2. Introduction to Heterogeneous Computing

Heterogeneous Programming Models

Sylabus

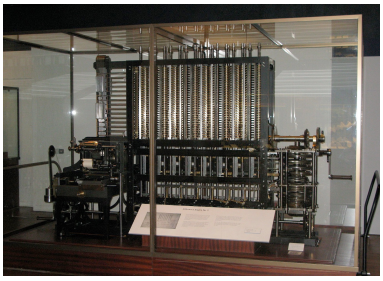
- A brief and *personal* history of computers ...
- From sequential to parallel execution
- A few notes on Multiprocessors & Amdahl's law

Goals

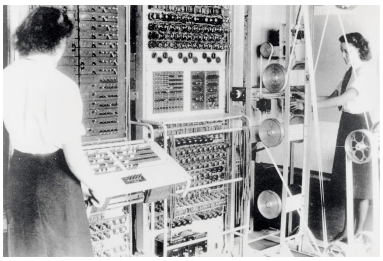
- Understand why heterogeneous computers are mostly prevalent today
- Know the differences between Von Neumann and Harvard Computer models
- Knows the basics of CPU, GPU, and multiprocessor microarchitecture

A bit of history

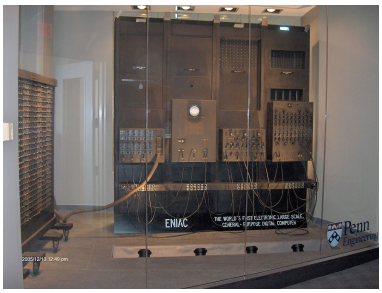
At the beginning … specialized hardware



Differential Engine
Babbage 1822
Navigation chart



Colossus computer
British codebreakers 1943
Cryptoanalysis



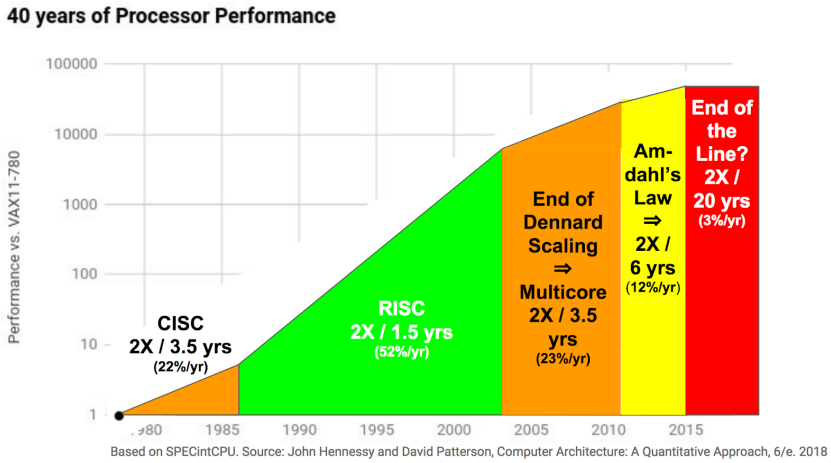
Electronic Numerical Integrator and Computer, ENIAC
University of Pennsylvania, 1945
Ballistic

Then, in the sixties: IBM 360

- A **general-purpose computer** from IBM that fused several specialized architectures: IBM 701 (scientific computation), IBM 650 (business and scientific computation), or IBM 1401 (business)

The rise of general-purpose processors

- Semiconductor technologies and computer architecture fueled processor performance



fuente:http://iscaconf.org/isca2018/docs/HennessyPattersonTuringLectureISCA4June2018.pdf

Technology improvements: Moore's law

Andy Grove, Robert Noyce, and Gordon Moore

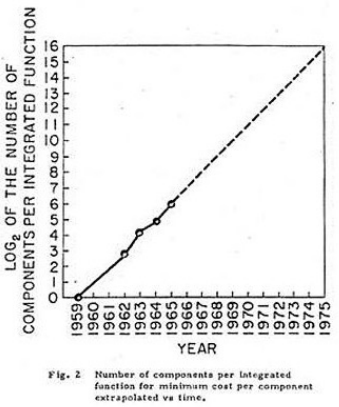
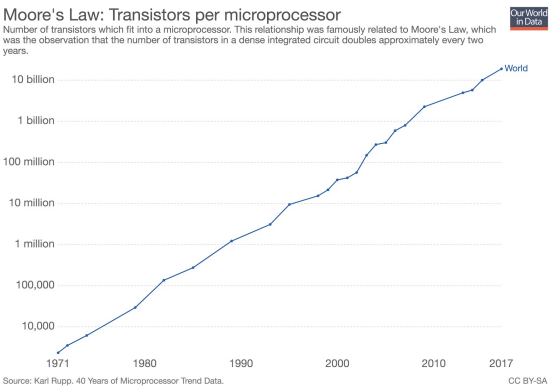
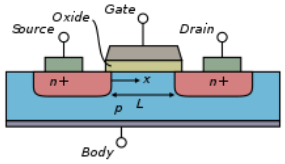


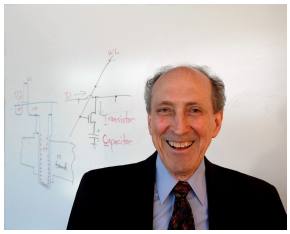
Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.



