SOFT 261

Embedded Programming and the Internet of the Things

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk
Website: https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Date
26/03/2019

School of Computing
(University of Plymouth)

Outline

- Modern Embedded Systems and architectures
  - Heterogeneous Systems
  - Comparison of different embedded system architectures
    - Application specific processors
      - DSPs
      - ASICs
      - ASIPs
    - Co-processors
      - FPGAs
      - GPUs

Hardware Trends on Embedded Systems - From single core processors to heterogeneous systems on a chip

Driven by
frequency scaling
Constrained by
power consumption

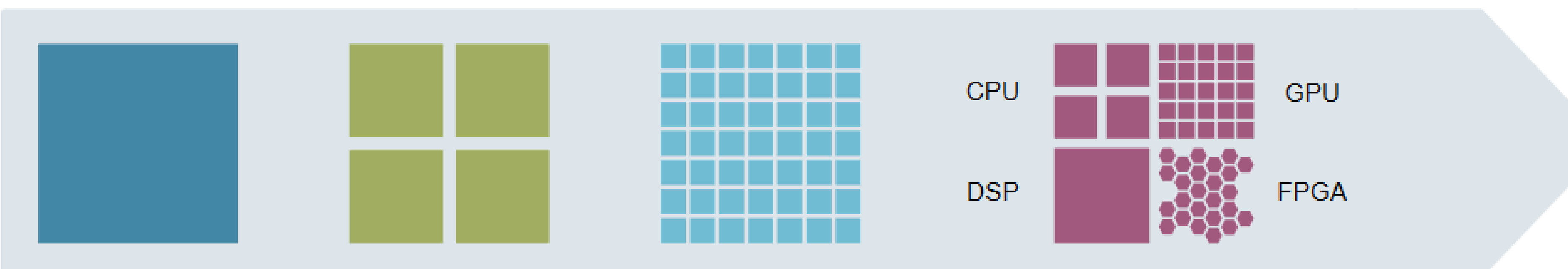
Single-Core
1975-2005

Multicore
since 2005

Manycore
since 2008

Heterogeneous systems
today

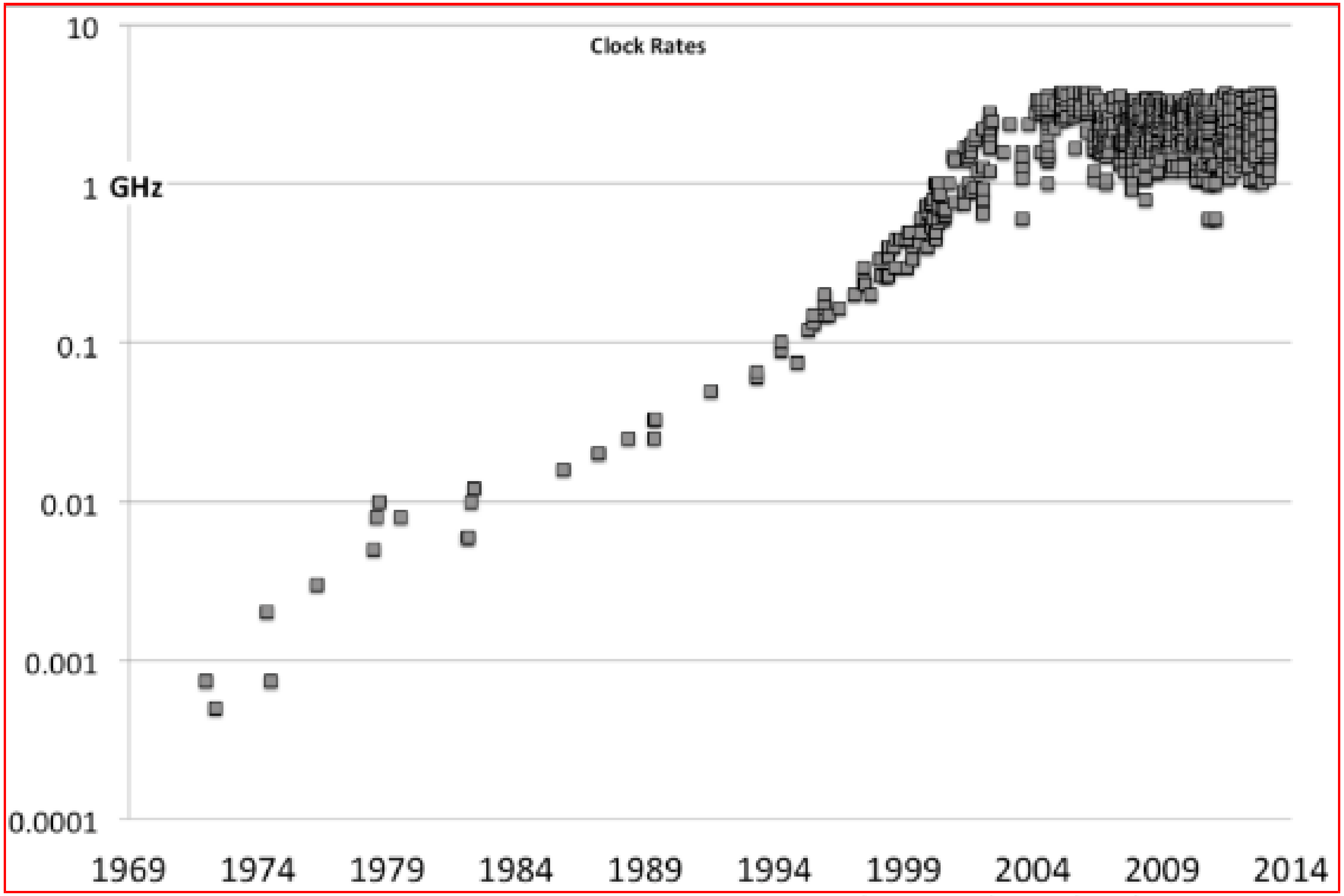
Driven by
high performance per Watt ratio
Constrained by
programming complexity



H. Esmaeilzadeh et al., "Dark silicon and the end of multicore scaling", International Symposium on Computer Architecture (ISCA). ACM, 2011.
M. Zahran, "Heterogeneous Computing Here to Stay". ACM Queue, Nov/Dev 2016.

Taken from https://embb.io/downloads/MTAPI_EMBB.pdf

The CPU frequency has ceased to grow

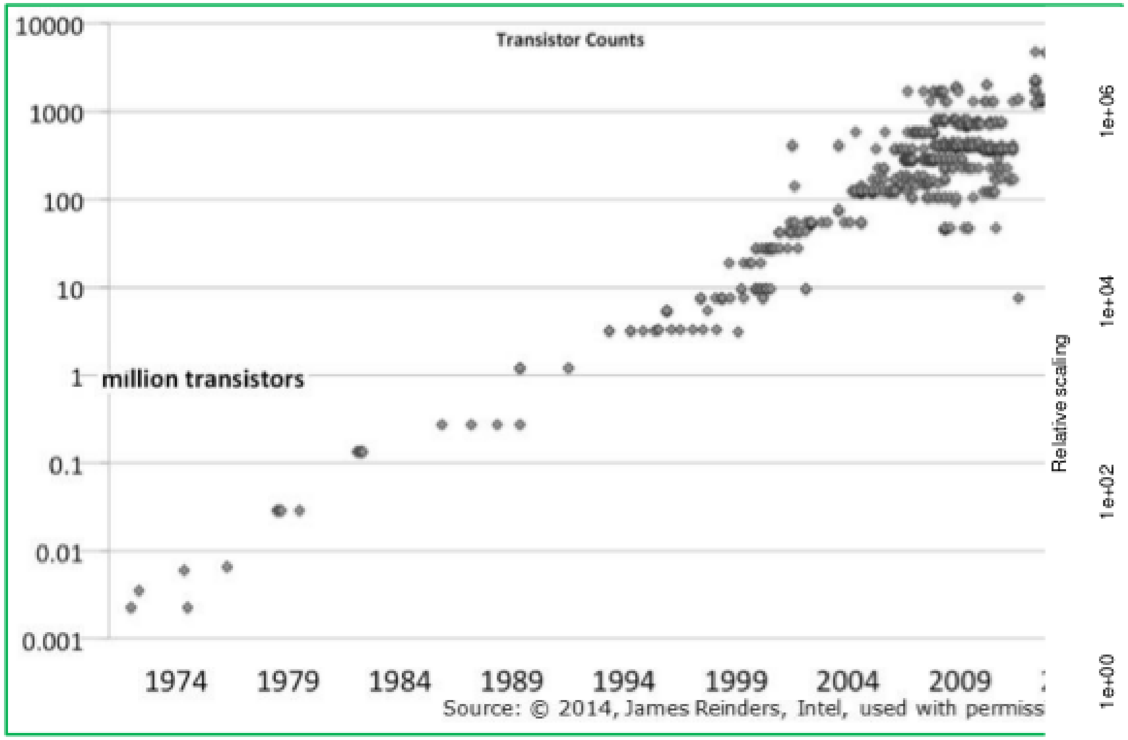


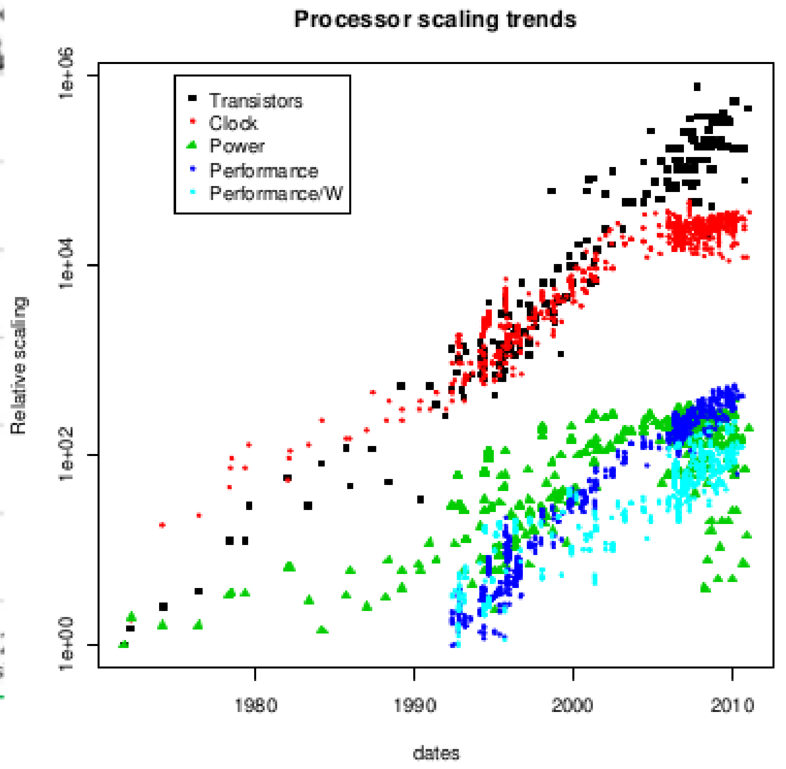
Moore's Law Is STILL Going Strong

Hardware performance potential **continues to grow**

"We think we can continue Moore's Law for at least another 10 years."

