Datový paralelismus SIMD

AVS – Architektury výpočetních systémů

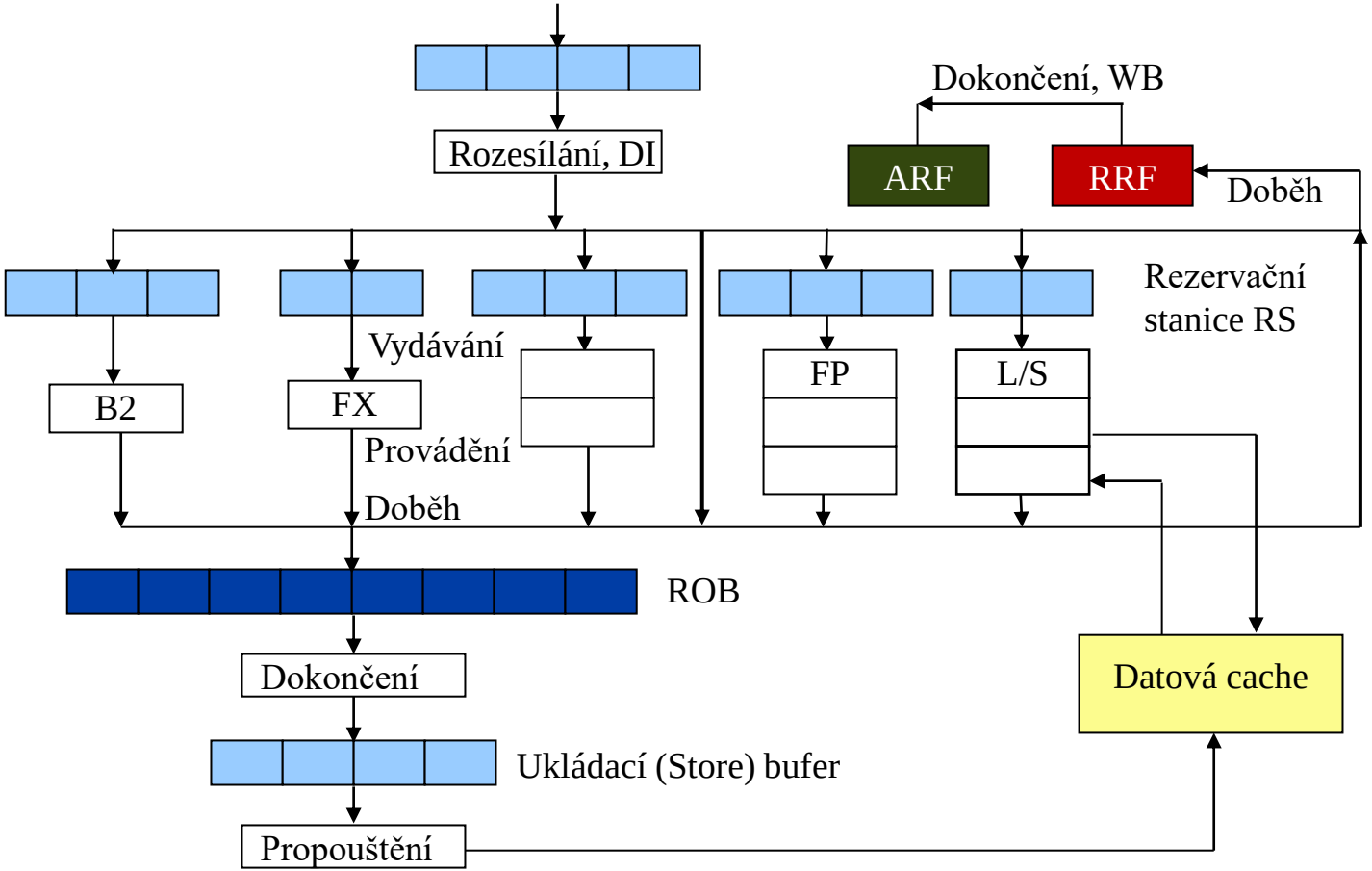
Týden 5, 2024/2025