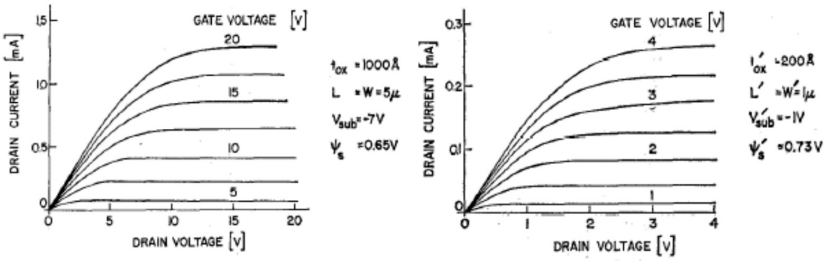
Technology improvements: Dennard's scaling



n-type MOSFET transistor



Robert H. Dennard



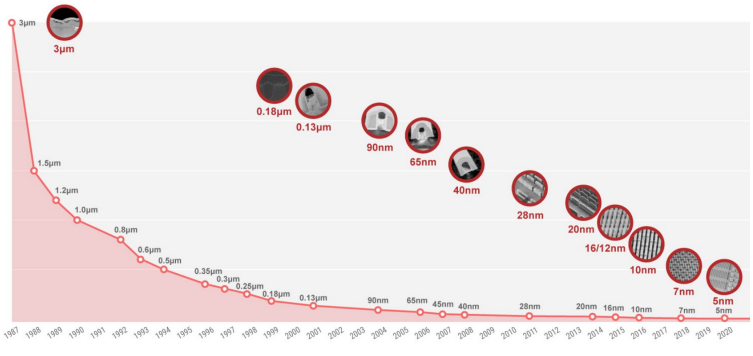
Scaling factor: α

The triple play:

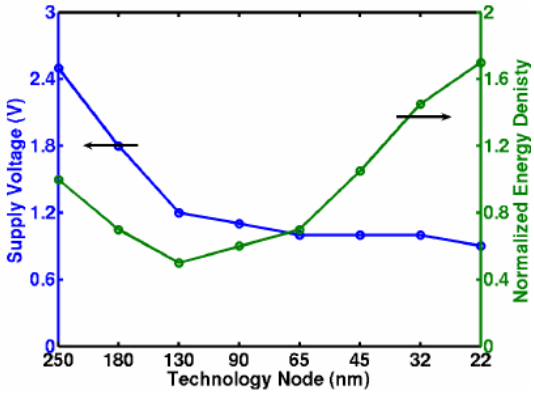
• Get more gates,	$1/L^2$	$1/\alpha^2$
• Gates get faster,	CV/i	α
• Energy per switch	CV^2	α^3

Source: M. Horowitz ISSCC 2014 Keynote, Computing's energy problem (and what we can do about it)

Now, the end of voltage scaling



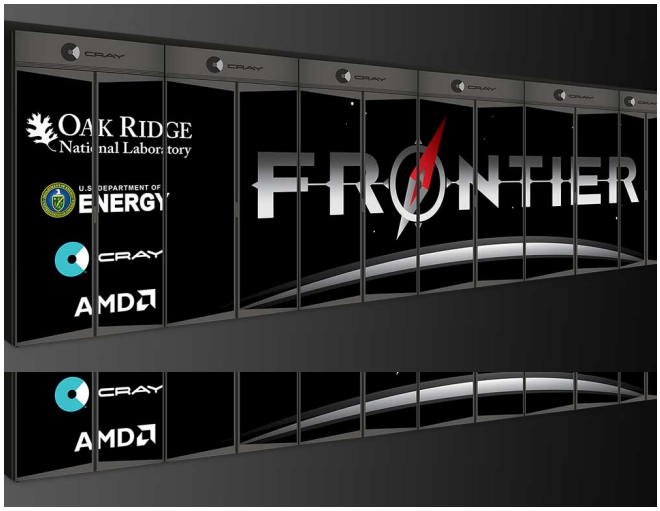
Source: https://www.tsmc.com/english/dedicatedFoundry/technology/logic.htm#l_5nm_technology



Source: Assessing the performance limits of parallelized near-threshold computing

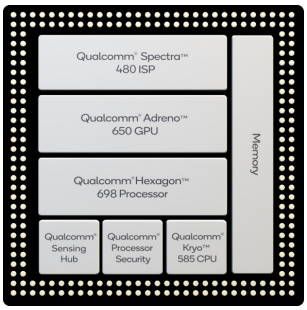
The return of specialized hardware

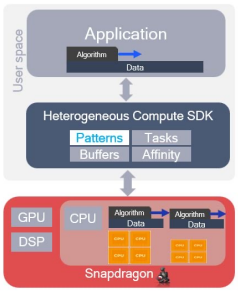
Currently, Heterogeneous system: from supercomputers …



- HPE Frontier Summit: 8,730,112 AMD Epyc CPUs + AMD GPU
- 1,1 Exaflops
- “Heterogeneous Computing Model” HIP
- Power: 21,100.00 kW
- Nº 1 TOP500 july 2022

Heterogeneous systems: … to mobile devices





- Qualcomm Snapdragon 865
  - Heterogeneous Kyro CPUs
  - Hexagon DSP
  - Adreno GPU
  - Spectra ISP
  - Sensing Hub (low power AI, audio voice, …)
  - Security Processor
- Heterogeneous Compute SDK programming model

From Sequential to Heterogeneous Execution

These slides are based on “Lecture 2: A Modern Multi-Core Processor”, Parallel Computing, Stanford CS149, Fall 2020, http://cs149.stanford.edu/fall20/

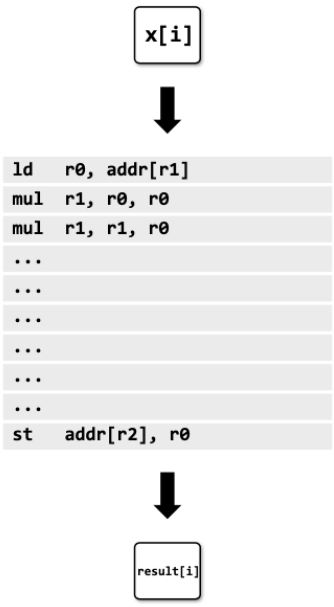
Example Program

- Compute sin(x) using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \ldots$

