A Modern Multi-Core Processor

Monday, March 30, 2020 12:00 PM

Part 1: parallel execution

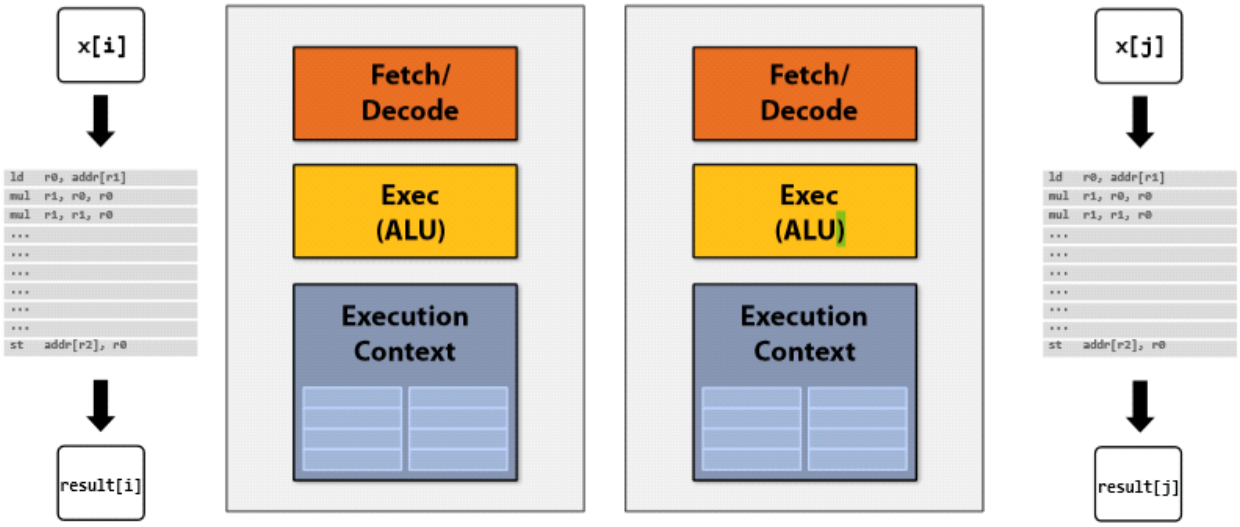
Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$ for each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

Screen clipping taken: 3/30/2020 12:47 PM



Simpler cores: each core is slower at running a single instruction stream than our original "fancy" core (e.g., 25% slower)

But there are now two cores: $2 \times 0.75 = 1.5$ (potential for speedup!)

Screen clipping taken: 3/30/2020 12:51 PM

But our program expresses no parallelism

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

This C program, compiled with gcc will run as one thread on one of the processor cores.

If each of the simpler processor cores was 25% slower than the original single complicated one, our program now runs 25% slower. :-(

Screen clipping taken: 3/30/2020 12:52 PM

Expressing parallelism using pthreads

```
typedef struct {
   int N;
   int terms;
   float* x;
   float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
       float value = x[i];
       float numer = x[i] * x[i] * x[i];
       int denom = 6;  // 3!
       int sign = -1;

       for (int j=1; j<=terms; j++)
       {
          value += sign * numer / denom
          numer *= x[i] * x[i];
          denom *= (2*j+2) * (2*j+3);
          sign *= -1;
       }

       result[i] = value;
   }
}
```

Screen clipping taken: 3/30/2020 12:52 PM

Data-parallel expression

(in Kayvon's fictitious data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

