CSc 8530
Parallel Algorithms

Spring 2019

February 28th, 2019

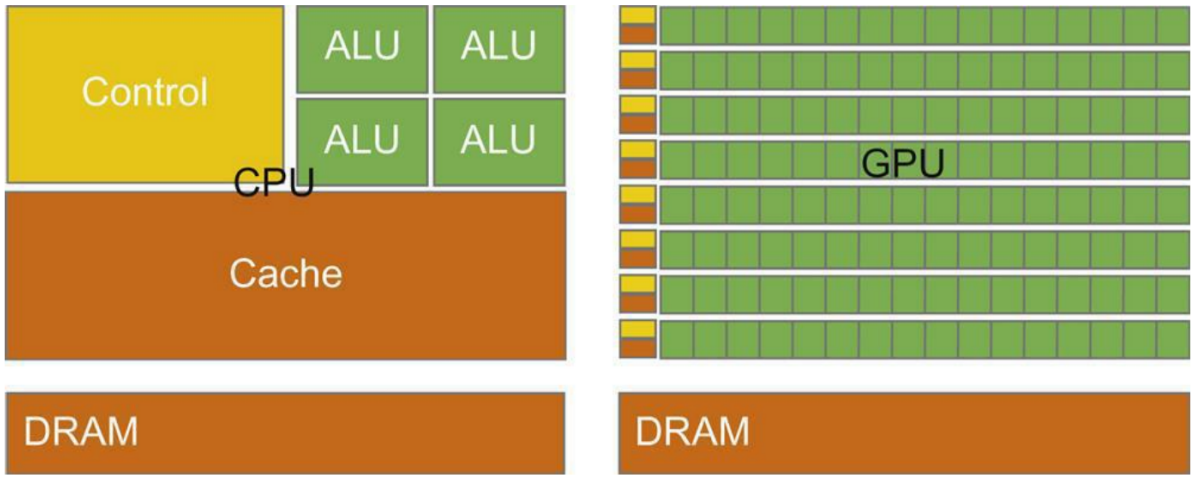
Central processing units

- Present-day computers are based around **central processing units (CPUs)**
  - Based on Von Neumann's single-threaded architecture
    - A **thread** is a sequence of operations in a program (**process** from the point of view of the operating system)
    - A process can have multiple threads
  - CPUs are optimized for sequential processing
- Since 2003, CPU clock speeds have stalled
  - Due to heat and energy consumption limitations
- This led to the introduction of **multicore architectures**
- However, the number of cores in most CPUs is in the single digits or low double digits
  - Still optimized for single-process applications

Graphical processing units

- Sometimes, we need to execute many simple operations at the same time
- Computer graphics are a prime example
  - We need to constantly draw elements on the screen
  - In many cases, we can handle individual pixels separately
- The highly parallel nature of graphics applications lead to the development of the **graphical processing unit (GPU)**
- Multicore CPUs are optimized for running *a few, complex* threads very quickly
  - Minimize **latency**
- GPUs are optimized for running *many, simple* threads quickly
  - Maximize **throughput**

CPUs vs. GPUs



- CPUs minimize single-thread latency and thus have sophisticated control logic and caches
- GPUs maximize throughput (especially of floating-point operations), so they have simple control logic and many arithmetic units