Intel Senior Fellow Mark Bohr, **2015**





Embedded Systems Hardware Evolution

- Scalar Processors
- Pipelined Processors
- Superscalar and VLIW Processors
- Out of order Processors
- Processors support Vectorization
- Multicore Processors
- Heterogeneous systems

Time

Heterogeneous computing (1)

Single core Era **->** Multi-core Era **->** Heterogeneous Systems Era

- **Heterogeneous computing refers to systems that use more than one kind of processors or cores**
  - These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar (co)-processors, usually incorporating specialized processing capabilities to handle particular tasks
  - Systems with General Purpose Processors (GPPs), GPUs, DSPs, ASIPs etc.

- **Heterogeneous systems offer the opportunity to significantly increase system performance and reduce system power consumption**

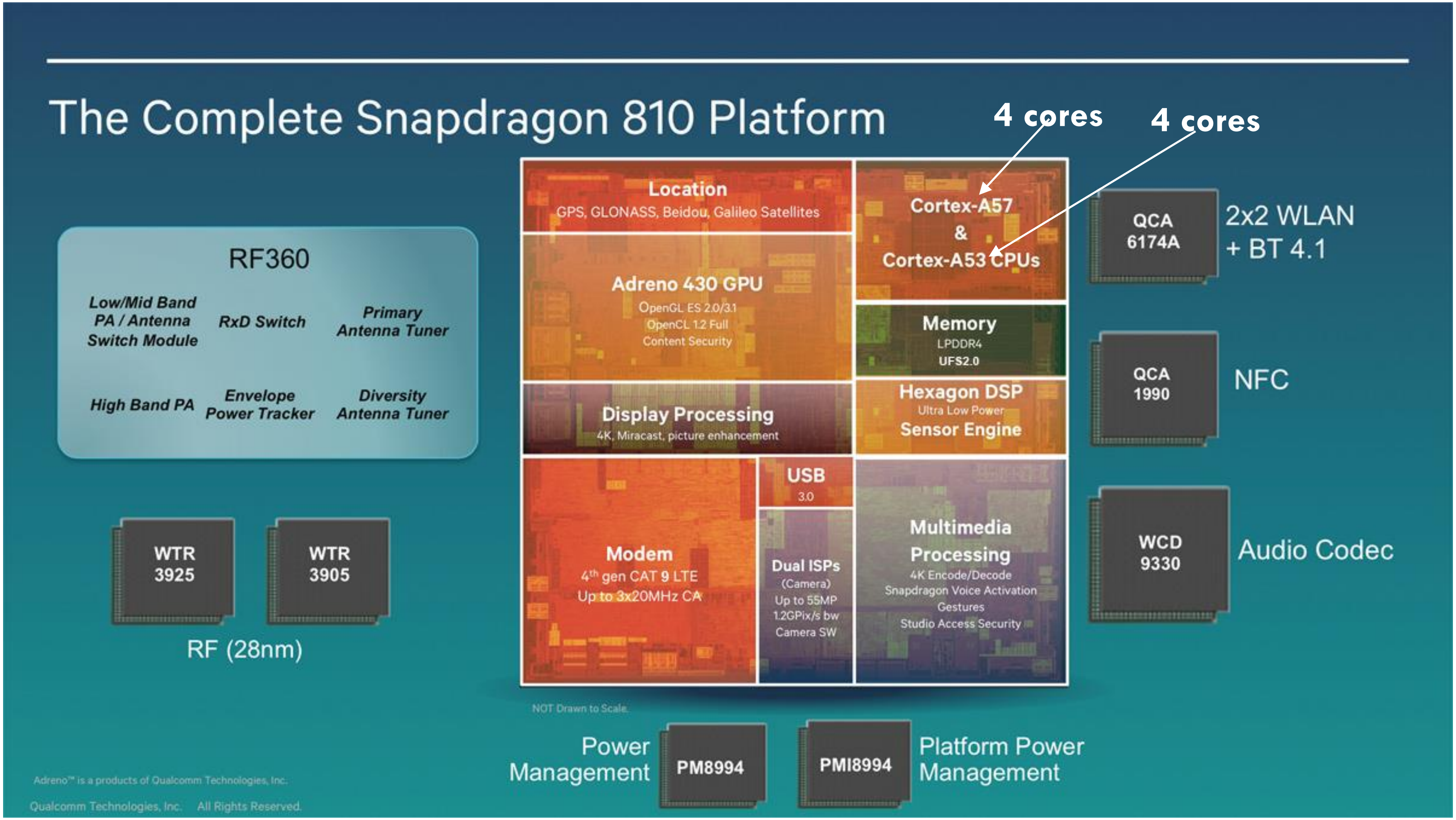
Heterogeneous computing (2)

- Software issues:
  - Offloading
  - Programmability – think about CPU code (C code), GPU code (CUDA), FPGA code (VHDL)
  - Portability - What happens if your code runs on a machine with an FPGA instead of a GPU

Comparisons between
Homogeneous and Heterogeneous Computing

Symmetric, Same cores (Usually CPUs)	Assymmetric, Different cores (CPUs, GPUs, DSPs and accelerators)
operation is guaranteed to be same at each core	operation cannot be supposed to be same at each core
easy to off load tasks	more complicated to off load tasks
good compatibility	less compatibility **specialized for specific tasks**

Heterogeneous computing (3) – A mobile phone system



Think-Pair-Share Exercise

- What is in your opinion the most appropriate computer architecture for a smart phone and why?
  a. 1 microcontroller
  b. 1 normal speed GPP, e.g., Pentium II
  c. 1 quad-core Intel i7
  d. A heterogeneous computer architecture with a CPU, a DSP, a GPU, some ASICs

Application Specific Processors

- General purpose processors offer good performance for all different applications but specific purpose processors offer better for a specific task
- **Application specific processors emerged as a solution for**
  - **higher performance**
  - **lower power consumption**
  - **Lower cost**
- Application specific processors have become a part of our life and can be found almost in every device we use on a daily basis
- Devices such as TVs, mobile phones and GPSs they all have application specific processors
- **They are classified into**
  1. **Digital Signal Processor (DSPs)**
  2. **Application Specific Instruction Set Processors (ASIPs)**
  3. **Application Specific Integrated Circuit (ASICs)**

Digital Signal Processors (DSPs)

1. **DSP**: Programmable microprocessor for extensive real-time mathematical computations
   - specialized microprocessor with its architecture optimized for the operational needs of digital signal processing
   - DSP processors are designed specifically to perform large numbers of complex arithmetic calculations and as quickly as possible
   - DSPs tend to have a different arithmetic Unit architecture;
     - specialized hardware units, such bit reversal, Multiply-accumulate units etc
     - Normally DSPs have a small instruction cache but no data cache memory

Application Specific Instruction set Processor (ASIP)

2. **ASIP**: Programmable microprocessor where hardware and instruction set are designed together for one special application
   - Instruction set, micro architecture and/or memory system are customised for an application or family of applications
   - Usually, they are divided into two parts: static logic which defines a minimum ISA and configurable logic which can be used to design new instructions
   - The configurable logic can be programmed and extend the instruction set similar to FPGAs
   - better performance, lower cost, and lower power consumption than GPP

Application Specific Integrated Circuit (ASIC)

3. **ASIC:** Algorithm completely implemented in hardware
   - An Integrated Circuit (IC) designed for a specific line of a company – full custom
   - It cannot modified – it is produced as a single, specific product for a particular application only
   - Proprietary by nature and not available to the general public
   - ASICs are full custom therefore they require very high development costs
   - ASIC is just built for one and only one customer
   - ASIC is used only in one product line
   - Only volume production of ASICs for one product can make sense which means low unit cost for high volume products, otherwise the cost is not efficient
   - **There is a lot of effort to implement an ASIC – there are specific languages such as VHDL and Verilog**

Building an application specific system on an embedded system (1)

Consider that we want to build and application specific system. We can choose**:**

1. GPP

- Functionality of the system is exclusively build on the software level
- it is not efficient in term of performance, power consumption, cost, chip area and heat dissipation

2. ASIC:

- No flexibility and extensibility

3. ASIP:

- a compromise between the two extremes
- used in embedded and system-on-chip solutions

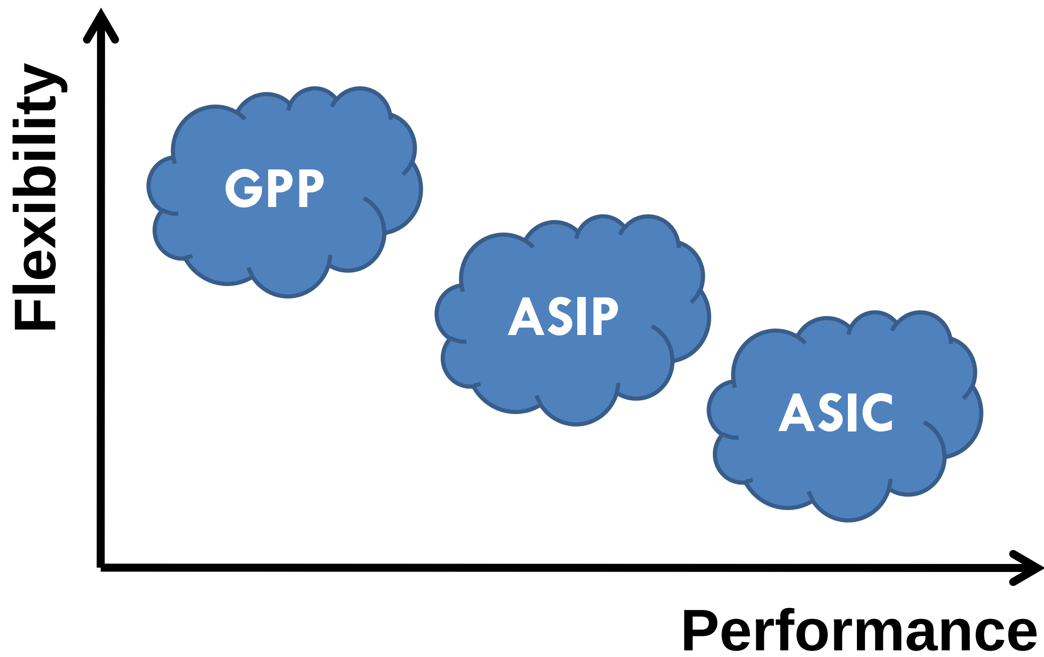


Fig.4. Comparison between Performance and flexibility

Building an application specific system on an embedded system (2)

Table 1. Comparison between different approaches for Building Embedded Systems [1]

	GPP	ASIP	ASIC
Performance	Low	High	Very High
Flexibility	Excellent	Good	Poor
HW design	None	Large	Very large
SW design	Small	Large	None
Power	Large	Medium	Small
reuse	Excellent	Good	Pure
market	Very large	Relatively large	Small
Cost	High	Medium	Volume sensitive

Accelerators - coprocessors

- Accelerators / co-processors are used to perform some functions more efficiently than the CPU