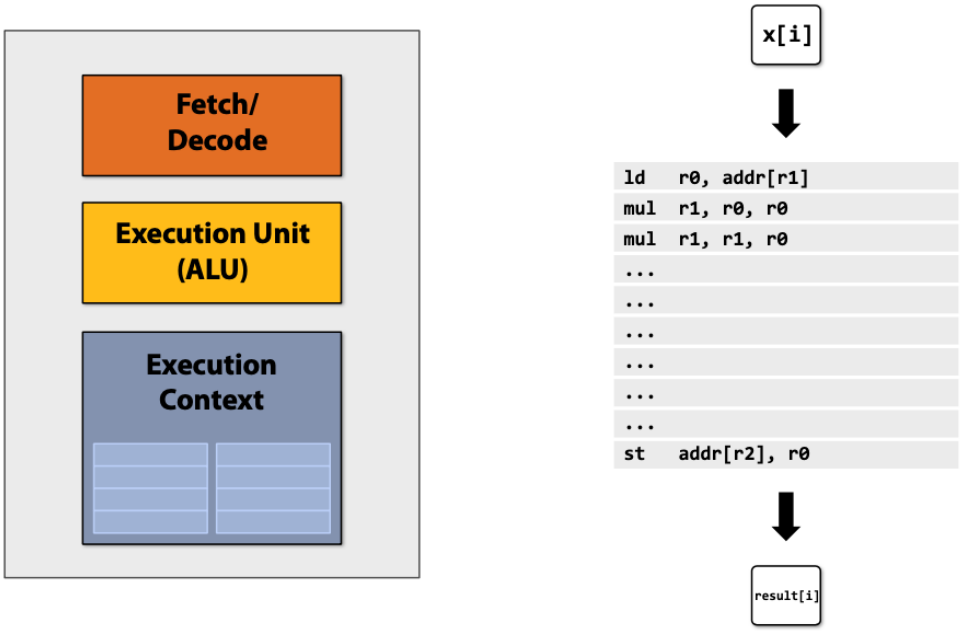
```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      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;
   }
}
```


Compile program

```
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;
   }
}
```