Jirka Jaroš

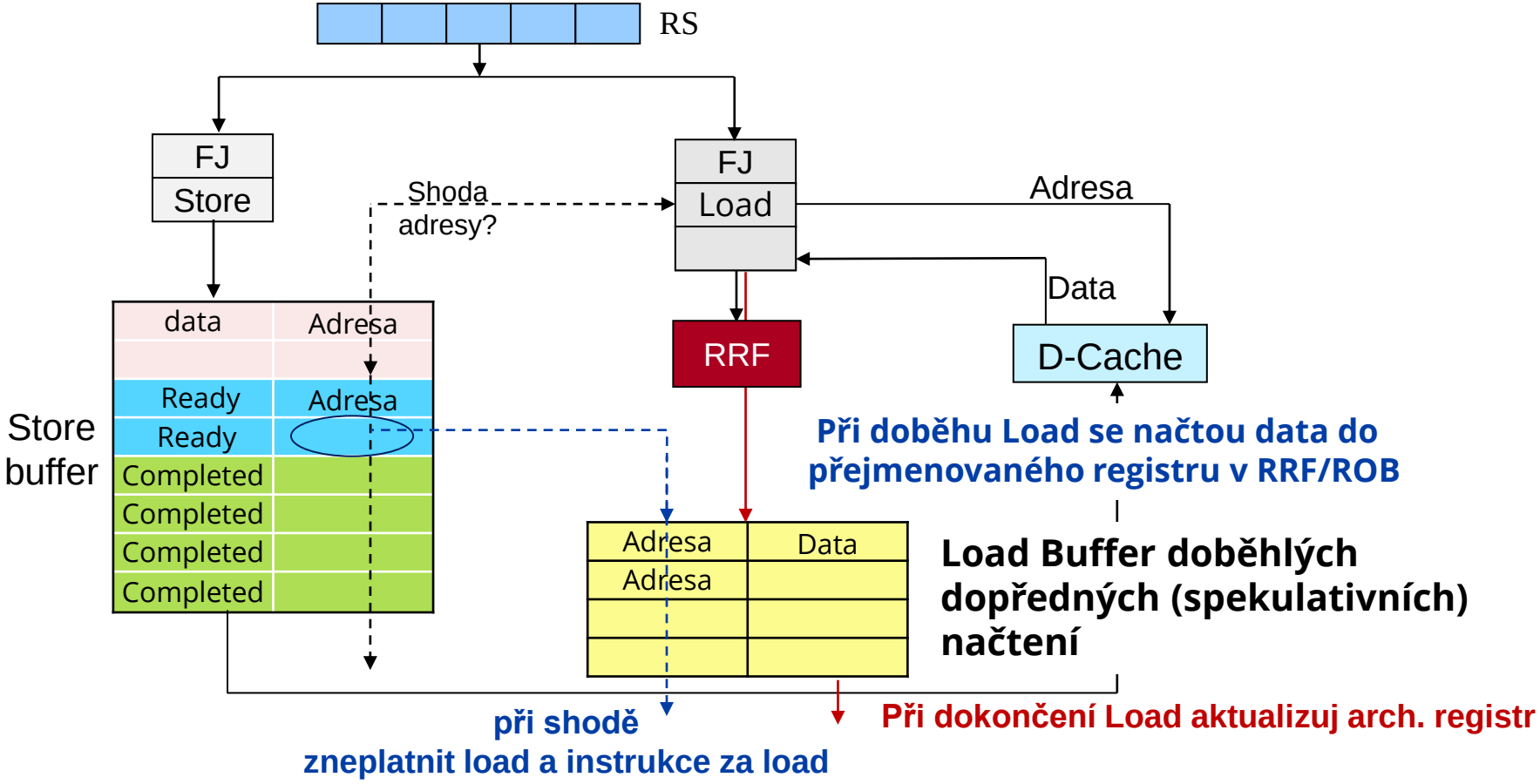
Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole
jarosjir@fit.vutbr.cz