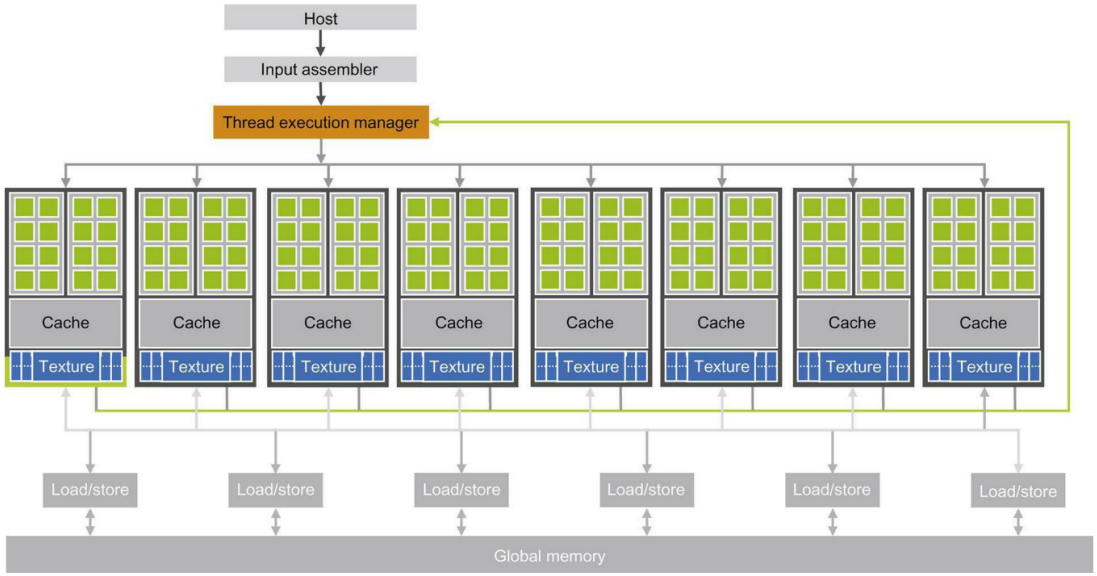
CPUs vs. GPUs

- CPUs and GPUs were designed to solve different types of problems
  - CPUs are best for the sequential parts of a program
  - GPUs for the parallel, numerically intensive parts
  - Programming languages such as CUDA are designed to support joint CPU-GPU programming
- By design, GPUs achieve much higher FLOPs (floating-point operations per second) than CPUs
  - The ratio over the last nine years has been around 10
  - Reducing latency is much more expensive than increasing throughput (why?)
- Also, due to legacy and I/O requirements, CPUs have worse memory bandwidth
  - GPUs can access their dynamic memory about 10 times faster

Latency vs. throughput – an analogy

- Imagine rain falling down a window
- Latency is how quickly individual drops reach the windowsill
- Throughout is how many drops reach the windowsill at (roughly) the same time

Modern GPU architecture



- GPUs have a PRAM-like architecture
  - There is a global or shared memory (synchronous dynamic RAM (DRAM))
- The green squares represent streaming microprocessors (SM)
- Each block of SMs share a cache and control logic
  - Equivalent to a single processor in the PRAM model

Modern GPU architecture

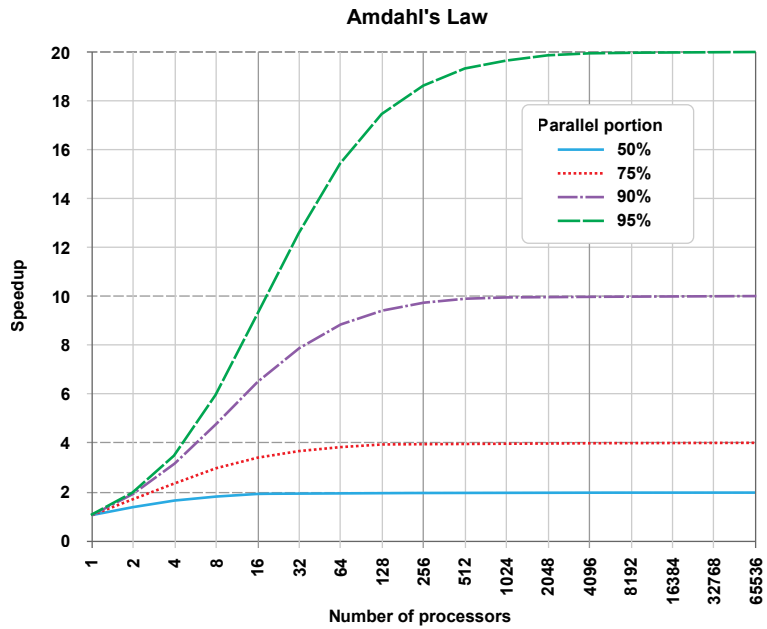
- The GPU typically communicates with the CPU through a PCI-Express interface
  - Around 8-16 GB/s transfer rate
  - NVIDIA also has a proprietary NVLink interface that can go up to 40 GB/s
- We generally want to minimize GPU-CPU communication
  - It is much slower than internal operations on either device
- As GPU DRAM grows, it is possible to keep more of the data needed in global memory at all times
  - e.g., current, high-end graphics cards have between 8-12 GBs of DRAM
- We can typically run between 5,000 and 12,000 threads on a single GPU
  - Compared to 2 to $\sim$32 on a CPU