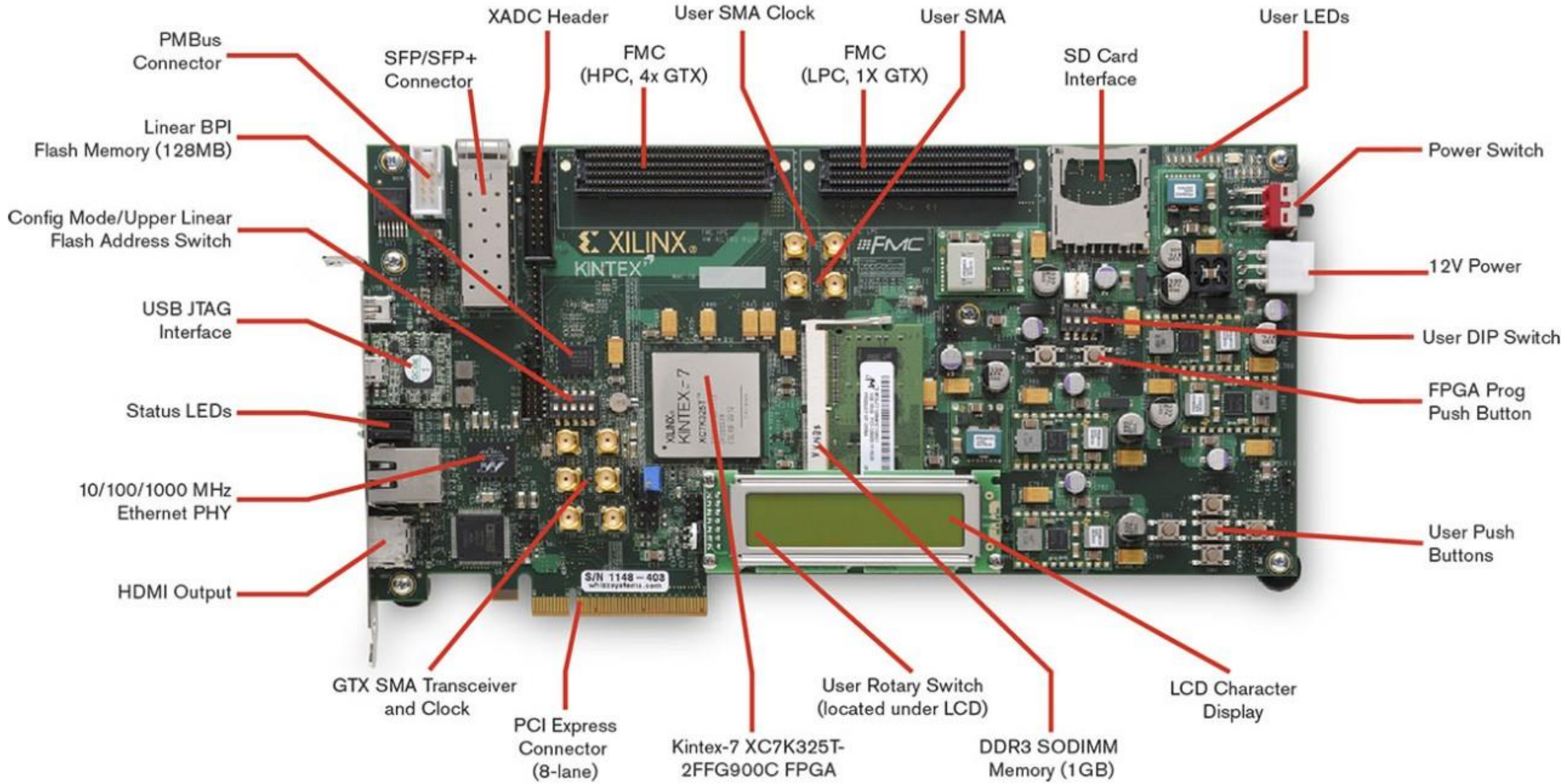
- They offer
  - Higher performance
  - Lower power consumption
  - But they are harder to program

Field Programmable Gate Arrays (FPGAs)

- FPGAs are devices that allow us to create our own digital circuits
- An FPGA (Field Programmable Gate Array) is an array of logic gates that can be hardware-programmed to fulfill user-specified tasks
  - **FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"**
  - An application can be implemented entirely in HW
  - The FPGA configuration is generally specified using a hardware description language (HDL) like VHDL and Verilog – **hard to program**
  - High Level Synthesis (HLS) provides a solution to this problem. Engineers write C/C++ code instead, but it is not that efficient yet

FPGAs (2)

- FPGAs come on a board. This board is connected to a PC and programmed. Then, it can work as a standalone component

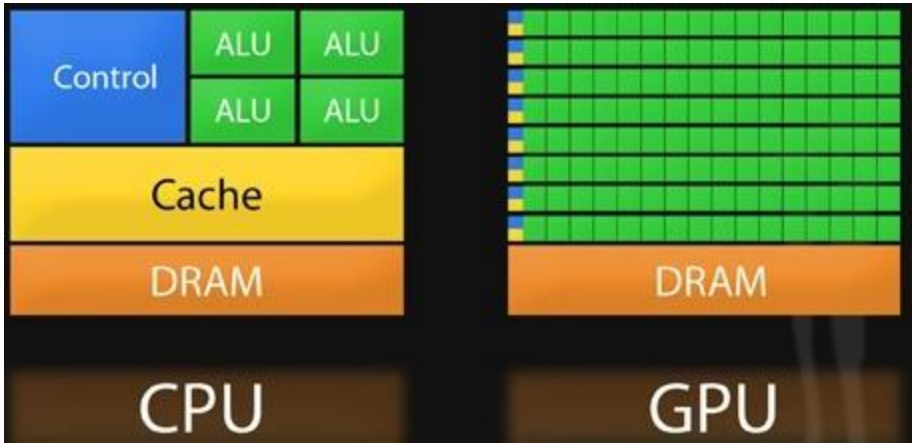


FPGAs (3)

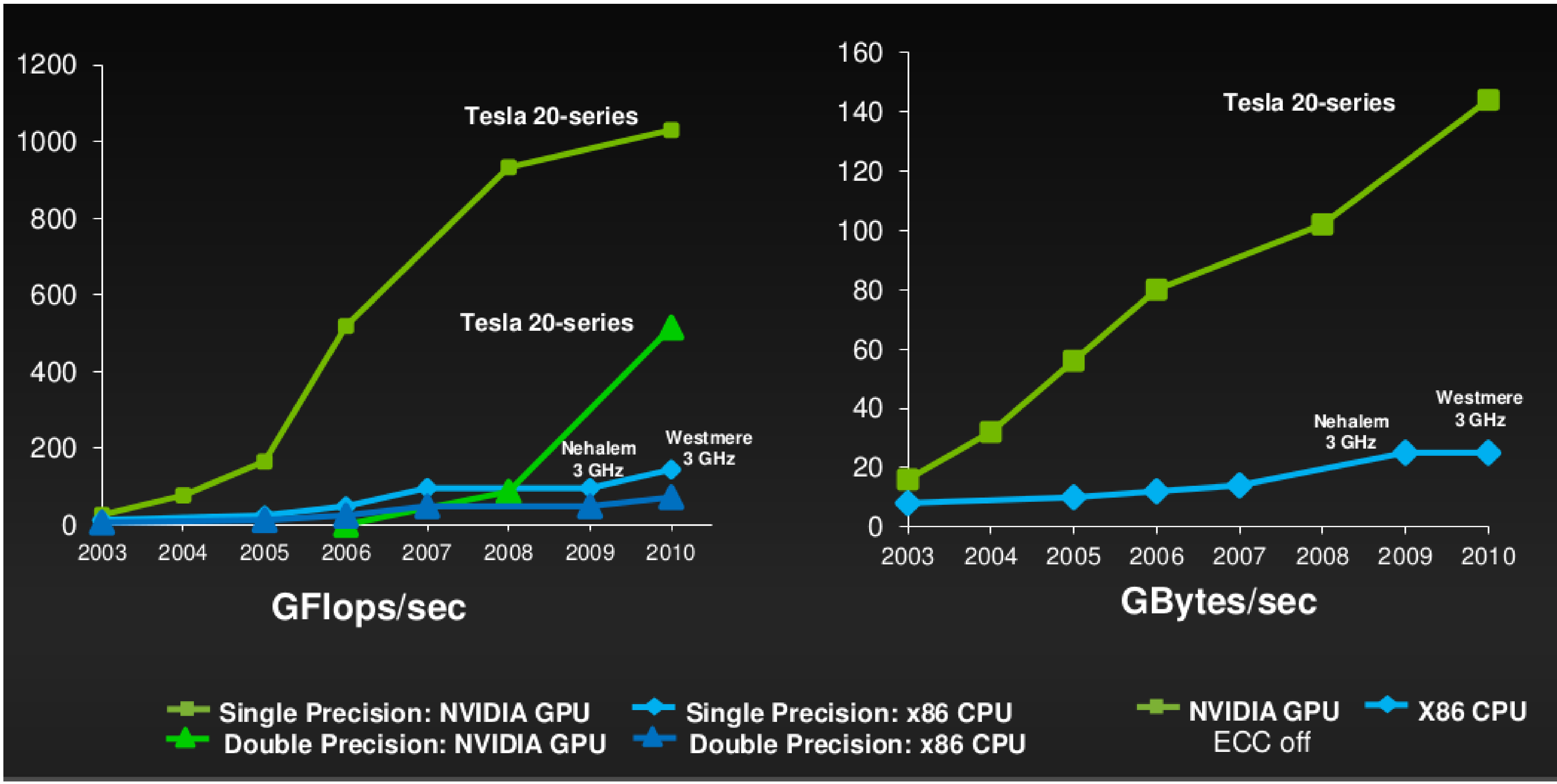
- **Unlike an ASIC the circuit design is not set and you can reconfigure an FPGA as many times as you like!**
  - Creating an ASIC also costs potentially millions of dollars and takes weeks or months to create.
  - However, the recurring cost is lower than the cost of the FPGA (no silicon area is wasted in ASICs).
  - ASICs are cheaper only when the production number is very high
- Intel plans hybrid CPU-FPGA chips

GPUs (1)

- Graphics Processing Unit (GPU)
  - The GPU's advanced capabilities were originally used primarily for 3D game graphics. But now those capabilities are being harnessed more broadly to accelerate computational workloads in other areas too
  - GPUs are very efficient for
    - Data parallel applications
    - Throughput intensive applications - the algorithm is going to process lots of data elements



