Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24

"Εισαγωγή"

Παναγιώτης Χατζηδούκας

Course description

https://hsis.upatras.gr/?page_id=133

- Μεταγλωττιστές
- Τεχνικές μεταγλωττιστών για βελτιστοποίηση ταχύτητας ενσωματωμένων υπολογιστών
- Βελτιστοποίηση απόδοσης λογισμικού
- Μετασχηματισμοί βρόχων
- Εξάρτηση δεδομένων
- Τεχνικές διαχείρισης μνήμης
- Επιτάχυνση χρόνου εκτέλεσης
- Παραλληλισμός επιπέδου εντολών
- Διανυσματικοί υπολογιστές
- Εργαλεία μέτρησης απόδοσης

Embedded systems programming

- Development of software for embedded systems, which are computer systems that are integrated into other devices or products.
- These systems
  - are designed to perform specific functions
  - often have limited resources
    - including memory, processing power, and energy.

- This software is typically written in low-level languages such as C and assembly language and is optimized for the specific hardware and application requirements of the system

Importance

- Embedded systems are used in a wide range of products
  - e.g., automobiles, medical devices, consumer electronics, industrial equipment.
- These systems must be reliable, efficient, and cost-effective
- Programming plays a critical role in achieving these goals.
- By optimizing software for the specific hardware and application requirements of the system, developers can
  - improve system performance
  - reduce energy consumption
  - minimize costs.

Programming Languages

- Some of the most commonly used programming languages for embedded systems programming include:
  - C: C Programming is a widely used programming language for embedded systems programming. It provides direct access to system resources and is well-suited for systems with limited resources.
  - Assembly language: Assembly language is a low-level programming language that provides direct access to system resources. It is often used for systems with very limited resources or for performance-critical applications.
  - C++: C++ is a high-level programming language that is often used for embedded systems programming. It provides object-oriented programming features and can be used to develop complex systems.

Best Practices

- Writing Efficient Code
  - Writing efficient code is critical for embedded systems, as these systems often have limited resources. Developers should focus on writing code that is optimized for the specific hardware and application requirements of the system and should avoid using unnecessary resources.
- Debugging Techniques
  - Debugging techniques are critical for identifying and fixing errors in software. Developers should use a range of debugging techniques, including stepping through code, setting breakpoints, and examining variables and memory.
- Testing and Validation
  - Testing and validation are critical for ensuring that software is reliable and efficient. Developers should use a range of testing and validation techniques, including unit testing, integration testing, and system testing.

Course Schedule (tentative)

1. Introduction: architectures, performance	02.10 – 1
2. Memory hierarchy and Roofline mode,	09.10 –
3. Debugging, Profiling	16.10 –
4. Compiler and code optimizations I	23.10 – 2
5. Compiler and code optimizations II	30.10 – 3
6. Vectorization	06.11 – 4
7. BLAS for Embedded Optimization	13.11 – 5
8. OpenMP basics	20.11 – 6
9. OpenMP optimization	27.11 – 7
10. OpenMP tasks	04.12 – 8
11. Recap	11.12 – 9
	18.12 – 10
	25.12
	01.01
	08.01 – 11

Use cases

- Matrix operations (AI related)
- Image processing

Evaluation (TBD)

- 2-3 programming assignments (70%)
- Written or oral exam (30%)