Parallel speedup – Amdahl's law

- How much using a GPU vs. a CPU will speed up an algorithm depends on how much of it we can parallelize:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

  - $S_{latency}$ is the theoretical speedup
  - p is the percentage of the work we can parallelize
  - s is how much parallelization improves the running time



By Daniels220 at English Wikipedia, CC BY-SA 3.0

Parallel speedup – Amdahl's law

- Example: we can parallelize 30% of the algorithm and run the parallel portion twice as fast:

$$S_{latency} = \frac{1}{(1-p) + \frac{p}{s}}$$
$$= \frac{1}{(1-0.3) + \frac{0.3}{2}} = 1.18$$

- Now, assume we run the parallel portion ten times as fast:

$$S_{latency} = \frac{1}{(1-0.3) + \frac{0.3}{10}} = 1.37$$

- The actual improvement in running time is much more modest

Parallel speedup – Amdahl's law

- Percentage improvement in running time

$$P = 100\left(1 - \frac{1}{S_{latency}}\right)$$

- In the previous example:

$$P(s = 2) = 100\left(1 - \frac{1}{1.18}\right) = 15.2\%$$

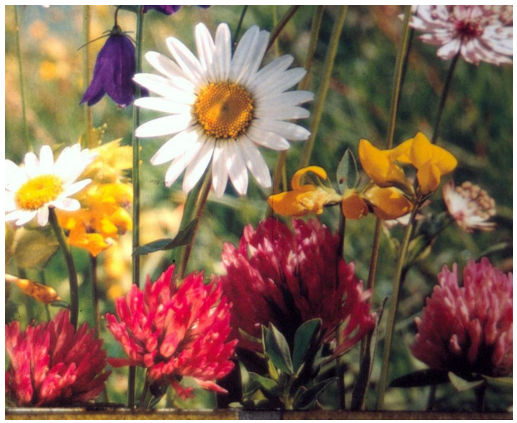
$$P(s = 10) = 100\left(1 - \frac{1}{1.37}\right) = 27\%$$

- How much parallelization (e.g., via GPUs) helps is constrained by the inherently sequential parts of the code

CUDA programming

- In this course, we will use the CUDA programming language
  - Currently the most mature framework for writing GPU-based code
  - Unfortunately, it is proprietary to NVIDIA (i.e., you need an NVIDIA GPU)
  - Potentially, future versions of the course will switch to an open standard (e.g. OpenCL)
- We will first show how to solve a simple, image processing problem (converting a color image to grayscale) with CUDA
  - An example of **data parallelism**
- Data parallelism occurs when we have to apply the same operation to many parts of the data **independently**
  - We have encountered this type of parallelism extensively in the course