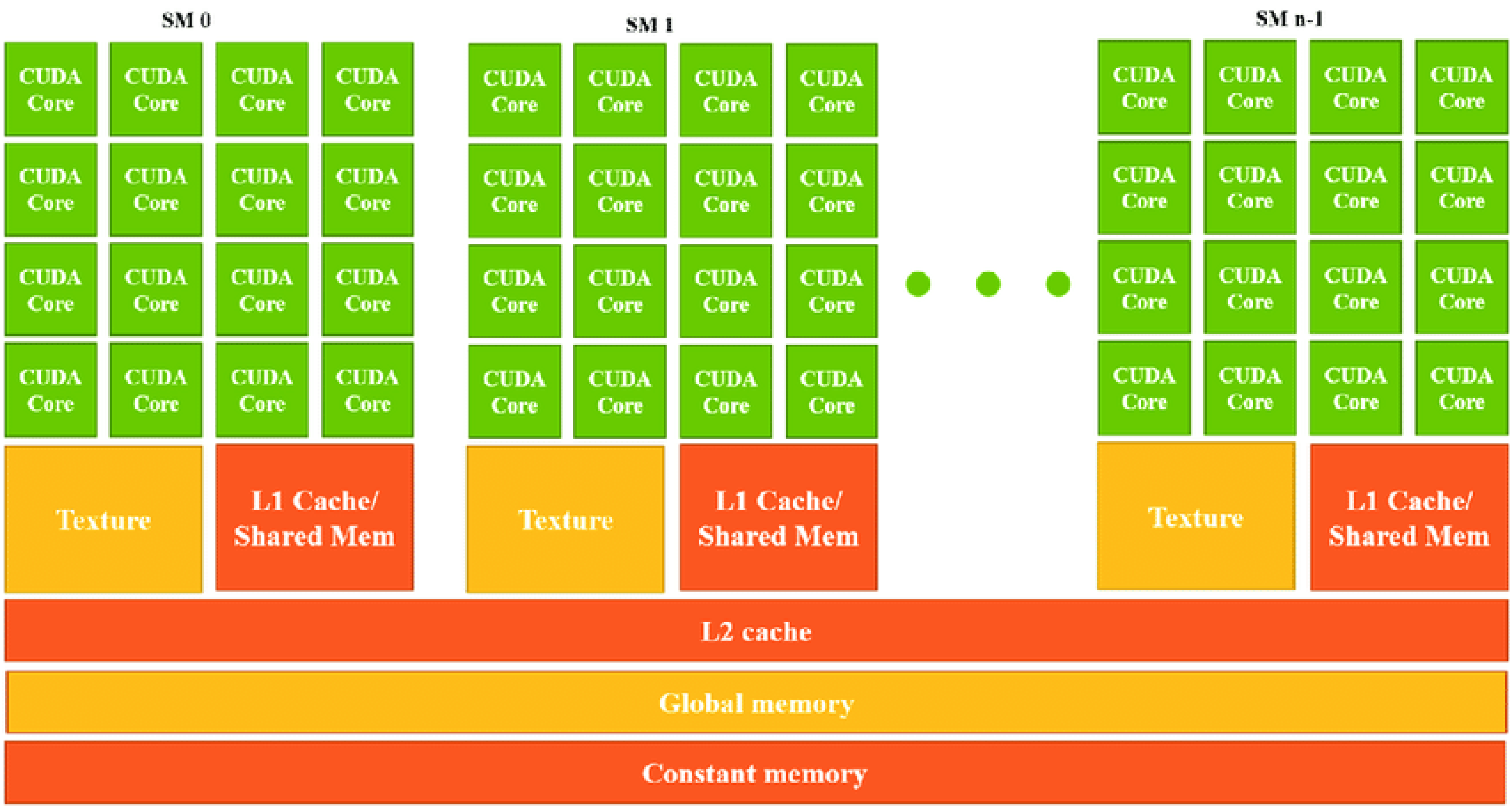
GPUs (2) – why do we need GPUs?



GPUs (3)

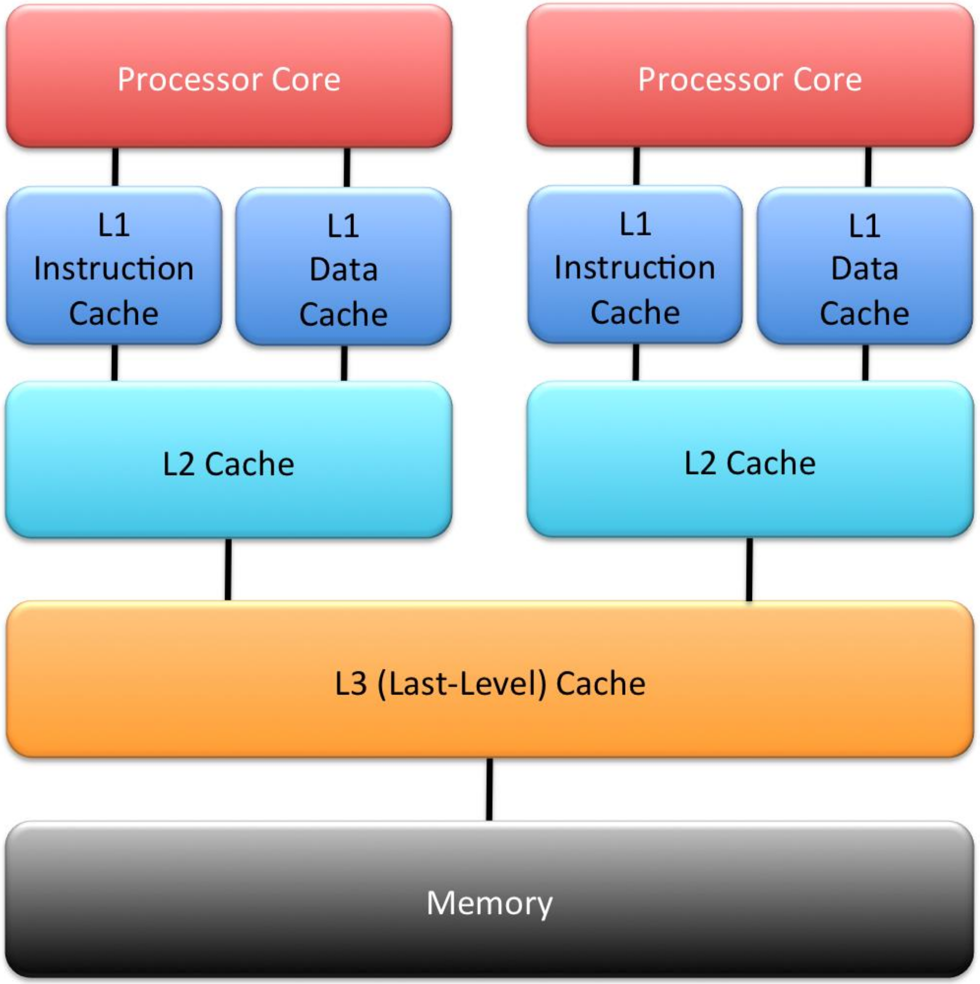
- A GPU is always connected to a CPU – GPUs are coprocessors
- GPUs work in lower frequencies than CPUs
- GPUs have many processing elements (up to 1000)
- GPUs have smaller and faster cache memories
- OpenCL is the dominant open general-purpose GPU computing language, and is an open standard
- The dominant proprietary framework is Nvidia CUDA