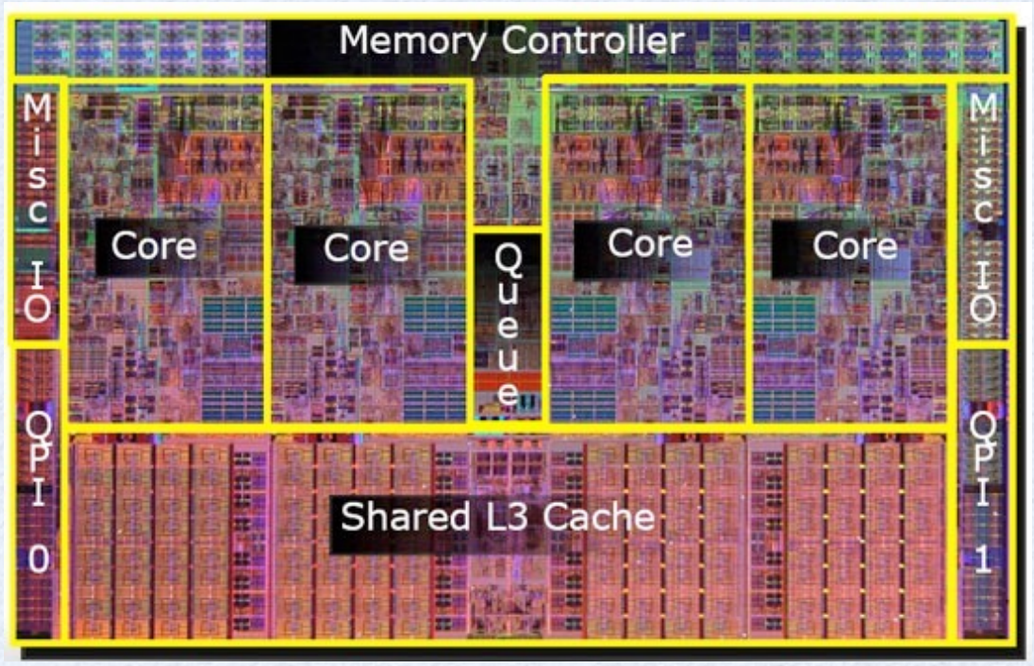
Class Website

https://eclass.upatras.gr/courses/CEID1418/

Computer layout (in a nutshell)

- CPU
  - does the computations
  - contains **multiple cores** (usually)
    - each core works mostly independently,
      copy of a single core with global coordination
  - contains several levels of caches to
    speed up reading/writing to memory
    (very relevant for high performance computing)