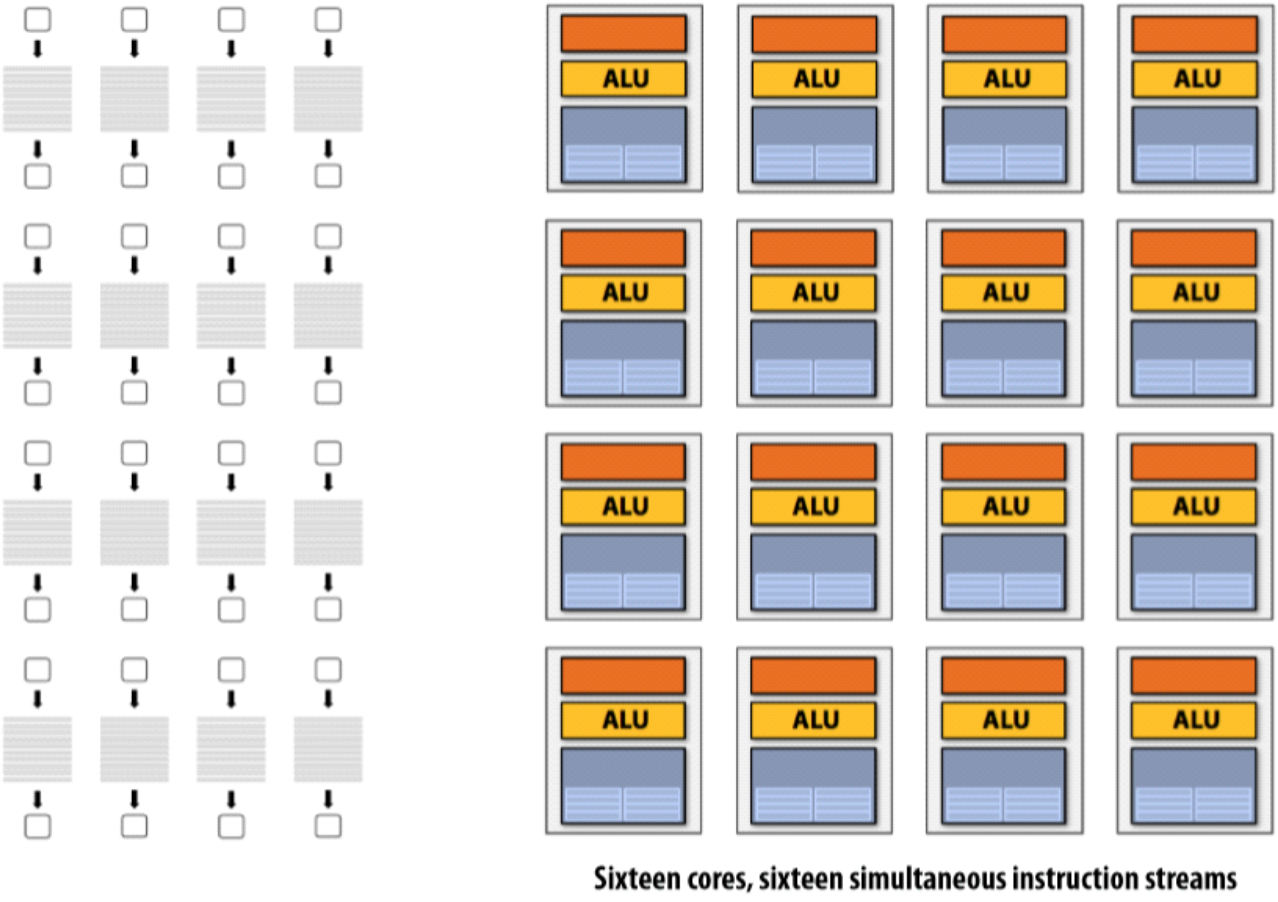
Loop iterations declared by the programmer to be independent

With this information, you could imagine how a compiler might automatically generate parallel threaded code

Screen clipping taken: 3/30/2020 12:53 PM

Fictitious = not real or true / imaginary

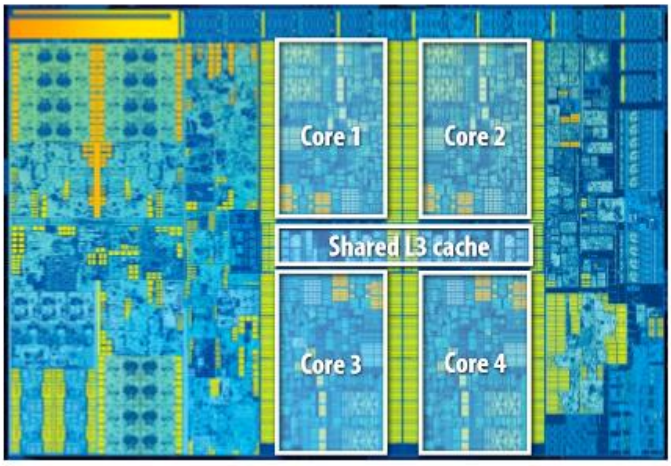
Sixteen cores: compute sixteen elements in parallel