Schematic of Nvidia GPU architecture

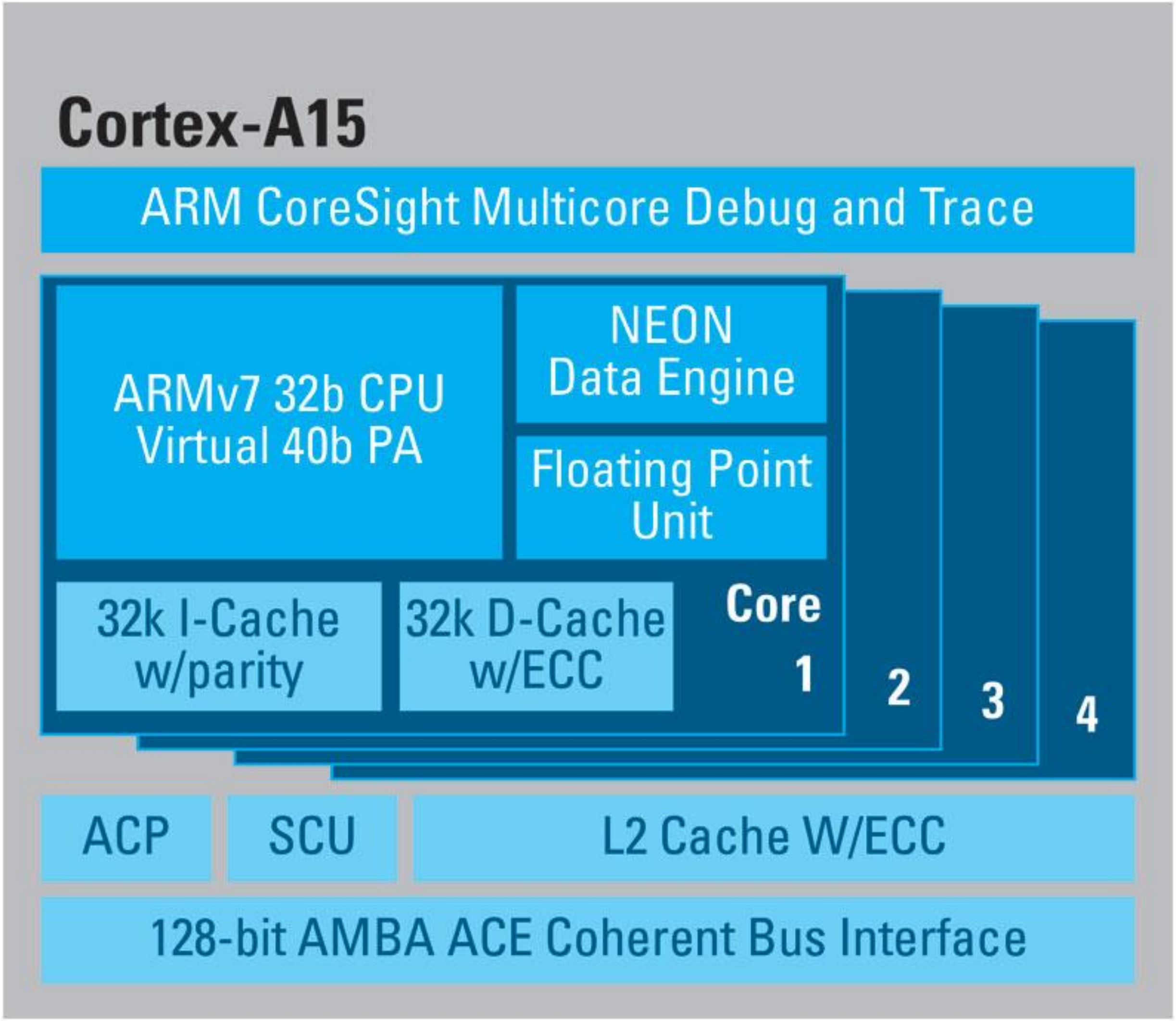


Multi-core CPUs

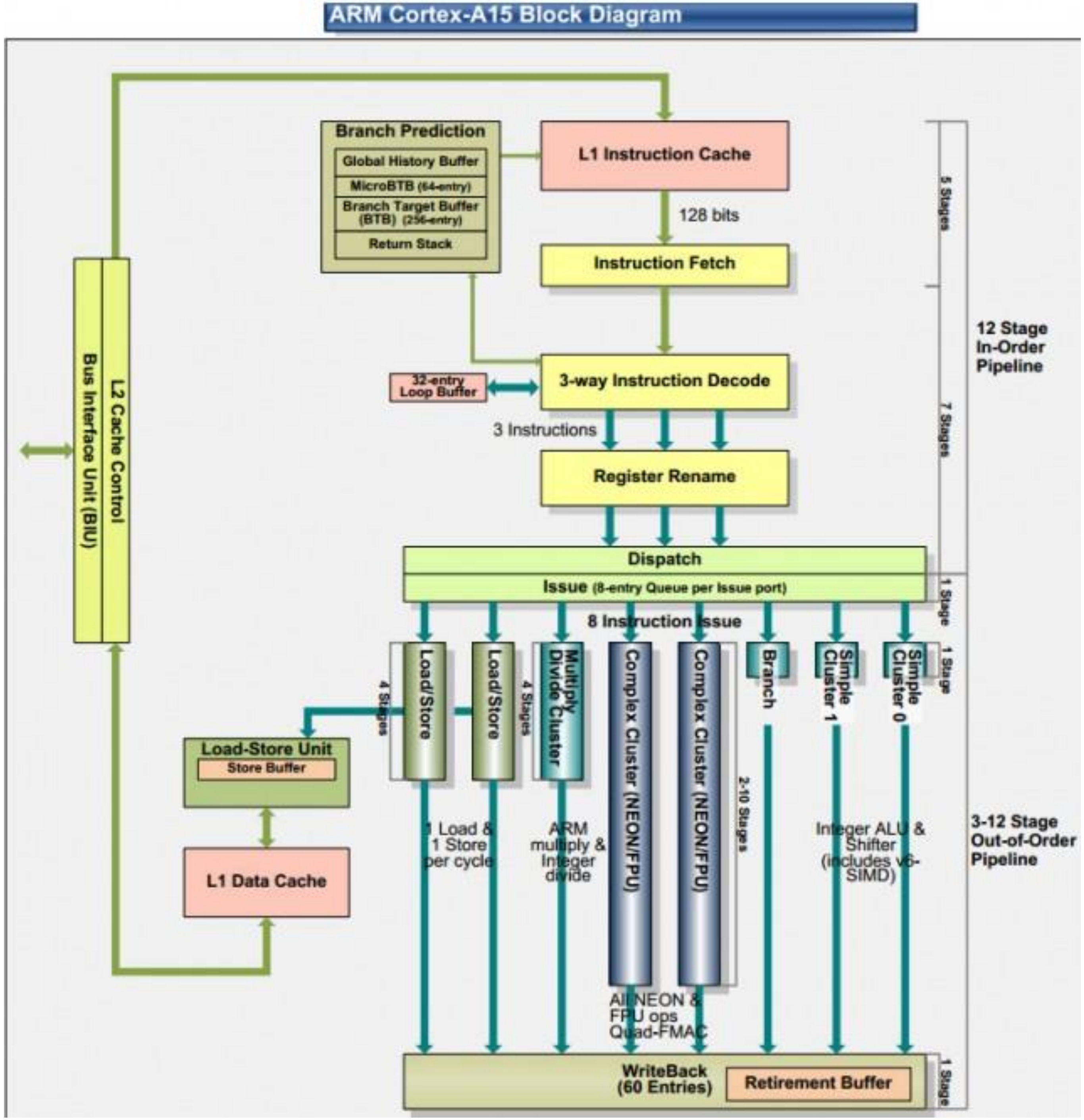
- Multiple cores on the same chip using a shared cache
- Typically from 2-8 cores
- Both cores compete for the same hardware resources
- Both cores are identical
- Every core is a superscalar out of order CPU



Multi-core CPUs – ARM Cortex-A15

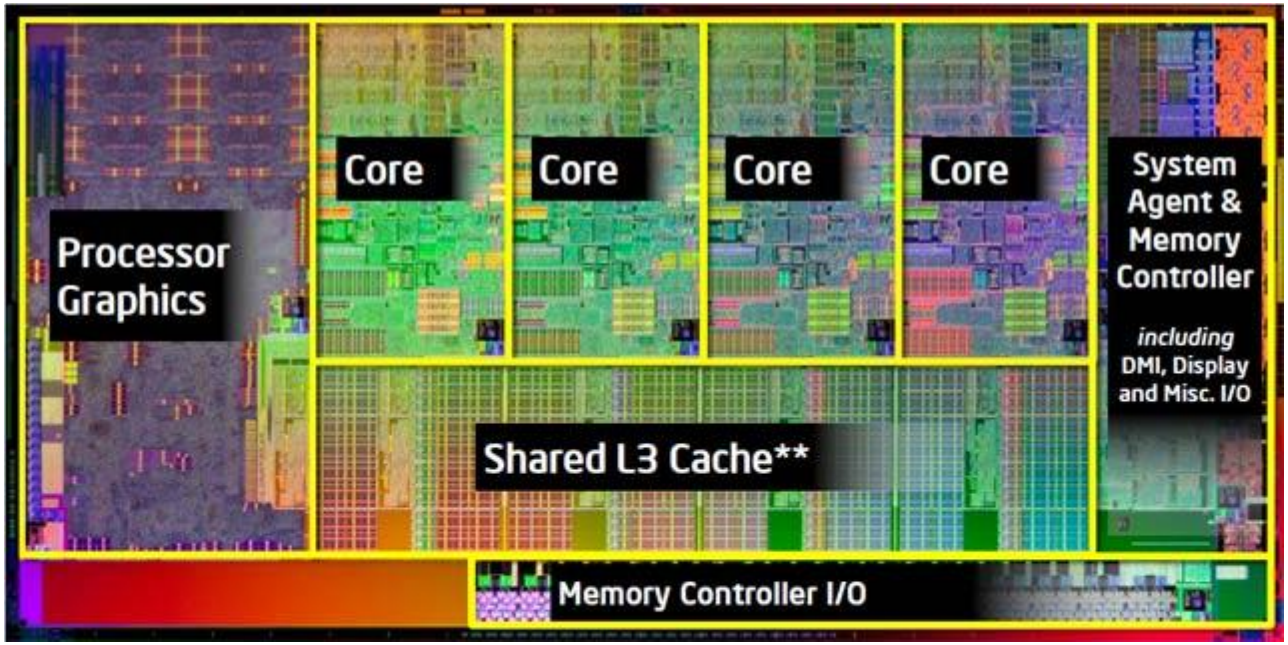


ARM Cortex-A15



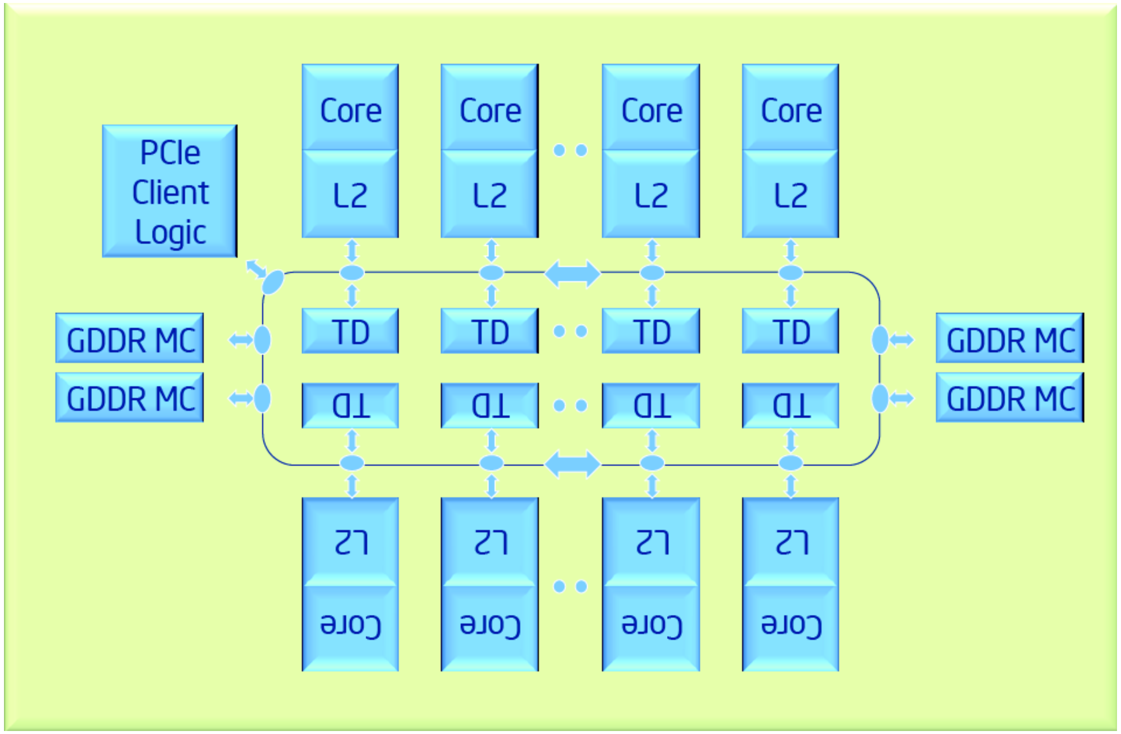
Multi-core CPUs - Intel i7 architecture

- In the figure below there is the Intel i7 CPU, where four CPU cores and the GPU reside in the same chip

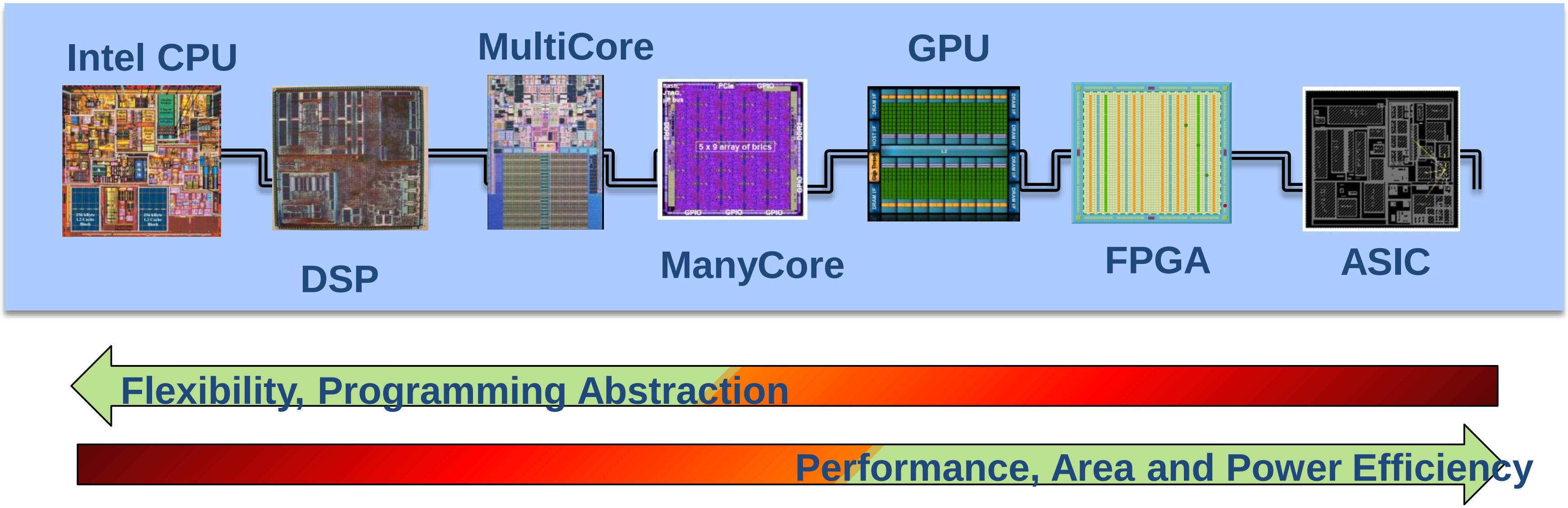


Many core Processors – Intel Xeon Phi

- They are intended for use in supercomputers, servers, and high-end workstations
- 57-61 in-order simpler than i7 cores
- 1-1.7 Ghz
- 512bit vector instructions
- each core is connected to a ring interconnect via the Core Ring Interface