Intel Core i7 CPU

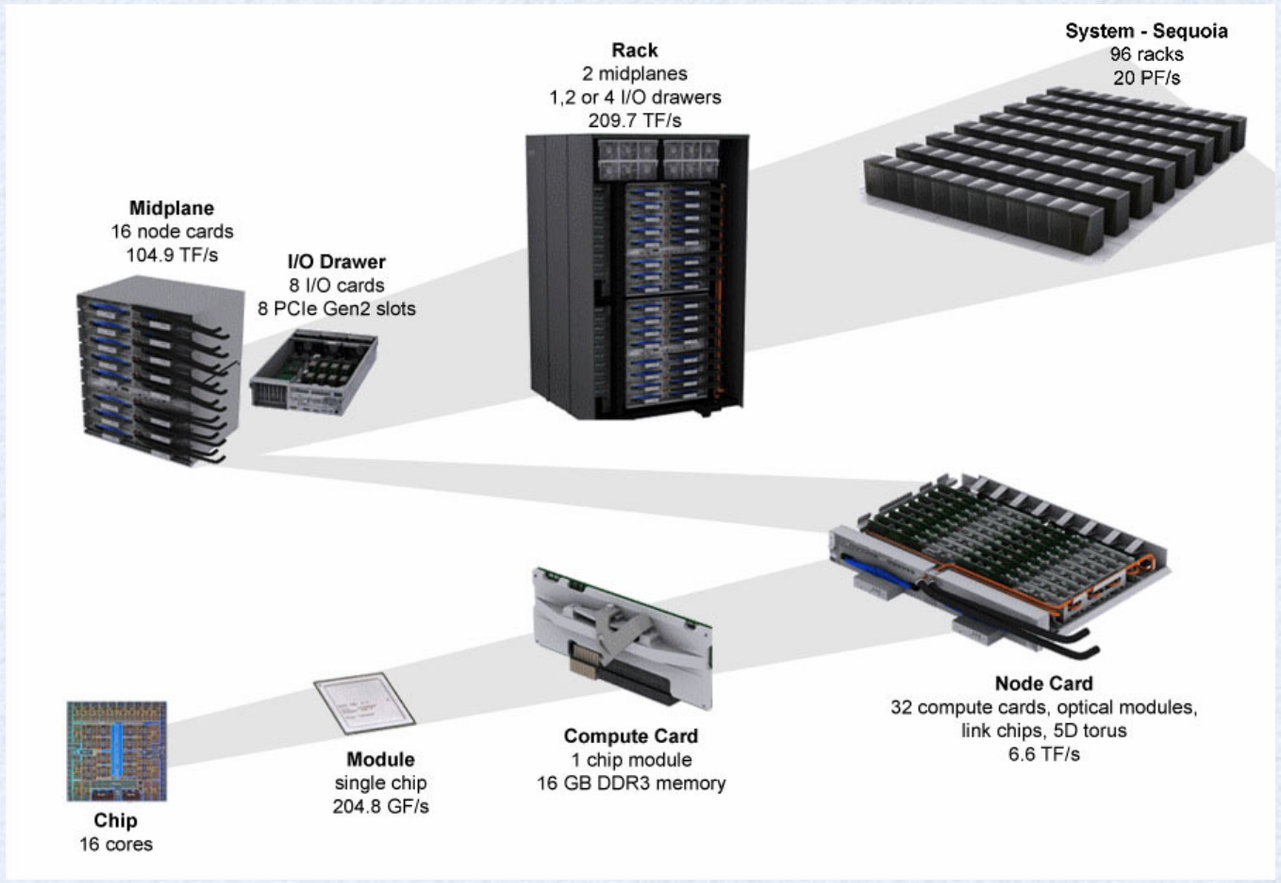


picture source: legitreviews

- Memory
  - stores data for computations
  - **shared** among the cores of the CPU (or multiple CPUs in a compute node)
- Network: connect compute nodes and connect to outer world
- Input/Output: displays, hard-drives, etc

Massively Parallel Computing

Sequoia IBM BlueGene/Q supercomputer (at Lawrence Livermore National Laboratory)