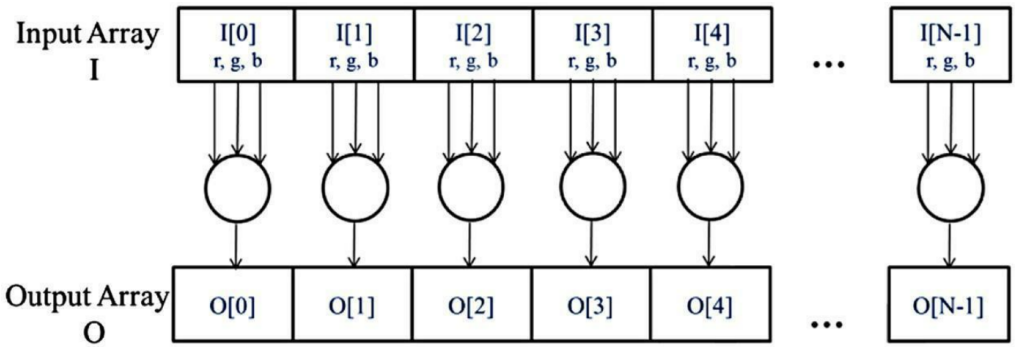
Grayscale conversion



- A color image is typically stored as an $n \times m \times 3$ matrix
  - The most common encoding is RGB (red, green, blue)
- A grayscale image is an $n \times m$ matrix in which we combine the three color channels into a single brightness channel
- A typical formula is $L = 0.21R + 0.72G + 0.07B$
  - The coefficients are derived from psychophysical experiments on how sensitive our eyes are to different wavelengths

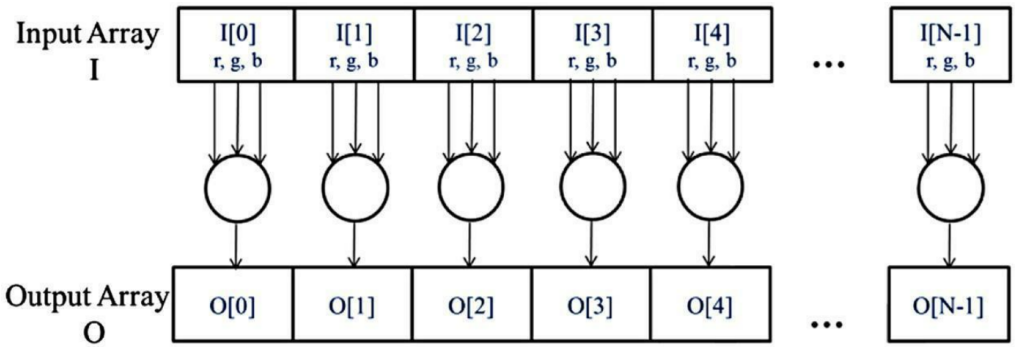
Image by A. Gundelach, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=65916

Grayscale conversion



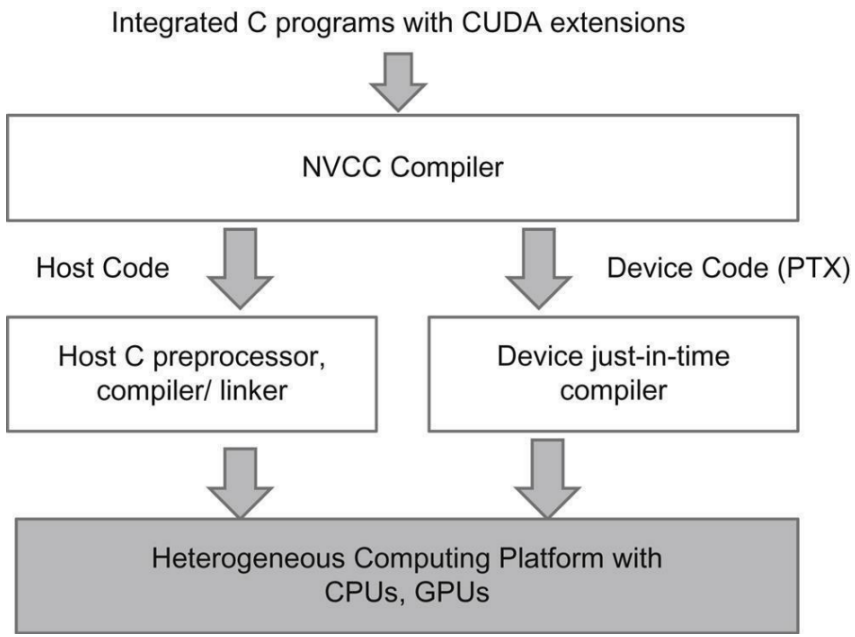
- The formula $L = 0.21R + 0.72G + 0.07B$ is computed on a **per-pixel** basis
- Pixel values are independent of each other
- Hence, we can easily parallelize this operation
- The above diagram can be interpreted as what kind of parallel model?