Comparison



CPU:
- Market-agnostic
- Accessible to many programmers (Python, C++)
- Flexible, portable

FPGA:
- Somewhat Restricted Market
- Harder to Program (VHDL, Verilog)
- More efficient than SW
- More expensive than ASIC

ASIC
- Market-specific
- Fewer programmers
- Rigid, less programmable
- Hard to build (physical)

Conclusions

- Modern Embedded Systems include Parallel Heterogeneous Computer Architectures
- General purpose processors + specific purpose processors + co-processors
- Heterogeneous systems offer the opportunity to significantly
  - increase performance
  - reduce power consumption
  - reduce cost
- Issues:
  - Programmability
  - Portability
  - Design good Compilers – optimize the code

Code Optimization

- **Week 9**
  - Basic and simple code optimizations
  - Real world image processing application
- **Week 10**
  - Register blocking
  - Real world use cases
- **Week 11**
  - Vectorization
  - More advanced optimizations
  - Real world use cases
- **Week 12**
  - Advanced optimizations
  - Real world use cases

Outline

- Code optimization
  - key problems
- Some **basic/simple code optimizations**/transformations and manually applied techniques:
  - Use the available Compiler Options
  - Reduce complex operations
  - Loop based strength reduction
  - Dead code elimination
  - Common subexpression elimination
  - Use the appropriate precision
  - Choose a better algorithm
  - Loop invariant code motion
  - Use table lookups
  - Function Inline
  - Loop unswitching
  - Loop unroll
  - Scalar replacement

What is Code Optimization?

- Optimization in terms of
  - Execution time
  - Energy consumption
  - Space (Memory size)
    - Reduce code size
    - Reduce data size

How to optimize ?

- **Optimizing the easy way**
  - Use a faster programing language, e.g., C instead of Python
  - Use a better compiler
  - Manually enable specific compiler's options
  - Normally, the optimization gain is limited
  - No expertise is needed

- **Optimizing the hard way**
  - use a profiler to identify performance bottlenecks, normally loop kernels
  - Manually apply code optimizations
  - Re-write parts of the code from scratch
  - Needs expertise
  - Optimization gain is high

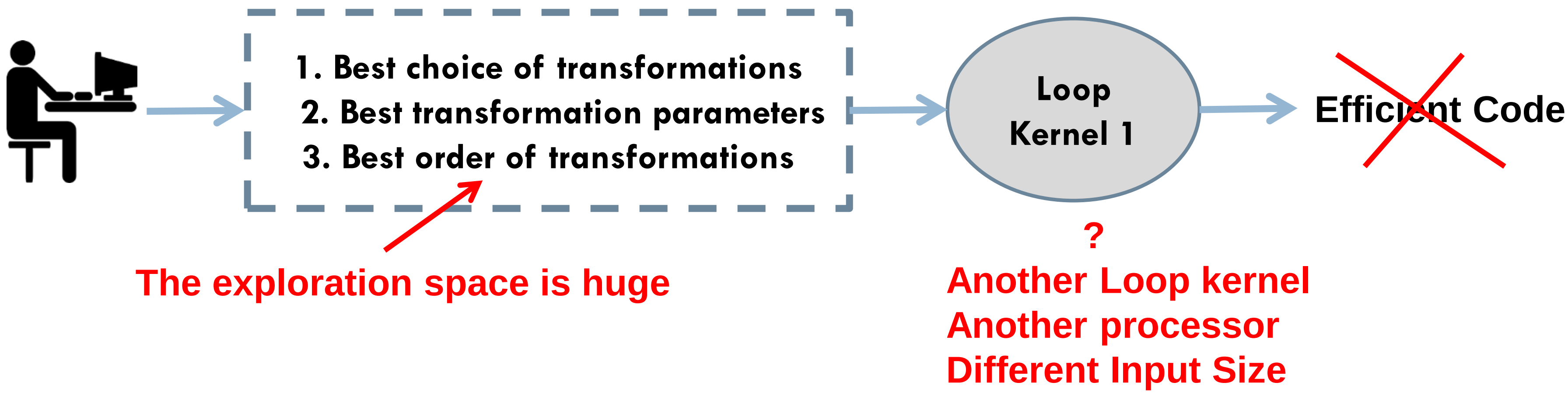
Introduction

- Loops represent the most computationally intensive part of a program.
- Improvements to loops will produce the most significant effect

- Loop optimization
  - **90% / 10% rule**
  - Normally, "90% of a program's execution time is spent in executing 10% of the code"
    - larger payoff to optimize the code within a loop

Which Compiler Options to use and when?

- Compilers offer a large number of transformation/optimization options
- This is a complex longstanding and unsolved problem for decades
  - Which compiler optimization/transformation to use?
  - Which parameters to use? Several optimizations include different parameters
  - In which order to apply them?



Optimizing SW - problem (1)

- **The key to optimizing software is the correct**
  - **Choice**
  - **Order**
  - **Parameters**

  of code optimizations

- One of the most used compilers is gcc
- You can find its options here https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html

- **But why optimizing SW is so hard?**

- Normally, the efficient optimizations for a specific code are not efficient for
  - another code
  - another processor
  - different hardware architecture details, e.g., cache line size
  - or even for a different input size

Optimizing SW – problem (2)

- Why compilers can't find the optimum choice, order and parameters of optimizations?
  1. Compilers are not smart enough to take into account
     - ✓ most of the hardware architecture details (e.g., cache size and associativity)
     - ✓ custom algorithm characteristics (e.g., data access patterns, data reuse, algorithm symmetries)
     - Even experienced programmers
       - Do not understand how software runs on the target hardware
       - Treat threads as black boxes
       - Blindly apply loop transformations

  - Peak performance demands going low level
    - Understand the hardware, compilers, ISA

Optimizing SW – problem (3)

- Why compilers can't find the optimum choice, order and parameters of optimizations?
  2. The compilation sub-problems depend on each other which makes the problem extremely difficult
     - ✓ these dependencies require that all the problems should be optimized together as one problem and not separately

  - Toward this much research has been done
    - Iterative compilation techniques
    - Methodologies that simultaneously optimize only two problems
    - Searching and empirical methods
    - Heuristics
    - But …
      - They are partially applicable
      - They cannot give the best solution

Optimizing SW – problem (4)

- Why compilers can’t find the optimum choice, order and parameters of optimizations?
  3. The exploration space (all different implementations/binaries) is so big that it cannot be searched; researchers try to decrease the space by using
     - machine learning compilation techniques
     - genetic algorithms
     - statistical techniques
     - exploration prediction models focusing on beneficial areas of optimization search space

     - however, the search space is still so big that it cannot be searched, even by using modern supercomputers

Basic and Simple techniques that will improve your code

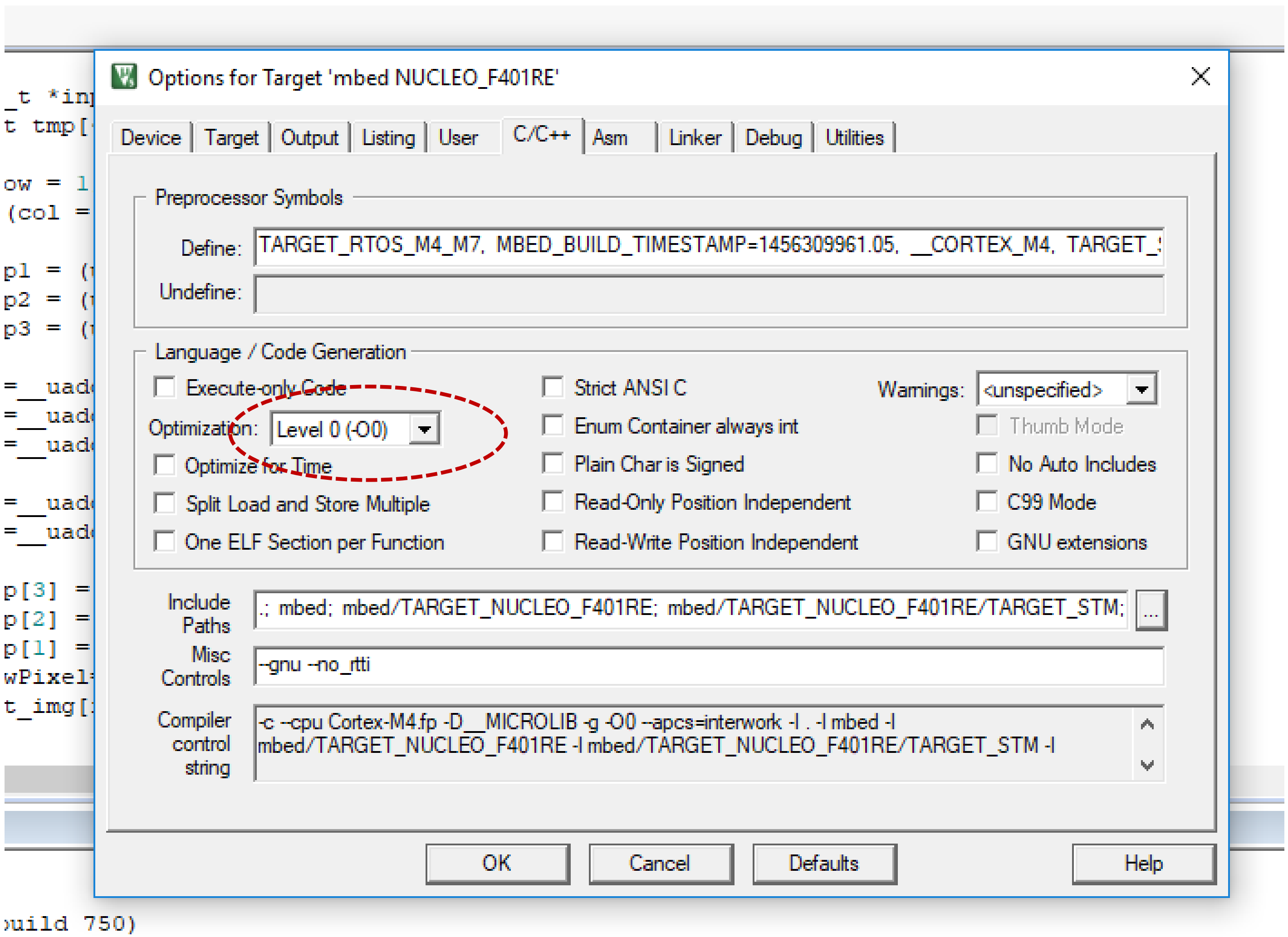
- Use the available Compiler Options
- Reduce complex operations
- Loop based strength reduction
- Dead code elimination
- Common subexpression elimination
- Use the appropriate precision
- Choose a better algorithm
- Loop invariant code motion
- Use table lookups
- Function Inline
- Loop unswitching
- Loop unroll
- Scalar replacement