OPAKOVÁNÍ

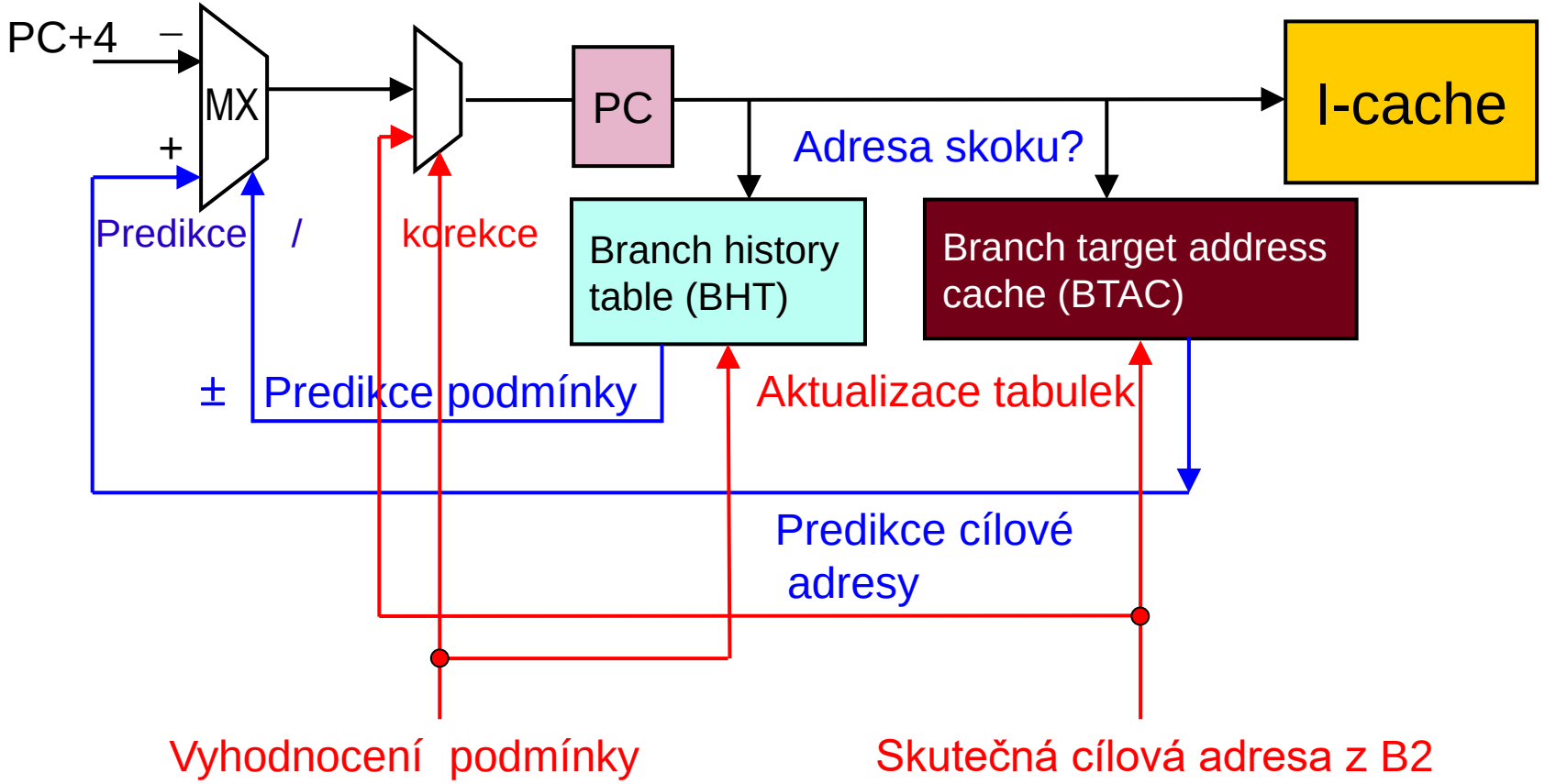
Superskalární procesor – Backend



Out-of-order Load/Store jednotka

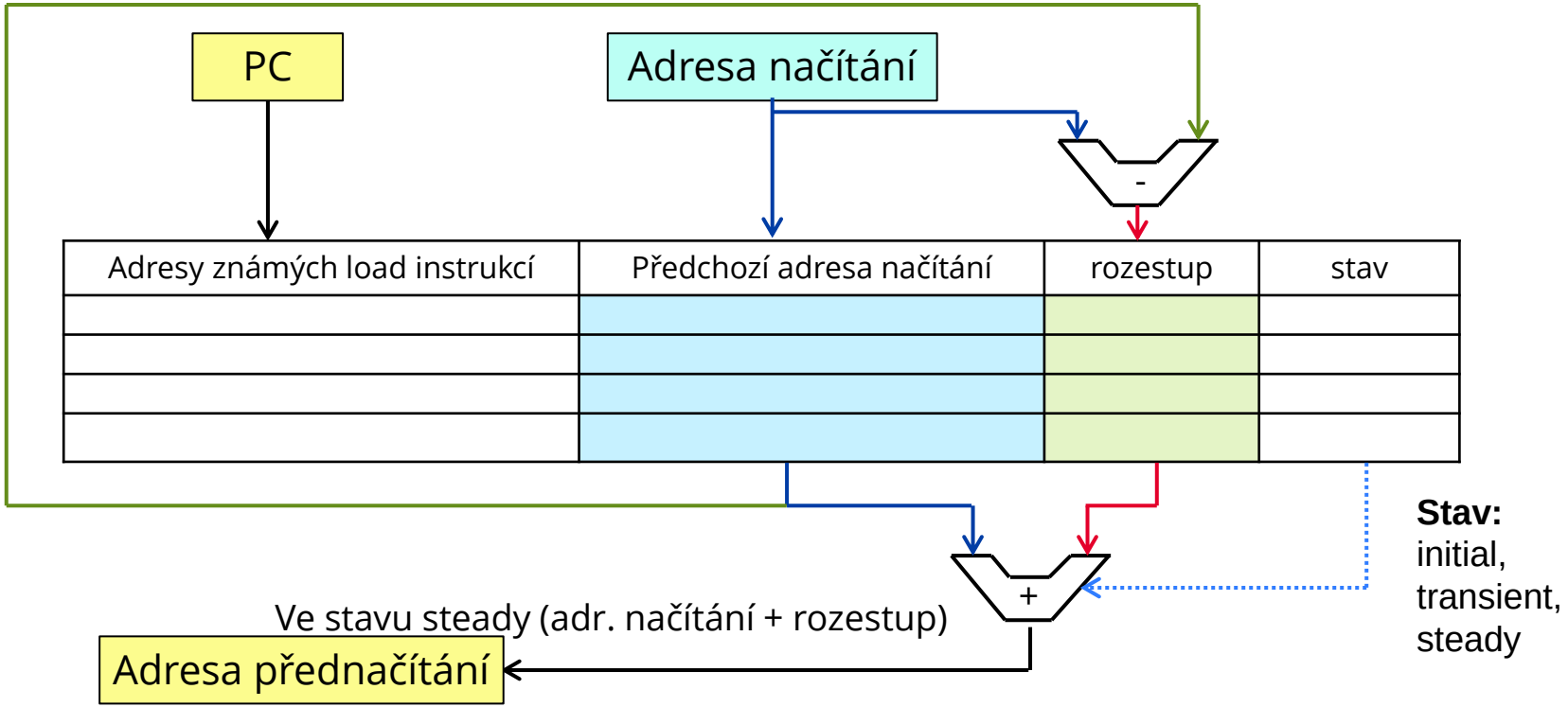


Predikce podmínky a cílové adresy skoku (B1)



Přednačítání operandů s rozestupem (HW)

- Load History Table (LHT) = malá cache již provedených instrukcí L
- Rozestup se získá ze dvou L na téže adrese.