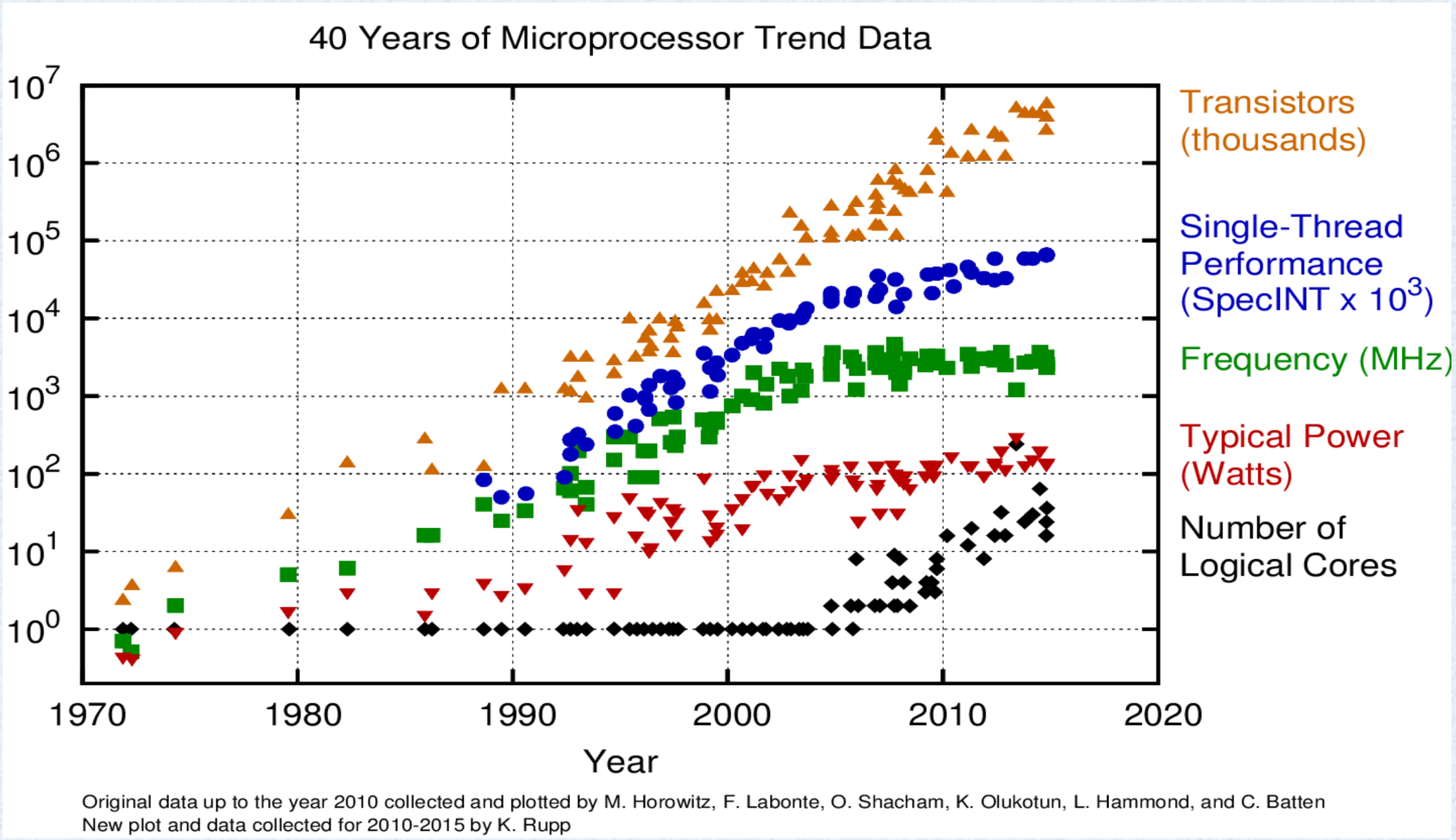


source:computing.llnl.gov

Components of a Supercomputer (roughly)

- Processors (CPUs) <= *note that those already contain multiple cores*
- Compute Node: collection of CPUs with a shared memory
  - nodes may also have “accelerators” like graphical processing units (GPU)
- Cluster: collection of nodes connected with a (very fast) network

Revolution in Processors



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

The TOP500 Project

- Listing the 500 most powerful computers in the world
- Yardstick: performance (Rmax) of Linpack
  - Solve Ax=b, dense problem, matrix is random
  - Dominated by dense matrix-matrix multiply
- Updated twice a year:
  - ISC'xy in June in Germany
  - SCxy in November in the U.S.
- TOP500 web site at: www.top500.org

TOP500 - June 2020

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	

Units of Measure

- High Performance Computing (HPC) units are:
  - **Flop:** floating point operation, usually double precision unless noted
  - **Flop/s:** floating point operations per second
  - **Bytes:** size of data (a double precision floating point number is 8 bytes)

- Typical sizes are millions, billions, trillions…
  - Mega MFlop/s = 10^6 flop/sec MByte = 2^20 ~ 10^6 bytes
  - Giga GFlop/s = 10^9 flop/sec GByte = 2^30 ~ 10^9 bytes
  - **Tera TFlop/s = 10^12 flop/sec TByte = 2^40 ~ 10^12 bytes**
  - **Peta PFlop/s = 10^15 flop/sec PByte = 2^50 ~ 10^15 bytes**
  - **Exa EFlop/s = 10^18 flop/sec EByte = 2^60 ~ 10^18 bytes**
  - Zetta ZFlop/s = 10^21 flop/sec ZByte = 2^70 ~ 10^21 bytes