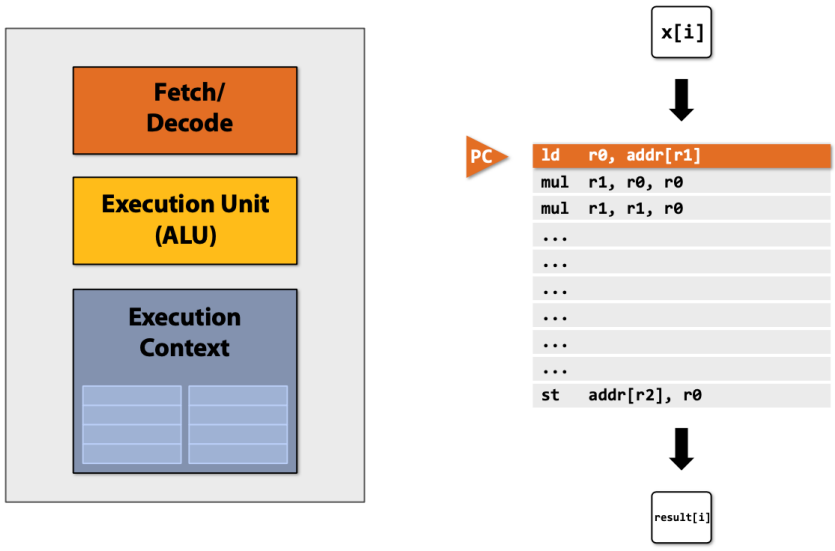


Execute program



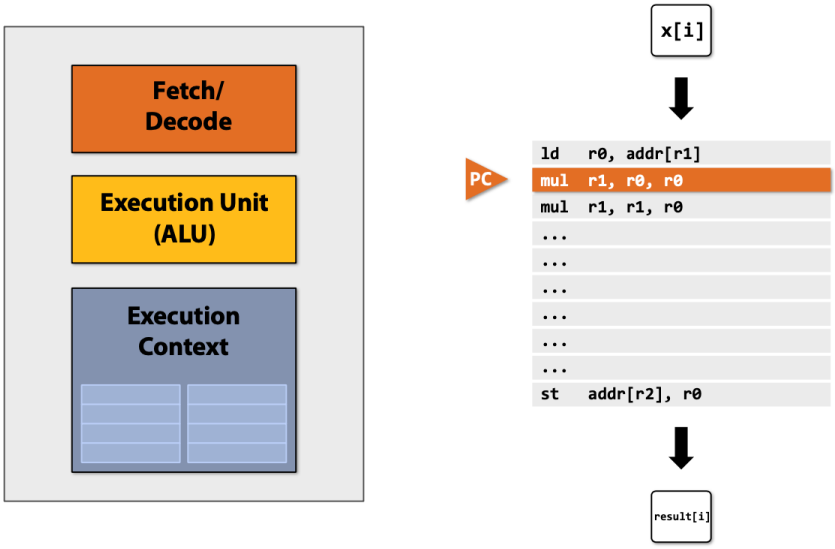
Execute Program

- A very simple processor executes one instruction per cycle



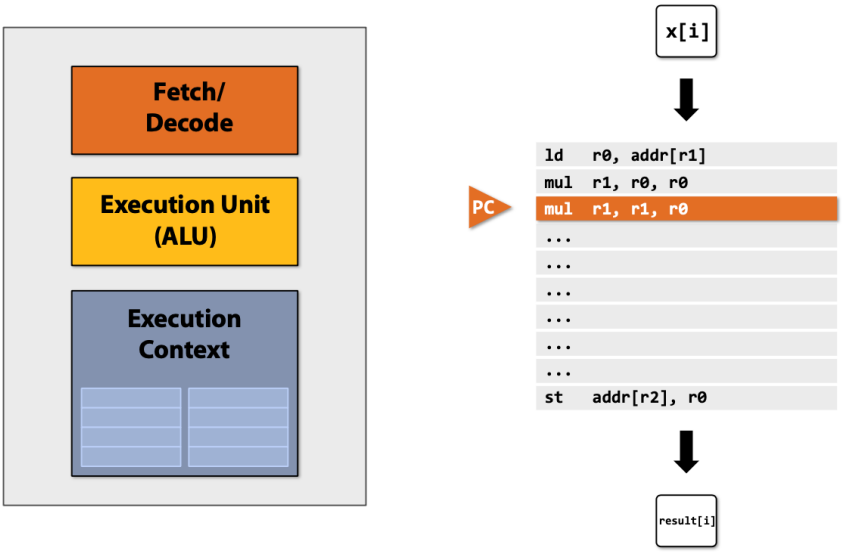
Execute Program

- A very simple processor executes one instruction per cycle



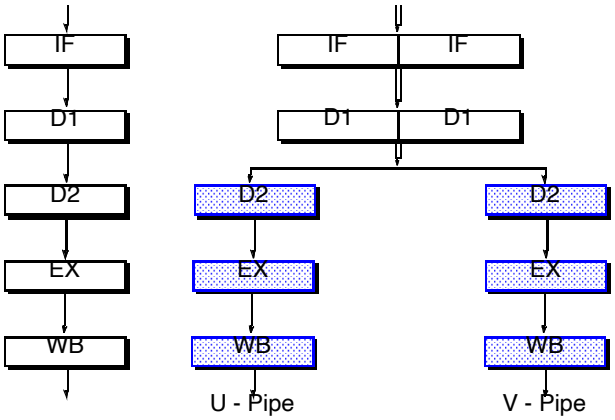
Execute Program

- A very simple processor executes one instruction per cycle



¿Can we execute multiple instructions at the same time?

- Yes, superscalar processor (execute multiple instructions at once)



Pipelined (486) Superscalar (Pentium)

source: Miko Lipasti

Problem: Track Instruction Dependencies and guarantee ordering

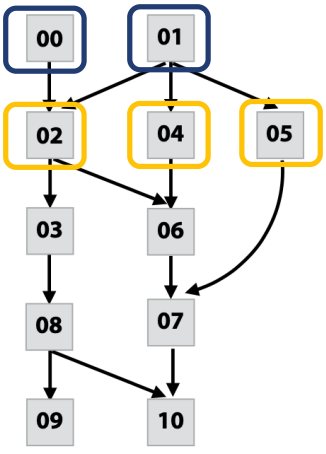
In-order vs Out-of-Order execution

- In-order completion is always required, but execution is not …
- Key: Instruction Level Parallelism (ILP)

Program (sequence of instructions)

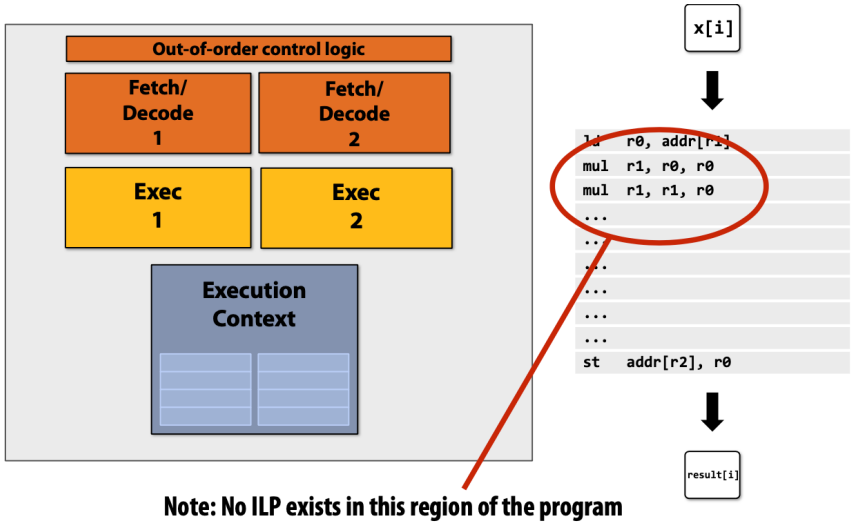
PC	Instruction
00	a = 2
01	b = 4
02	tmp2 = a + b // 6
03	tmp3 = tmp2 + a // 8
04	tmp4 = b + b // 8
05	tmp5 = b * b // 16
06	tmp6 = tmp2 + tmp4 // 14
07	tmp7 = tmp5 + tmp6 // 30
08	if (tmp3 > 7)
09	print tmp3
	else
10	print tmp7