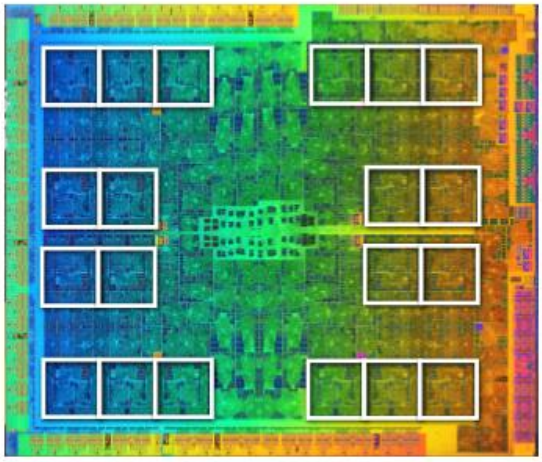
Sixteen cores, sixteen simultaneous instruction streams

Screen clipping taken: 3/30/2020 12:54 PM

Multi-core examples



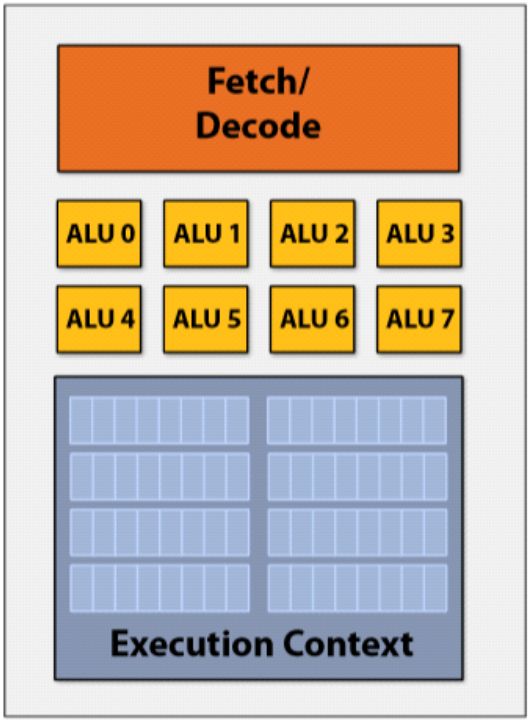
Intel "Skylake" Core i7 quad-core CPU
(2015)



NVIDIA GP104 (GTX 1080) GPU
20 replicated ("SM") cores
(2016)

Screen clipping taken: 3/30/2020 12:54 PM

Add ALUs to increase compute capability



Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs
Executed in parallel on all ALUs

Screen clipping taken: 3/30/2020 12:57 PM

Vector program (using AVX intrinsics)

Intrinsics available to C programmers

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
       __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom = _mm256_broadcast_ss(&three_fact);
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         // value += sign * numer / denom
         __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);
         value = _mm256_add_ps(value, tmp);

         numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
         denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
         sign *= -1;
      }
      _mm256_store_ps(&result[i], value);
   }
}
```

Screen clipping taken: 3/30/2020 4:00 PM

SIMD execution on modern CPUs

- SSE instructions: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)
- AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
- AVX512 instruction: 512 bit operations: 16x32 bits…

- Instructions are generated by the compiler
  - Parallelism explicitly requested by programmer using intrinsics
  - Parallelism conveyed using parallel language semantics (e.g., `forall` example)
  - Parallelism inferred by dependency analysis of loops (hard problem, even best compilers are not great on arbitrary C/C++ code)

- Terminology: "explicit SIMD": SIMD parallelization is performed at compile time
  - Can inspect program binary and see instructions (`vstoreps`, `vmulps`, etc.)

Screen clipping taken: 3/30/2020 4:16 PM

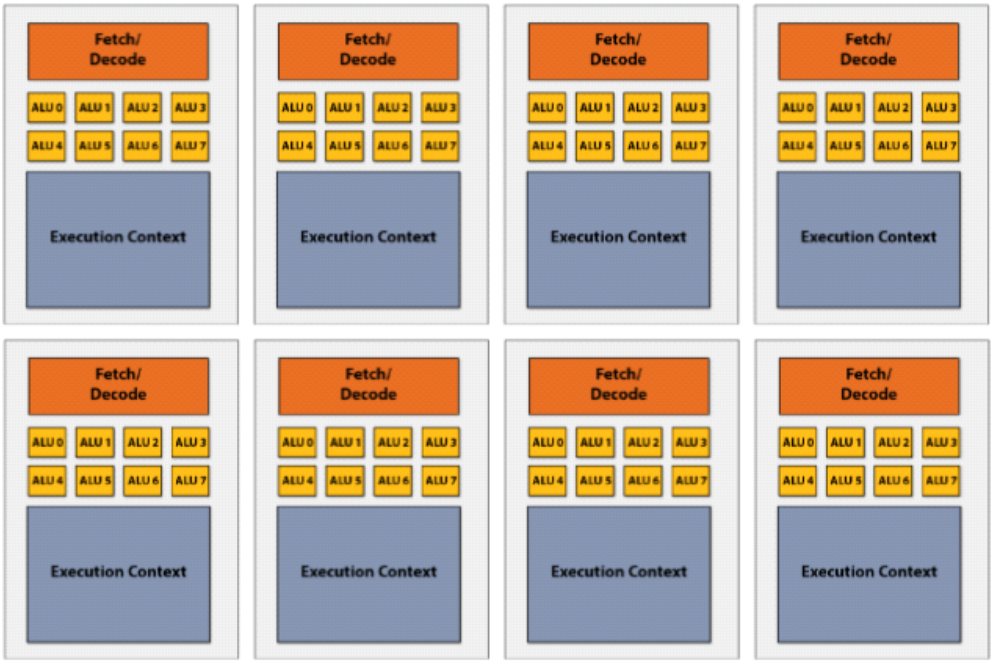
SIMD execution on many modern GPUs

- "Implicit SIMD"
  - Compiler generates a scalar binary (scalar instructions)
  - But N instances of the program are *always run* together on the processor
    ```
    execute(my_function, N)  // execute my_function N times
    ```
  - In other words, the interface to the hardware itself is data parallel
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple instances on different data on SIMD ALUs

- SIMD width of most modern GPUs ranges from 8 to 32
  - Divergence can be a big issue
    (poorly written code might execute at 1/32 the peak capability of the machine!)

Screen clipping taken: 3/30/2020 4:17 PM

Example: eight-core Intel Xeon E5-1660 v4



8 cores
8 SIMD ALUs per core (AVX2 instructions)

490 GFLOPs (@3.2 GHz)
(140 Watts)

Screen clipping taken: 3/30/2020 4:18 PM

Example: NVIDIA GTX 1080

20 cores ("SMs")

128 SIMD ALUs per core (@1.6 GHz) = 8.1 TFLOPs (180 Watts)

Stanford CS149, Fall 201

Screen clipping taken: 3/30/2020 4:18 PM

Summary: parallel execution

- **Several forms of parallel execution in modern processors**
  - Multi-core: use multiple processing cores
    - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
    - Software decides when to create threads (e.g., via pthreads API)
  - SIMD: use multiple ALUs controlled by same instruction stream (within a core)
    - Efficient design for data-parallel workloads: control amortized over many ALUs
    - Vectorization can be done by compiler (explicit SIMD) or at runtime by hardware
    - [Lack of] dependencies is known prior to execution (usually declared by programmer, but can be inferred by loop analysis by advanced compiler)
  - Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
    - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)

    Not addressed further in this class. That's for a proper computer architecture course.

Screen clipping taken: 3/30/2020 5:08 PM