- Current fastest (public) machine: ~ 537 PFlop/s, 7.6M cores

How Rpeak is computed

- Rpeak = Nominal Peak Performance (**PP**)
- **PP** [Flop/s] = **f** [Hz = cycles/s] x **c** [Flop/cycle] x **v** [-] x **n** [-]
  - **f** : core frequency in CPU cycles per second
  - **c** : how many Flops per cycle
  - **v** : SIMD width in number of doubles (or floats)
  - **n** : # cores

These features can be found at the hardware specifications

- Example: IBM BGQ chip (one compute node)
  - **f** = 1.6 GHz, **c** = 2 (supports FMA), **v** = 4, **n** = 16
  - **PP** = 1.6 * 2 * 4 * 16 GFlop/s = 204.8 GFlop/s

FMA (fused multiply-add): a*b+c in one step

- BGQ Rack (1024 nodes): 1024*204.8 = 209715.2 GFlop/s = 209.7 TFlop/s
- IBM Sequoia @ LLNL (96 racks): 96 * 209.7 TFlop/s = 20.13 PFlop/s

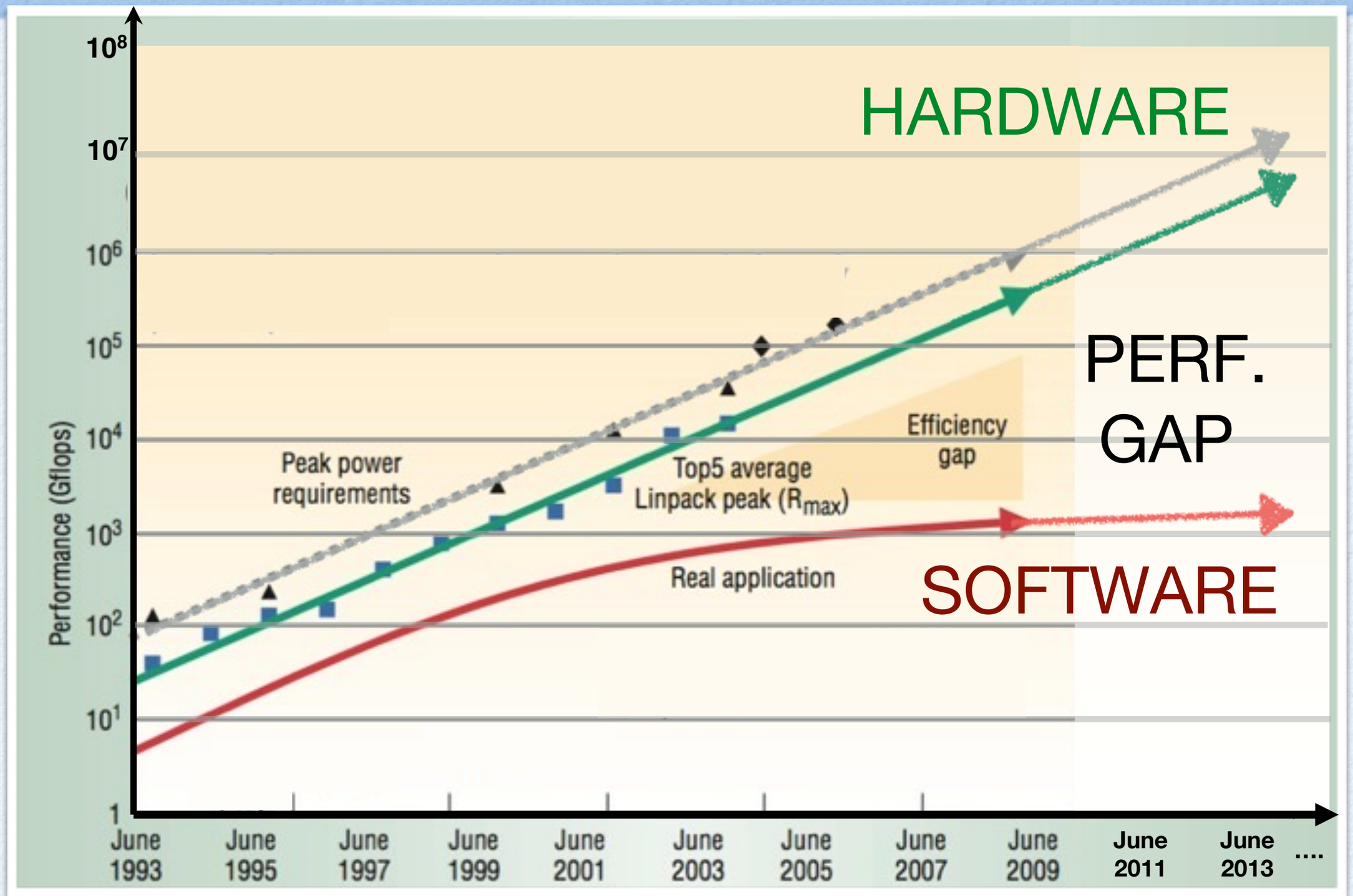
How Rmax is computed

- Application Performance
  - How many FP operations the application performs
  - Execution time (in seconds)
  - Fraction of the peak = Attained/Nominal performance