Grayscale conversion



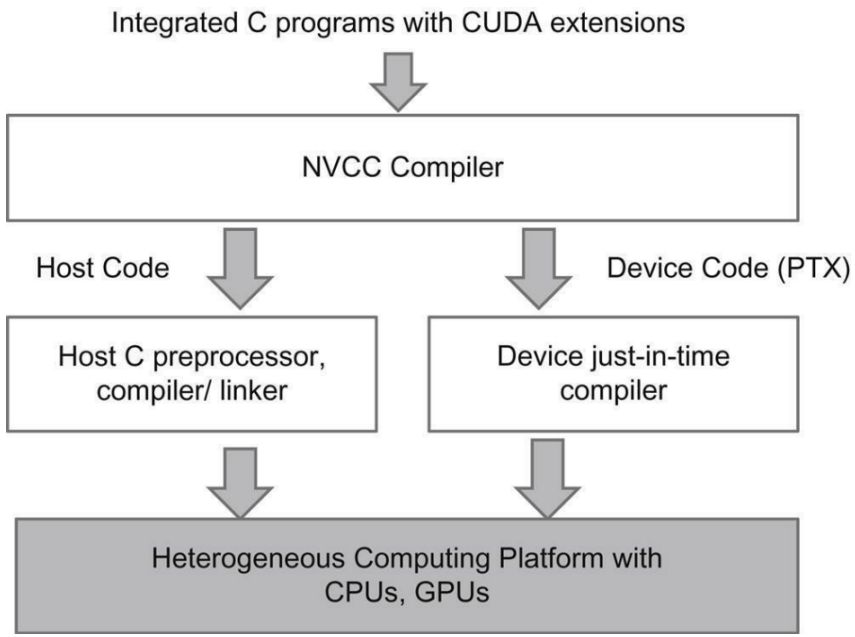
- The formula $L = 0.21R + 0.72G + 0.07B$ is computed on a **per-pixel** basis
- Pixel values are independent of each other
- Hence, we can easily parallelize this operation
- The above diagram can be interpreted as what kind of parallel model?
  - A one-level dag

CUDA compilation process



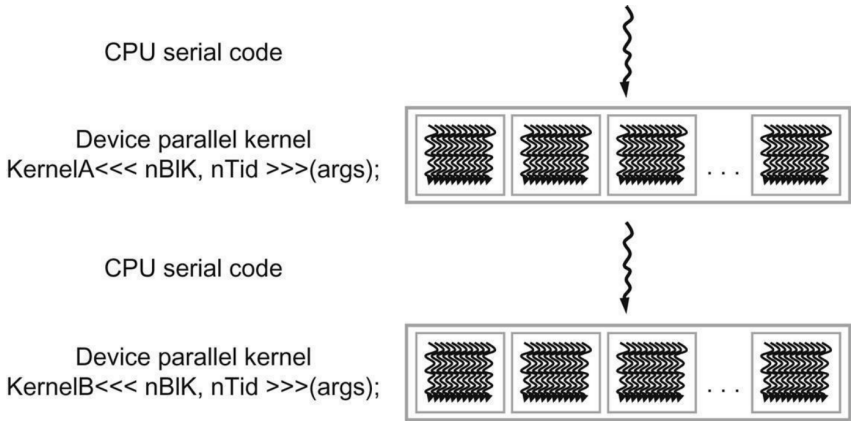
- The NVCC compiler splits code into two parts:
  - Host code (CPU)
  - Device code (GPU)
- Regular C programs are also CUDA C compatible (why?)

