to make the pipeline faster overall*

**[ 30s ]** - Arithmetic Cores

[](https://camo.githubusercontent.com/6b2c80d6dbea5c21ef2a1c9c3dbc570b701023d4ce6817d8b12d7b2e106f965a/68747470733a2f2f692e6962622e636f2f54776770346b782f53637265656e2d53686f742d323032302d30352d30342d61742d372d33392d35362d504d2e706e67)https://github.com/ahirsharan/32-Bit-Floating-Point-Adder



**[ 30s ]** - Special Function Cores

**[ 30s ]** - Tensor Cores

A diagram of a computer system

Description automatically generated rev.pdf

**[ 10s ]** *So through scheduling, we leverage pipelining which takes advantage of specialization. That’s how computers multitask efficiently. Thanks for listening.*

300s

**[ 50s ]** - GPU architecture

So we have a program that wants to do 1 billion tasks, and a single GPU. Here’s how it works. The program will first split the tasks into a number of blocks. Each block will be further subdivided into a number of threads, each thread representing one task we need to execute. A number of blocks will then each be assigned to a warp. This warp will then be sent to a single GPU processor, known as a Streaming Multiprocessor. The Streaming Multiprocessor will then execute each block depending on a warp scheduler. The typical warp scheduler algorithm that is used is based off Loose Round Robin. This works as follows: Each block is assigned an allocated amount of time to execute its instructions, when this amount of time is up, the scheduler will then switch the streaming multiprocessor to execute a different block. This continues until all blocks have finished execution, which means that at any given time, all blocks are at around the same point in execution. A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread) that is described in SIMT Architecture. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading as detailed in Hardware Multithreading. Now, I’ll go into detail on what happens inside the streaming multiprocessor. Since every block has a number of threads, we will need multiple processing points to achieve parallelism. This is represented by processing units in the streaming multiprocessor, of which consist of multiple different circuits, where data will pass through across multiple clock cycles. When we want to execute an instruction in the gpu, there are often multiple steps required to do so. For instance, maybe we’d want to execute a multiply instruction. For instance, in the first clock cycle, we may alter a circuit state to take in operands 1 and 2, then the next k clock cycles we will incrementally calculate the final result of a multiply operation. *This is useful in that when we execute 1 instruction, we can follow up with another of the same instruction to save time through pipelining.* If we want to do a 2nd multiply operation after the 1st, we can immediately start executing it after the 1st operation. *And since we’re pipelining, we can use specialized cores for optimized calculation of operations to make the pipeline faster overall.* Although the internal design of computing units is not disclosed by NVIDIA, some information has been experimentally obtained by researchers. The CUDA core contains both a int unit and a fp32 unit. Tensor Core has the following structure:

Tensor core must be able to do 16 4x4 dot product multiplications each cycle.

Explain how a fp16 multiplier works.

Explain how a fp32 adder works.

Explaining per octet, starting with octet 0, there are 3 operand buses, since ptx supports up to 3 operands in an instruction. There is also a connection to the register file associated with the streaming multiprocessor. operand bus 2 is sent to a multiplexer, and used as control signals to indicate which operation should be executed by the tensor core. Operand Bus 1 is sent to the matrix buffers, while operand bus 3 is sent to an accumulator. Registers can also be used to add to the matrix optionally. In each tensor core there are 2 octets, making 16 dot product units.

Roll Credits.

80 tensor cores -> 320x32 matrix \

Tasks are serialized into GPU-native instructions, organized into blocks of 32 tasks, and assigned across multiple GPU sub-processors, known as Streaming Multiprocessors.

Instructions may be stored on the L1 or L0 instruction cache, memory units local to the SM for faster retrieval. A block’s first instructions are sent for processing before moving on to subsequent blocks, a strategy known as Round Robin Scheduling which is carried out by the Warp Scheduler.

The instructions to be processed arrive at the Math Dispatch Unit, which delegates instructions to specialized computation units, which process different types of arithmetic and memory operations. Execution of arithmetic operations typically takes many hundreds of clock cycles. However, we can take advantage of the same tasks being executed repeatedly by leveraging the concept of pipelining. Execution of a calculation goes through multiple different circuits, leaving circuits that are not immediately participating in a calculation inactive. We can fill this inactivity by following up with a second calculation, turning our few hundred clock cycle execution into effectively a 1 clock cycle execution time on average. Arithmetic that will take longer will often be prioritized, to overlap operations and maximize use of pipeline stages.

First, we need our operands. Memory operations are queued to the Load Store Unit (LSU), which resolves memory locations for requested data, and sends lookup requests to memory units outside the SM. The LSU then stores our requested operands in the register file. Since fetching data can be slow, we can continue execution while we wait for our data. Our retrieved data can also be stored in caches. We have caches for constant values, arrays and register values among others.

Now, to compute. GPU arithmetic units are split into two sections. One being fast units, which take a set number of clock cycles to execute, and slow units, which take a variable number of clock cycles. Some example of fast units are those for floating point and integer operations, while examples of slow units are the transcendental unit, which calculates trigonometric and root functions, or the double precision unit, which handles 64 bit floating point operations. There also exists an Arithmetic Logic Unit for bitwise operations and half precision floating point conversion.

In recent years, GPUs have included arithmetic units for matrix operations, called the tensor core. Explaining per octet, starting with octet 0, there are 3 operand buses, since ptx supports up to 3 operands in an instruction. There is also a connection to the register file associated with the streaming multiprocessor. operand bus 2 is sent to a multiplexer, and used as control signals to indicate which operation should be executed by the tensor core. Operand Bus 1 is sent to the matrix buffers, while operand bus 3 is sent to an accumulator. Registers can also be used to add to the matrix optionally. In each tensor core there are 2 octets, making 16 dot product units.

Finally, we have the crossbar, which facilities communication between caches and computation units across the SM. Once our computation is complete, it is dispatched across multiple data buses to an IO port, where it is sent to the rest of the system.

(summary summatum)