Use the available compiler options

- **The most used optimization flags/options are the following** (have a look at Keil's options)
  - **' –O0'** - Disables all optimizations, but the compilation time is very low
  - **' –O1'** - Enables basic optimizations
  - **' –O2'** - Enables more optimizations
  - **' –O3'** - turns on all optimizations specified by -O2 and enables more aggressive loop transformations such as register blocking, loop interchange etc
  - **'-Ofast' option - be careful:** it is not always safe for codes using floating point arithmetic
  - **'Osize' option –** Optimizes for code size

- *gcc options can be found here:* https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html

Use the available compiler options



Loop unroll transformation (1)

- Creates additional copies of loop body
- Always safe

```c
//C-code1
for (i=0; i < 100; i++)
  A[i] = B[i];
```

⇨

```c
//C-code2
for (i=0; i < 100; i+=4) {
  A[i] = B[i];
  A[i+1] = B[i+1];
  A[i+2] = B[i+2];
  A[i+3] = B[i+3];
}
```

Pros:

✓ Reduces the number of instructions

✓ Increase instruction parallelism

Cons:

– Increases code size

– Increases register pressure

Loop unroll transformation (2)

// C code1

```
for (i=0; i<100; i++) {
…
}
```

100 times

// assembly code1

A[i] = B[i];

```
loop_i …
…
inc i         // increment i
cmp i, 100    // compare i to 100
jl loop_i     // jump if  i lower to 100
```

// C code2

```
for (i=0; i<100; i+=4) {
…
}
```

100/4 times

// assembly code2

```
loop_i …
…
…
…
…
inc i         // increment i
cmp i, 100    // compare i to 100
jl loop_i     // jump if lower
```

A[i] = B[i];
A[i+1] = B[i+1];
A[i+2] = B[i+2];
A[i+3] = B[i+3];

✓ The number of arithmetical instructions is reduced

1. Less add instructions for i, i.e., i=i+4 instead of i=i+1
2. Less compare instructions, i.e., i==100 ?
3. Less jump instructions

Scalar replacement transformation

- Converts array reference to scalar reference
- Most compilers will do this for you automatically by specifying '-O2' option
- Always safe

```
//Code-1
for (i=0; i < 100; i++){
  A[i] = … + B[i];
  C[i] = … + B[i];
  D[i] = … + B[i];
}
```

⇨

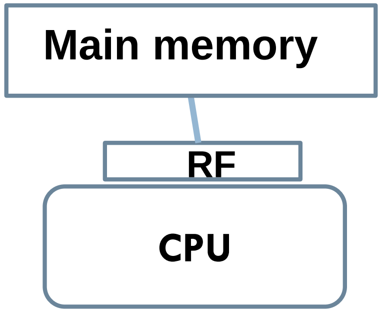
```
//Code-2
for (i=0; i < 100; i++){
  t=B[i];
  A[i] = … + t;
  C[i] = … + t;
  D[i] = … + t;
}
```

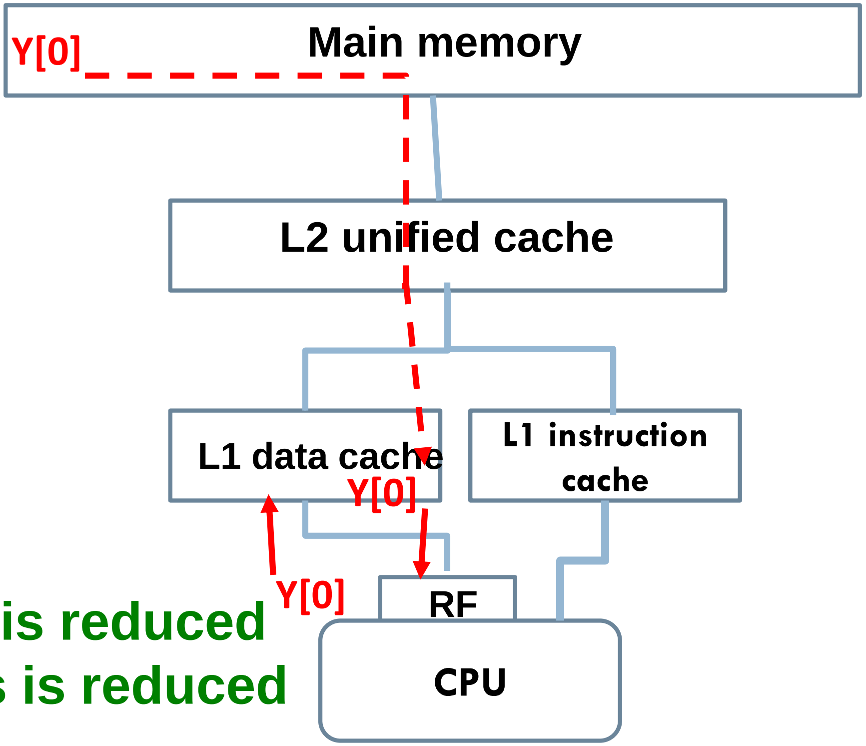
✓ Reduces the number of L/S instructions
✓ Reduces the number of memory accesses

Scalar Replacement Transformation example (1)


```
// C-code1
for (i=0; i<300; i++)
  for (j=0; j<300; j++)
    Y[i] += A[i][j] * X[j];
```

```
// C-code2
for (i=0; i<300; i++) {
   tmp=Y[i];
  for (j=0; j<300; j++) {
   tmp+= A[i][j] * X[j];
   }
  Y[i]=tmp;
}
```



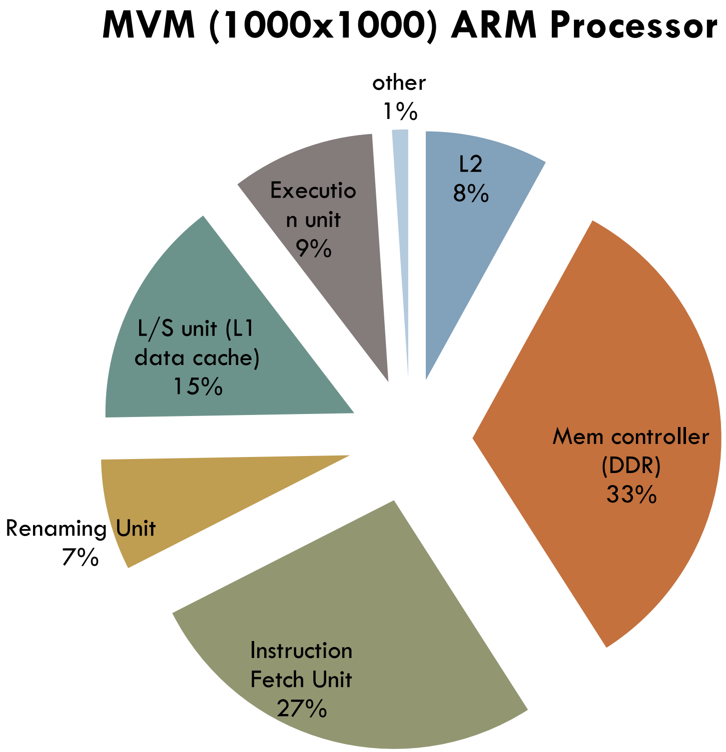


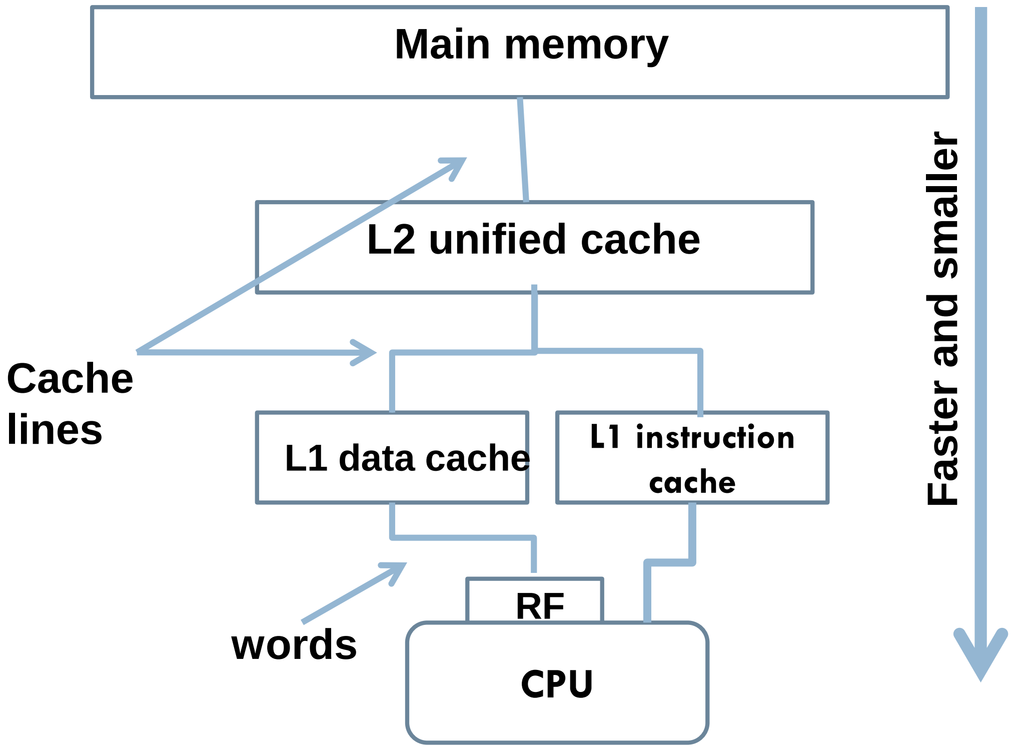
- Y[i] is not affected by j loop
- For every j, Y[i] is redundantly loaded/stored from/to memory
- A load/store instruction needs 1-3 CPU cycles

✓ **the number of L/S instructions is reduced**

✓ **the number of L1 data accesses is reduced**

Energy Consumption on ARM Cortex-A for Matrix-Vector Multiplication algorithm





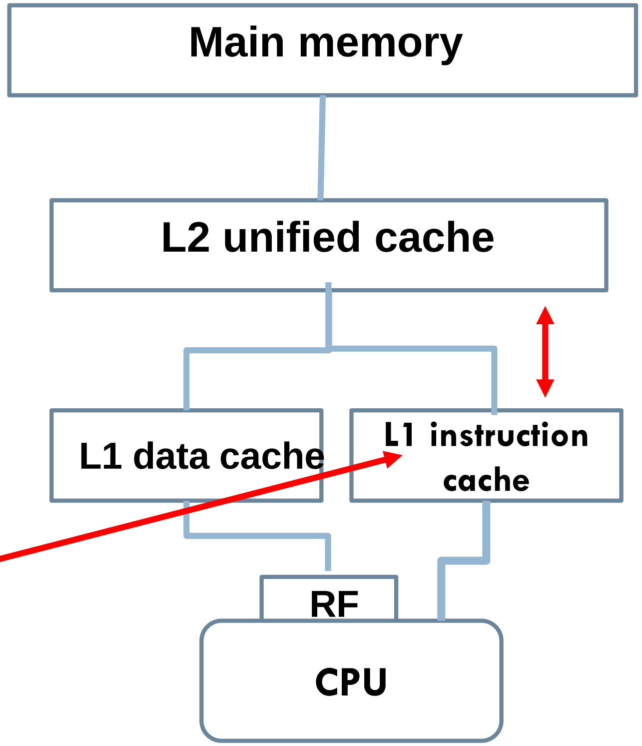
You have learned that the largest the loop unroll factor, the largest the gain in instructions, but is it always efficient?