CUDA compilation process



- The NVCC compiler splits code into two parts:
  - Host code (CPU)
  - Device code (GPU)
- Regular C programs are also CUDA C compatible (why?)
  - They only have host code

CUDA program execution



- When a **kernel function** (device code) is launched, it is executed in parallel on the GPU
- All the **threads** generated by one kernel call are called a **grid**
  - Generally, we want many threads to maximize throughput
- In the RGB-to-grayscale example, we can have one kernel thread per pixel
- Hardware support makes generating many threads at once efficient

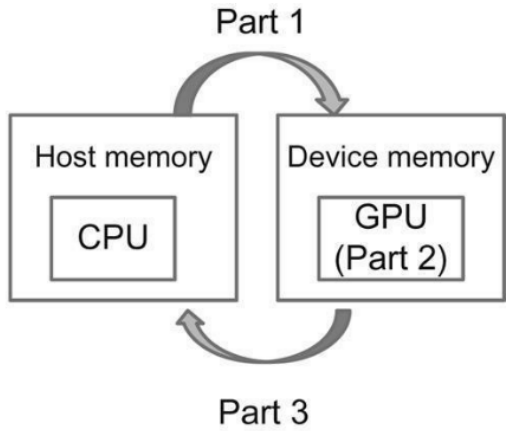
CUDA example: vector addition

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

- We will now show how to do simple vector addition using CUDA
- The above code is the sequential, basic C implementation

CUDA example: vector addition

```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
  int  size = n* sizeof(float);
  float  *d_A *d_B, *d_C;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



- Above is a sketch of the CUDA version
- We have the additional steps of **transferring the data to and from the GPU**

CUDA example: vector addition

cudaMalloc()

- Allocates object in the device global memory
- Two parameters
  - **Address of a pointer** to the allocated object
  - **Size** of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
  - **Pointer** to freed object

- CUDA has built-in functions for allocating and freeing up memory on the GPU
- Analogous to the regular **malloc** and **free** C functions

CUDA example: vector addition

cudaMemcpy()

- Memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

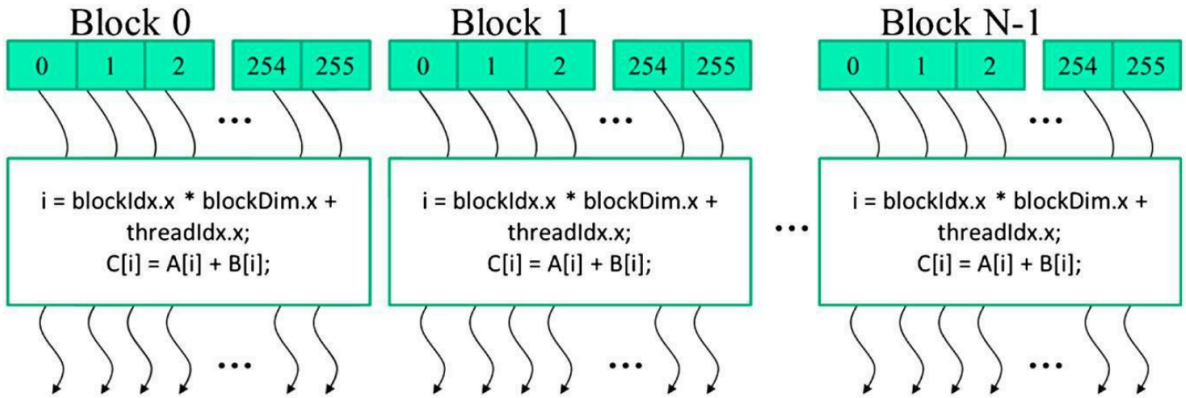
- The **cudaMemcpy** function allows us to copy data to and from the CPU and GPU
- We need to allocate the destination **beforehand**

CUDA example: vector addition

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

- Here is a more filled-in version of the CUDA code
- The built-in functions allow the CPU and GPU to talk to each other

Grid organization



- Threads are organized in a two-level hierarchy
  - N **blocks**, each with k **threads** (usually a multiple of 32)
- You access individual threads using two indices, akin to a phone number
  - Block IDs are like area codes
  - Thread IDs like the local number