Instruction dependency graph



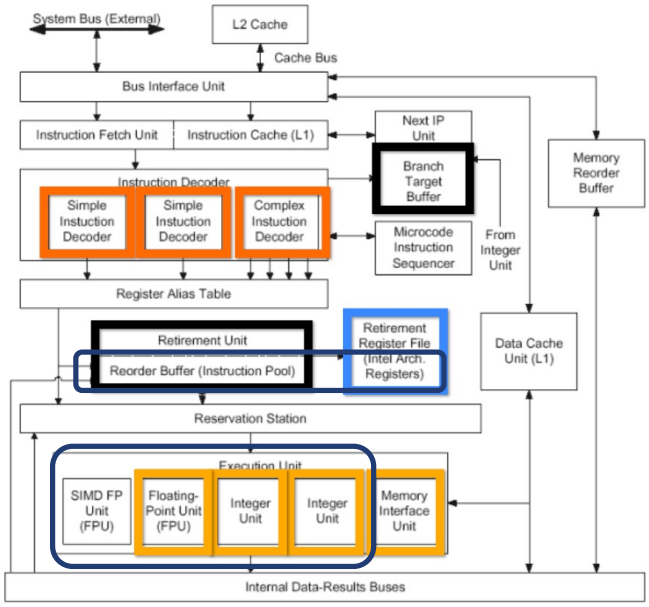
Out-of-Order Superscalar Processor

- Decode and execute two instructions per cycle (if possible)



Instruction Level Parallelism (ILP) refers to the number of independent instructions that can be executed in parallel

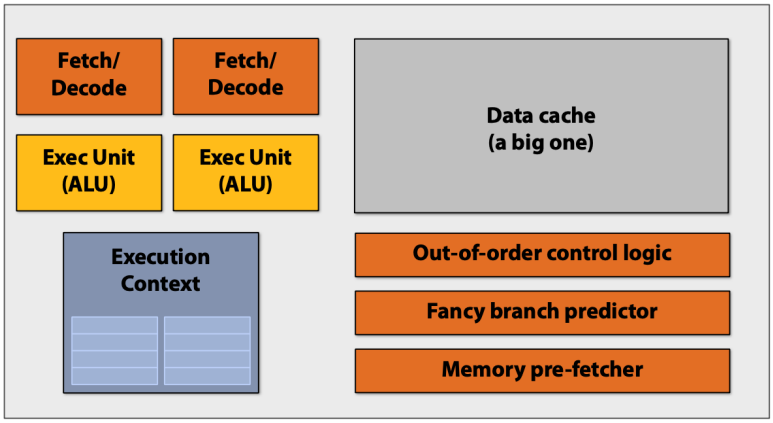
Aside: Pentium IV

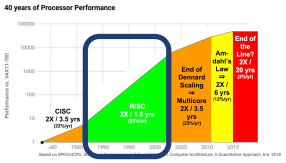


Processing Units:
Register File (TEMP)
ALUS
...
The rest is control and memory ...

Processor Arch: pre-multi core era

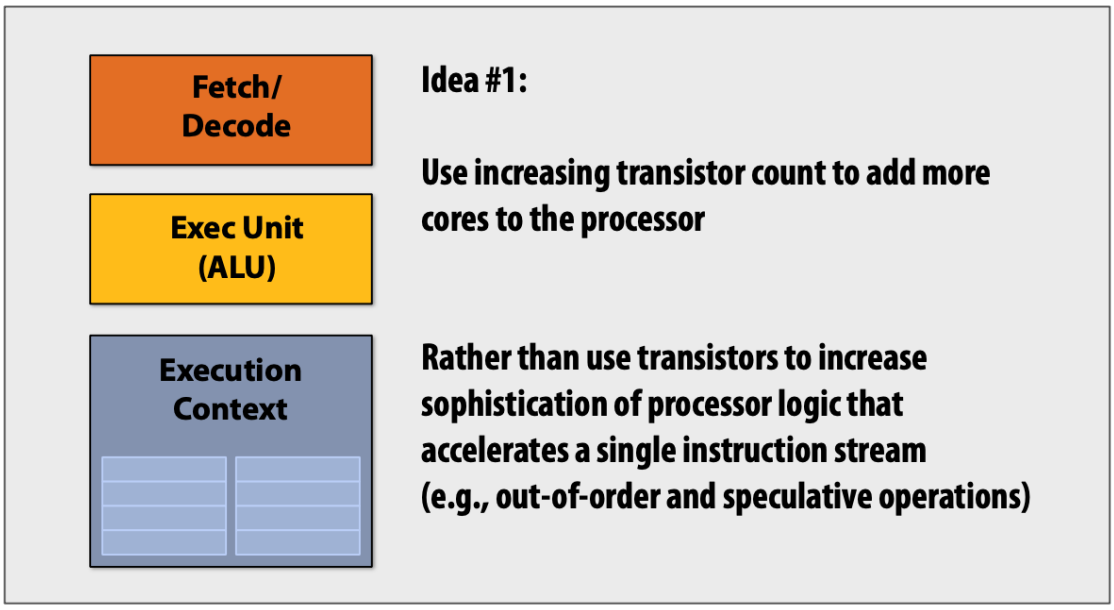
- **Majority** of chip **transistors** used to speed-up single-instruction stream execution (**control**)



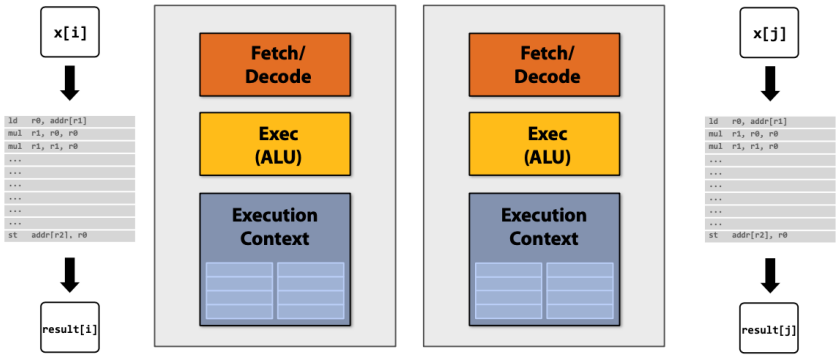


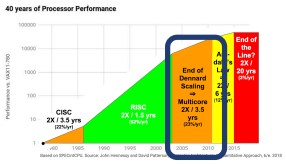
- More transistors = larger cache memories, Out-of-Order execution, branch prediction, ...
- Also: more transistors ➜ smaller transistors ➜ faster transistors (Dennard's Scaling)