- When code2 is faster than code1?
  a) Always
  b) Never
  c) It depends on the hardware architecture
  d) It is impossible to know

When the code2 size becomes larger than L1 instruction cache size, code2 is no longer efficient

```
//code1
N=1000000;
for (i=0; i < N; i++)
  A[i] = B[i];
```

```
//code2
N=1000000;
for (i=0; i < N; i+=10000) {
  A[i] = B[i];
  A[i+1] = B[i+1];
  A[i+2] = B[i+2];
  A[i+3] = B[i+3];
        ...
   A[i+9999] = B[i+9999];
}
```



Use as less complex operations as possible (1)

- **Division is expensive**
  - On most CPUs the division operator is significantly more expensive (i.e. takes many more clock cycles) than all other operators. When possible, refactor your code to not use division.
  - Use multiplication instead
  - For example, change ***' / 5.0 '*** to ***' * 0.2 '***

- Use shift operations instead of multiplication and division
  - Only for multiplications and division with powers of 2
  - Compilers will do that for you though

Use as less complex operations as possible (2)

- **Functions such as pow(), sqrt()** etc are expensive, so avoid them when possible
  - E.g., avoid calling functions such as strlen() all the time, call it once (x=strlen()) and then x++ or x-- when you add or remove a character.
- **Avoid Standard Library Functions**
  - Many of them are expensive only because they try to handle all possible cases
  - Think of writing your own simplified version of a function, if possible, tailored to your application
  - E.g., pow(a, b) function where b is an integer and b=[1,10]

Strength Reduction (1)

- Strength reduction is the replacement of an expression by a different expression that yields the same value but is cheaper to compute
- Most compilers will do this for you automatically by specifying '-O1' option

```
do i = 1, n
  a[i] = a[i] + c*i
end do
```

(a) original loop

- Normally, addition needs less CPU cycles than multiplication
- In each iteration c is added to T

```
T = c
do i = 1, n
  a[i] = a[i] + T
  T = T + c
end do
```

(b) after strength reduction

Strength Reduction (2)

- Some other examples

1. Bitwise AND is cheaper than reminder
   - Substitute $a = b \% power.of.2.number$ to
     $a = b \ \& \ (power.of.2.number-1)$
   - For example, $a = b \% 8$ -> $a = b \ \& \ 7$
2. Shift and add is cheaper than multiplication (not always though…)
   - Substitute $a = b * 33$ *to* $a = (b << 5) + b$
3. pow() is very time consuming
   - Substitute $a = pow(b, 2.0)$ *to* $a = b * b;$

An example

Bitwise AND is cheaper than reminder

- Substitute $a = b \ \% \ power.of.2.number$ to
  $a = b \ \& \ (power.of.2.number-1)$
- For example, $a = b \ \% \ 8$ -> $a = b \ \& \ 7$

Using modulo: if $b=15_{10}$, $b=1111_2$ *then* $15\%8=7$

Using AND: if $b=7_{10}$ *then* $(15 AND 7 = 7)$

$$
\begin{array}{rl}
 & 0000\ 1111_2 = 15_{10} \\
\text{AND} & 0000\ 0111_2 = 7_{10} \\
\hline
 & 0000\ 0111_2 = 7_{10}
\end{array}
$$

Dead Code Elimination

- Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated
- **Compilers will do this for you automatically by specifying '-O1' option**

```
int main() {

int i , j;
double tmp=3.234;

printf("\nHello World");

return 0;

}
```

```
int main() {

printf("\nHello World");

return 0;

}
```

Common Subexpression Elimination

- Applicable when an expression has been previously computed and the values of the operands have not been changed since then
- The value of the previous computation is used (t1) and thus we don't re-compute the expression (x+y)
- **Compilers will do this for you automatically by specifying '-O1' option**

```
i = x + y + 1;
j = x + y;
```

⇒

```
t1 = x + y;
i = t1 + 1;
j = t1;
```

Use the appropriate precision

- Many technical computing algorithms require double precision instead of single precision
  - 64 bits vs. 32 bits or equivalently 8bytes vs. 4 bytes
- **When selecting data types for a given algorithm, or even a sub-part of the algorithm, try to determine if single precision is adequate**
- Single precision will consume half the amount of memory
  - Loading/storing will be faster
  - If the CPU supports 64-bit operations, the 64-bit arithmetic operations will not execute slower than the 32-bit ones though – performance will be the same

- If the target CPU is 16-bit, then short int operations will be more efficient than the 32-bit ones

Choose a better algorithm

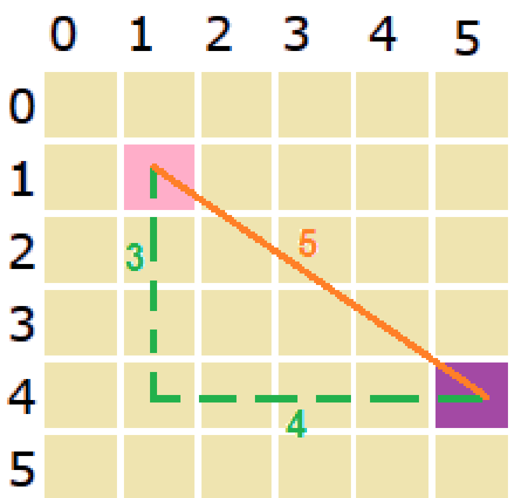
- Think about what the code is *really* doing
- Learn and use the most appropriate algorithms

- Some examples:
  - Linear search which gives O(n) complexity vs binary search which gives O(logn) complexity
  - Arrays vs linked lists
  - Euclidian distance vs Manhattan distance
  - Instead of applying Matrix-Vector multiplication for a Toeplitz matrix (all elements that belong to the same diagonal have identical values), we can use three FFTs and one vector multiplication instead

Choose a better algorithm (2)

For example

Euclidian distance vs Manhattan distance



Point P1 = (1,1)

Point P2 = (5,4)

Euclidean distance = $\sqrt{(5-1)^2 + (4-1)^2} = 5$

Manhattan distance = $|5-1| + |4-1| = 7$

Loop-Invariant Code Motion

- Any part of a computation that does not depend on the loop variable and which is not subject to side effects can be moved out of the loop entirely
- **Most compilers will do this for you automatically by specifying '-O1' option**

```
do i = 1,n
  a[i] = a[i] + sqrt(x)
end do
```

(a) original loop

- The value of sqrt(x) is not affected by the loop
- Therefore, its value is computed just once, outside of the loop

```
if (n > 0) C = sqrt(x)
do i = 1,n
  a[i] = a[i] + C
end do
```

(b) after code motion

- If n<1, the loop is not executed and therefore C must not be assigned with the sqrt(x) value

Use a table lookup if possible

- There are cases where some of the computation can be applied 'off-line' (before the program starts)
- Consider the following example
  - Just 7 values of sqrt() are computed N*N times, i.e., sqrt(2-8)
  - Their values can be found at compile time – before the program starts
  - Why not to compute them and store them into an table (array) ?
    - This will save a large number of CPU cycles
    - However, if the array is very large, then we must consider the extra cost of loading these values

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++) {
   A[i][j]=0.0;
   for (k=2; k<9; k++) {
     A[i][j] += i * sqrt(k) + j * sqrt(k);
   }
 }
```

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++) {
   A[i][j]=0.0;
   for (k=2; k<9; k++) {
     A[i][j] += i * B(k) + j * B(k);
   }
 }
```

Function Inline

- Replace a function call with the body of the function
- It can be applied in many different ways
  - Either manually or automatically
  - '-O1' applies function inline
  - In C, a good option is to use macros instead (if possible)
- **Pros :-**

  1. It speeds up your program by avoiding function calling overhead
  2. It saves the overhead of pushing/poping on the stack
  3. It saves overhead of return call from a function
  4. It increases locality of reference by utilizing instruction cache
- **Cons**
  - The main drawback is that it increases the code size

Loop Unswitching

- A loop containing a loop-invariant IF statement can be transformed into an IF statement containing two loops
- After unswitching, the IF expression is only executed once, thus improving run-time performance
- After unswitching, the loop body does not contain an IF condition and therefore it can be better optimized by the compiler
- **Most compilers will do this for you automatically by specifying '-O3' option**

```
for (i = 0; i < N; i++) {
  if (x<0)
    a[i] = 0;
  else
    b[i] = 0;
}
```

```
if (x<0)
  for (i = 0; i < N; i++) {
    a[i] = 0;
  }
else
  for (i = 0; i < N; i++) {
    b[i] = 0;
  }
```

Inline Assembly

- Write inline assembly code
  - Remember 90% of the execution time is spent on 10% of the code, in loops
  - Writing assembly code is hard, but writing 10 lines of inline assembly code is not that hard
  - You can write assembly code inside a loop by using the following command for each instruction *__asm__(" … ")*

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {  //C code
      __asm{
            //assembly code
           }
} }
```

Other Optimizations

1. **Avoid unnecessary copying** - Passing by reference

2. **Avoid writing if conditions inside loops**
   - Disables other optimizations

3. **Use switch instead of if conditions** (when many cases exist)
   - A switch statement might be faster than ifs provided number of cases is more than 5
   - If a switch contains more than five items, it's implemented using a lookup table
   - This means that all items get the same access time, compared to a list of ifs where the last item takes much more time to reach as it has to evaluate every previous condition first.

Conclusions

- This week you have learned some important code optimization techniques
- These techniques are widely used
- Most of the techniques are used by both the compiler (by specifying the appropriate optimization level) and the developers
  - However, compilers are not that advanced yet…
  - Normally, by applying optimizations manually, higher performance is achieved
  - Techniques such as using the appropriate algorithm, data type and table lookups are not used by compilers

Next Week

- More advanced Code Optimizations
  - Register Blocking
  - Loop interchange
  - Case study – Matrix Vector Multiplication
- Case study #2 – Matrix-Matrix Multiplication
  - More in depth analysis