- In many cases, FP operations can be replaced with INT operations, interactions, transactions, etc. (per second)

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)	Rmax/Rpeak (%)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371	74.1%
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808	61.7%
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209	64.9%
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890	85.3%

Performance GAP

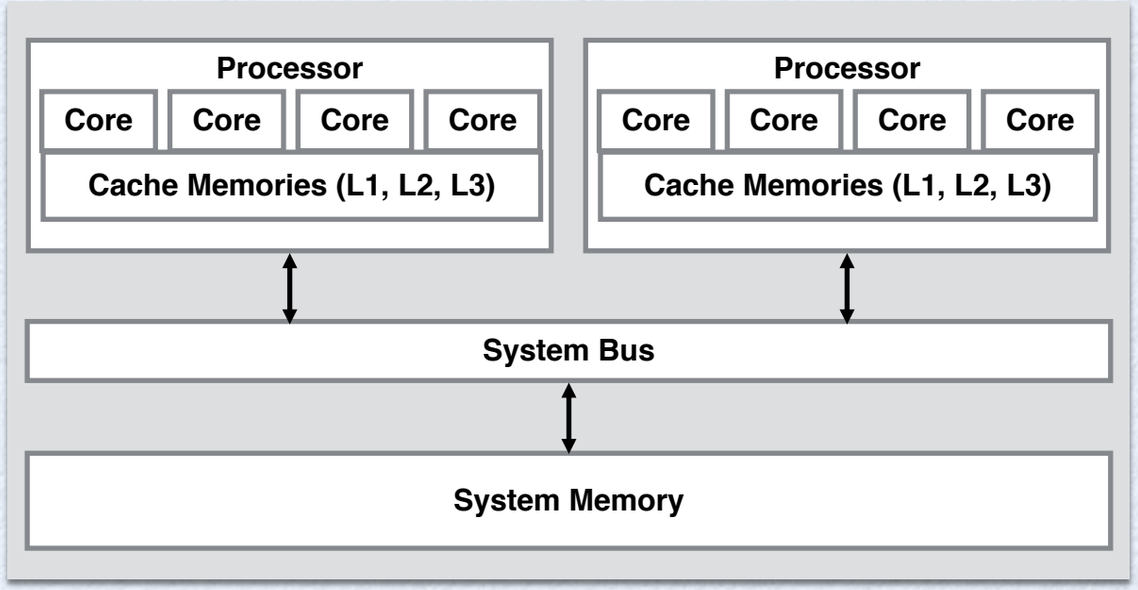


Cameron et al, IEEE Computer 2005

~99% of SOFTWARE uses < 10 % of HARDWARE

Some terminology

- Parallelism in Hardware:
  - multiple **cores** and **memory**
- Parallelism in Software:
  - **process**: executed program (has it's own memory space etc), can contain multiple threads, can run in parallel, can communicate with other processes
  - **thread**: can run in parallel and all threads of the same process share the application data (memory)




```
int a[1000];

int main( int argc, char** argv )
{
   for(int i =   0; i <  500; i++ ) a[i] = 0;
   for(int i = 500; i < 1000; i++ ) a[i] = 1;

   return 0;
}
```