Processor Arch: multi-core era



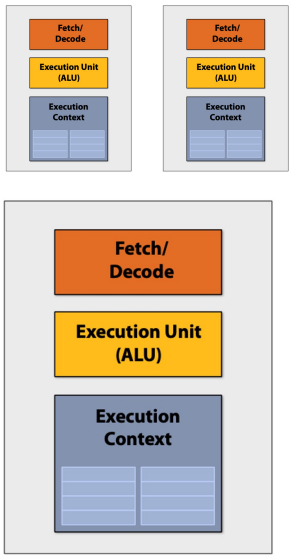
Two cores: compute two elements in parallel



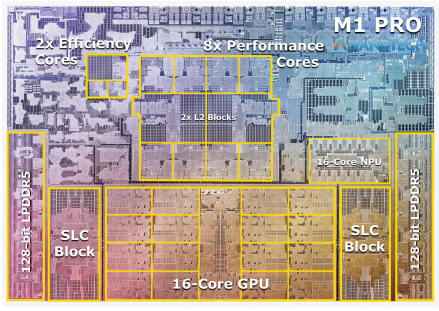


- Simpler cores: each core is slower at running a single instruction stream than our original "fancy" core (e.g., 25% slower)
- But there are now two cores: 2 x 0.75 = 1.5 (potential for speedup!)

Several cores: same ISA, different µArch



- All programs compiled with the same ISA
- Move to fast and power-hungry cores when required



Apple M1 PRO (2021)

But our sine program does not express 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 ☹

Expressing parallelism using C++ std::thread

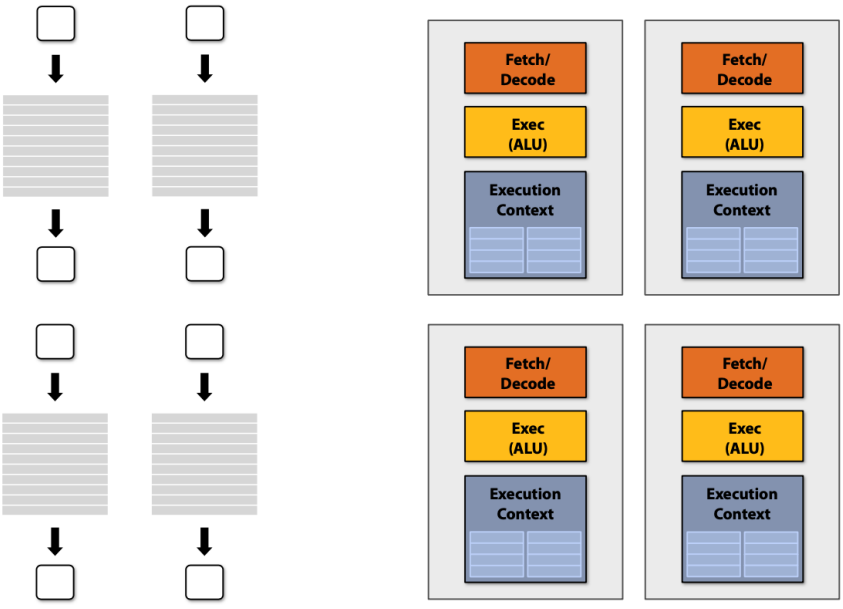
```
void sinx(int BEGIN, int END, int terms, float* x, float* result)
{
    for (int i=BEGIN; i<END; i++)
    {
         float value = x[i];
         float numer = x[i] * x[i] * x[i];  int denom = 6; // 3!
         int sign = -1;
         for (int j=1; j<=terms; j++)
         {
              value += sign * numer / denom  numer *= x[i] * x[i];
              denom *= (2*j+2) * (2*j+3);  sign *= -1;
        }
        result[i] = value;
    }
}
```

```
void parallel_sinx(int N, int terms, float* x, float* result)
{
     std::thread t0(sinx, 0, N – N/2, terms, std::ref(args.x), std::ref(args.result)); // create and
     launch thread
     sinx(N – N/2, N, terms, x + args.N, result + N/2); // do work
     t0.join(); // wait the completion of the thread
}
```

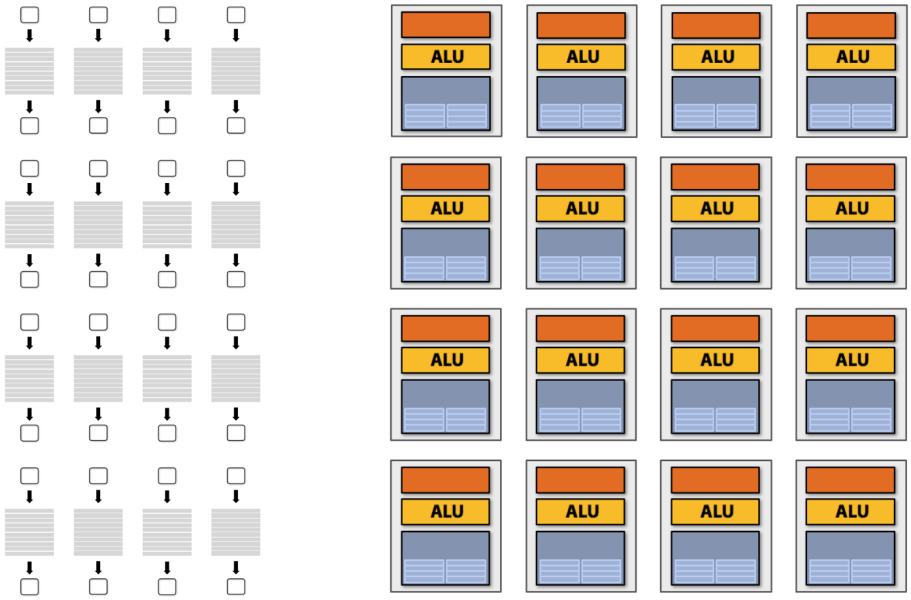
Data-parallel expression (invented language)

```
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;
   }
}
```


Four cores: compute four elements in parallel

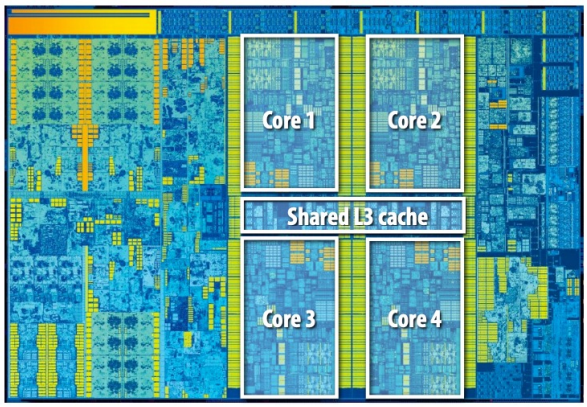


Sixteen cores: compute 16 elements in parallel

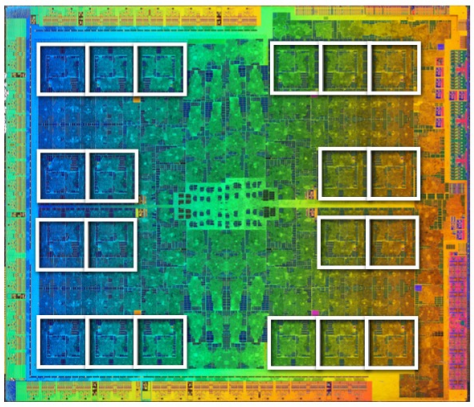


Sixteen cores, sixteen simultaneous instruction streams

Multi-core layout examples

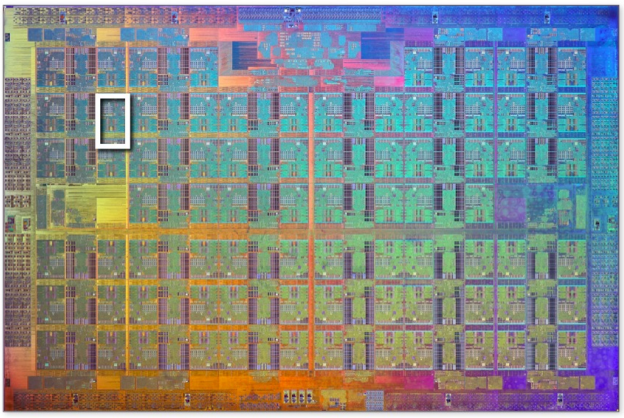


Intel "Skylake" Core i7 quad-core CPU (2015)

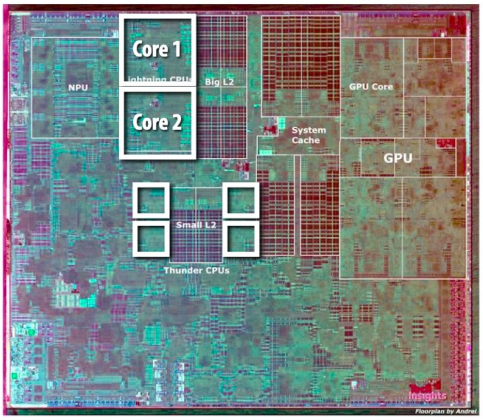


NVIDIA GP104 (GTX 1080) GPU 20 replicated ("SM") cores (2016)

More multi-core layout examples



Intel Xeon Phi "Knights Corner" 72-core CPU (2016)



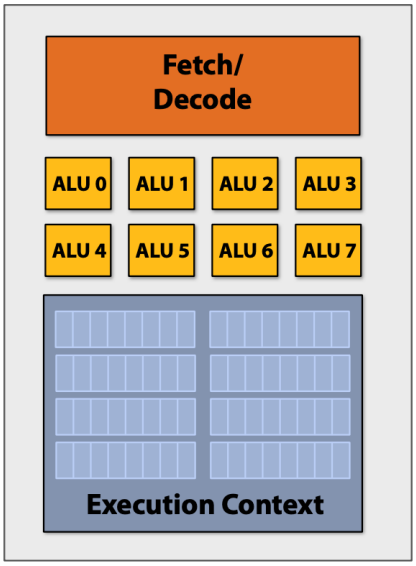
Apple A13: two "big" cores + four "small" cores (2019)

Source A13: Anandtech/ TechInsights Inc.