Sequential Code

```
int main(int argc, char** argv )
{
    // vector size
    const int N = 1600000;

    // initialize vectors
    std::vector<float> x(N,-1.2), y(N,3.4), z(N);

    // do the sum z = x + y
    for(int i = 0; i < N; i++) z[i] = x[i] + y[i];

    return 0;
}
```

SIMD

```
int main(int argc, char** argv)
{
    // vector size
    const int N = 1600000;

    // initialize vectors (assume correct memory alignment)
    std::vector<float> x(N,-1.2), y(N,3.4), z(N);

    // DO THE SUM z = x + y with SSE (width=4)
    for( int i = 0; i < N; i += 4 )
    {
        // z[i] = x[i] + y[i];
        __m128 xx = _mm_load_ps( &x[i] );
        __m128 yy = _mm_load_ps( &y[i] );
        __m128 zz = _mm_add_ps( xx, yy );
        _mm_store_ps( &z[i], zz );
    }

    return 0;
}
```

POSIX Threads

```
struct arg_t
{
    float *x;
    float *y;
    float *z;
    int t;
    int chunk;
};


void *work(void *argument)
{
    struct arg_t *args = (struct arg_t *)argument;

    float *x = args->x;
    float *y = args->y;
    float *z = args->z;
    int t = args->t;
    int chunk = args->chunk;

    for (int i = t*chunk; i < (t+1)*chunk; i++)
           z[i] = x[i] + y[i];

    return NULL;
}
```

```
int main(int argc, char** argv)
{
    // vector size
    const int N = 1600000;

    // initialize vectors
    std::vector<float> x(N,-1.2), y(N,3.4), z(N);


    // DO THE SUM z = x + y using 4 threads
    int num_threads = 4;
    int chunk = N / num_threads;

    struct arg_t args[num_threads];
    pthread_t threads[num_threads];

    for (int t = 0; t < num_threads; t++)
    {
        args[t].x = &x[0];
        args[t].y = &y[0];
        args[t].z = &z[0];
        args[t].t = t;
        args[t].chunk = chunk;

        pthread_create(&threads[t], NULL, add, &args[t]);
    }

    for (int t = 0; t < num_threads; ++t)
        pthread_join(threads[t], NULL);

   return 0;

}
```

OpenMP

```
int main(int argc, char** argv)
{
    // vector size
    const int N = 1600000;

    // initialize vectors
    std::vector<float> x(N,-1.2), y(N,3.4), z(N);

    // do the sum z = x + y
    #pragma omp parallel for
    for (int i = 0; i < N; i++) z[i] = x[i] + y[i];

    return 0;
}
```

Code Optimization 1

```
// Faster Speed

main()
{           ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
}
```

```
// Smaller Size

main()
{           ...
            ...
            call_routine();
            ...
            ...
            call_routine();
            ...
            ...
            call_routine();
            ...
            ...
            call_routine();
            ...
            ...
}

void call_routine(void)
{
            XXXX
            YYYY
            ZZZZZ
}
```

Code Optimization 2

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

int main() {
  int n = 5;
  int result = factorial(n);
  printf("Factorial of %d is %d\n", n, result);
  return 0;
}
```

```
int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    n--;
  }
  return result;
}

int main() {
  int n = 5;
  int result = factorial(n);
  printf("Factorial of %d is %d\n", n, result);
  return 0;
}
```

Code Optimization 3

```
#define NUM_SINE_VALUES 256
const float sine_table[NUM_SINE_VALUES] = {
0, 0.0245, 0.0491, 0.0736, // ...and so on
};

float sine(float angle) {
  int index = (int) (angle / (2 * PI) * NUM_SINE_VALUES) % NUM_SINE_VALUES;
  return sine_table[index];
}
```