Data-parallel expression (invented language)

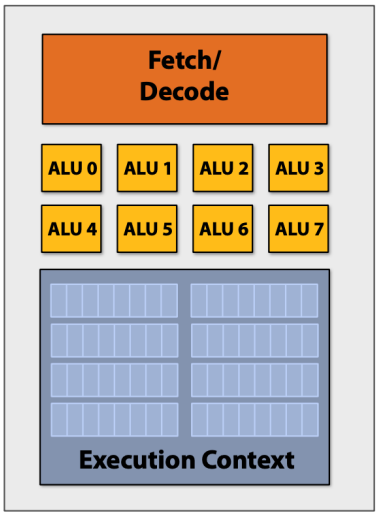
```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      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;
   }
}
```

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

Add ALUs to increase compute capability




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

- Recall original compiled program:
  - Instruction stream processes one array element at a time using scalar instructions on scalar register; e.g., 32-bit floats

Scalar program

```
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;
   }
}
```

- Original compiled program:

  Processes one array element using scalar instructions on scalar registers; e.g., 32-bit floats
- Single Instruction, Single Data

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```


Vector program (using AVX instrinsics)

```
#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);
   }
}
```

Intrinsics available to C programmers

Operate on vectors of eight 32-bit values

Vector program (using AVX instrinsics)

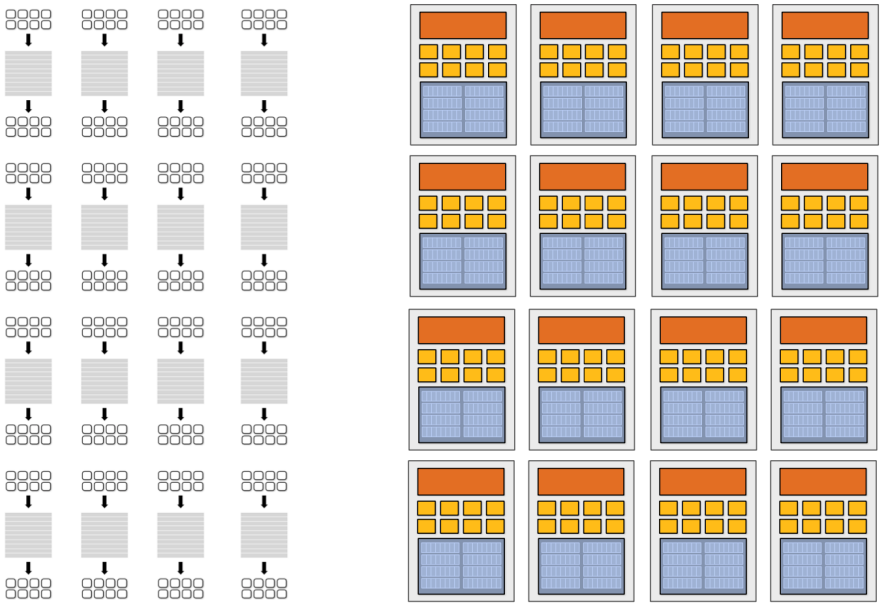
```
#include <immintrin.h>

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);
   }
}
```

```
vloadps  xmm0, addr[r1]
vmulps   xmm1, xmm0, xmm0
vmulps   xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps  addr[xmm2], xmm0
```

- Compiled program:

  Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

16 SIMD cores: 128 elements in parallel

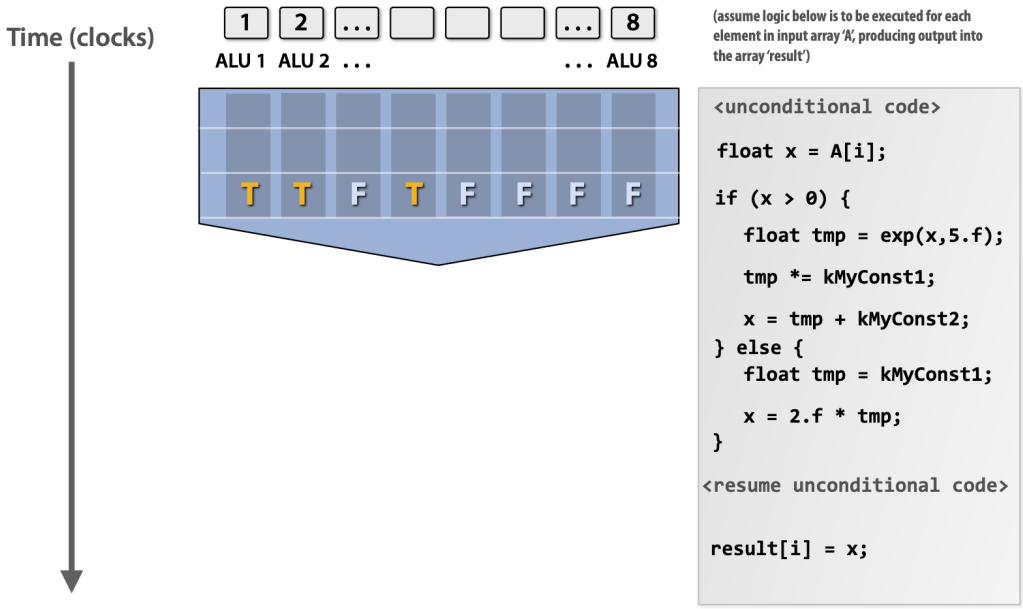


16 cores, 128 ALUs, 16 simultaneous instruction streams

Data-parallel expression (invented 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;
   }
}
```


What about conditional execution?



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., C++11 std::thread
  - 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)
  - SIMD: 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)

References

- Onur Mutlu, Digital Design and Computer Architecture, Spring 2020, ETHZ, https://safari.ethz.ch/digitaltechnik/spring2020/doku.php?id=start
- Nicholas Weaver, Great Ideas in Computer Architecture, 2019 UC Berkeley, https://inst.eecs.berkeley.edu/~cs61c/sp19/
- Kayvon Fatahalian & Kunle Olukotun, Parallel Computing, Stanford CS149, Fall 2020, http://cs149.stanford.edu/fall20/