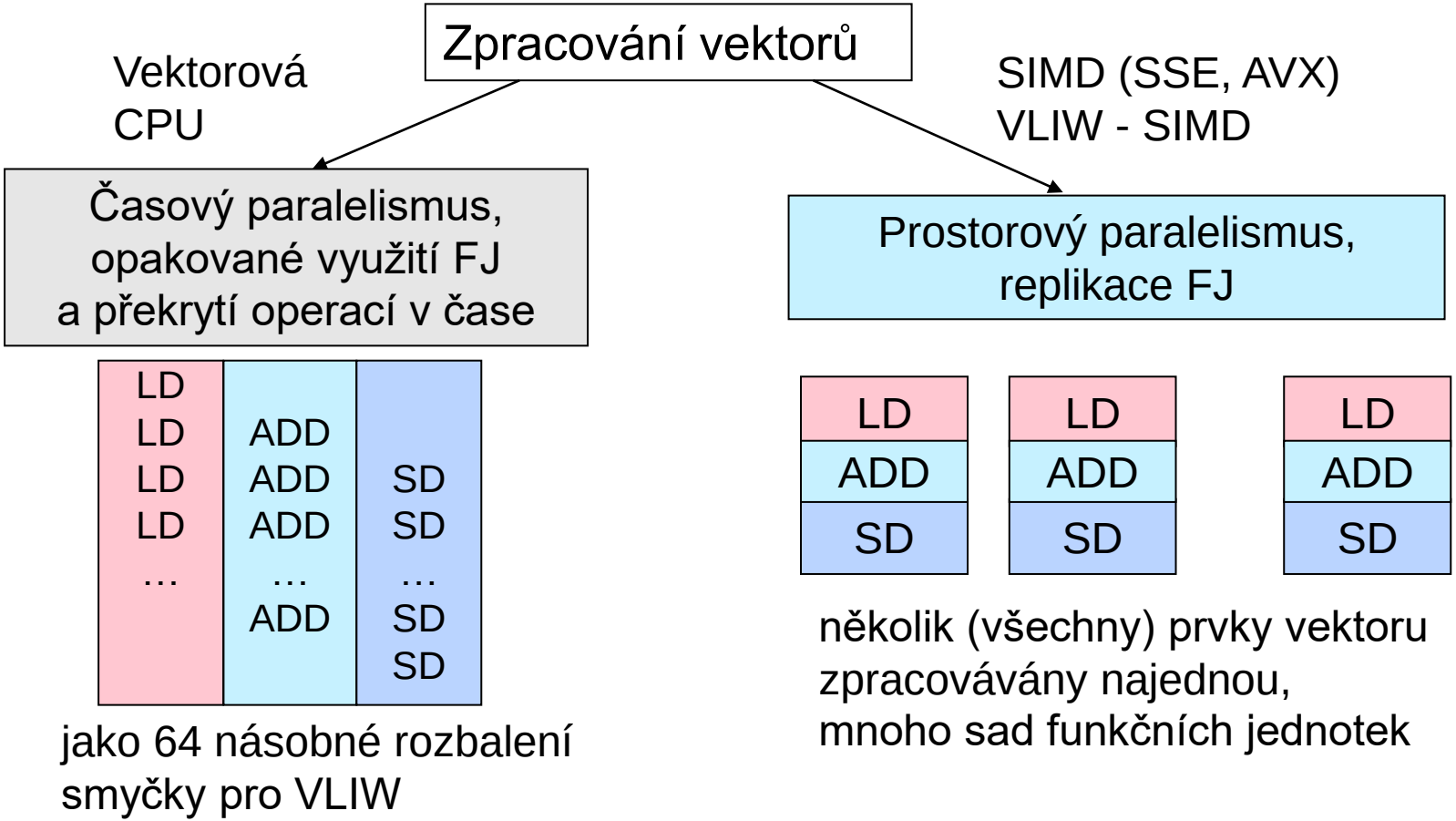
DATOVÝ PARALELISMUS

Paralelismus v procesorech

1. **Funkční paralelismus** využívá obecně možností paralelního provádění funkcí v jemné (ILP)
   - Superskalární procesory – ILP řízen pomocí hardware (OOO, pipeline)
   - Procesory s dlouhým instrukčním slovem VLIW – ILP řízen software (kompilátor)

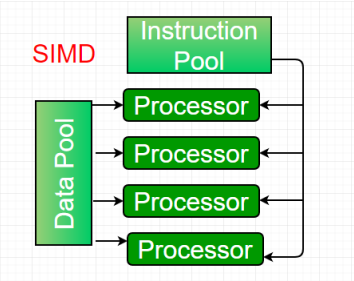
   nebo hrubé granularitě (vlákna – TLP, procesy).
2. **Datový paralelismus** využívá možnosti provádět stejné operace s mnoha nezávislými datovými položkami
   - v čase řetězeným zpracováním (vektorové procesory)
   - v prostoru na replikovaných funkčních jednotkách (multiprocesory SIMD, SIMD Within A Register = SWAR)
   - v obou dimenzích (paralelní vektorové linky nebo technologie Single Instruction Multiple Threads = SIMT)

Datový paralelismus – zpracování více elementů v 1 instrukci

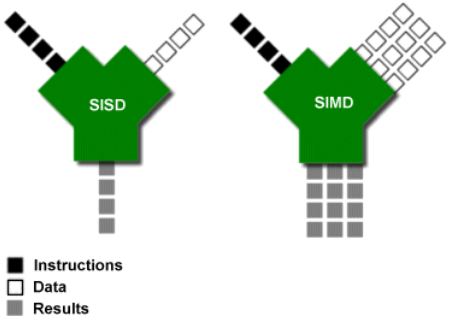


Datový paralelismus a SIMD

- SIMD – Single Instruction Multiple Data
  - Soubor procesorů řízený centrální jednotkou pracuje synchronně po instrukcích - všechny procesory dělají totéž
  - Některé procesory mohou stát (NOP)



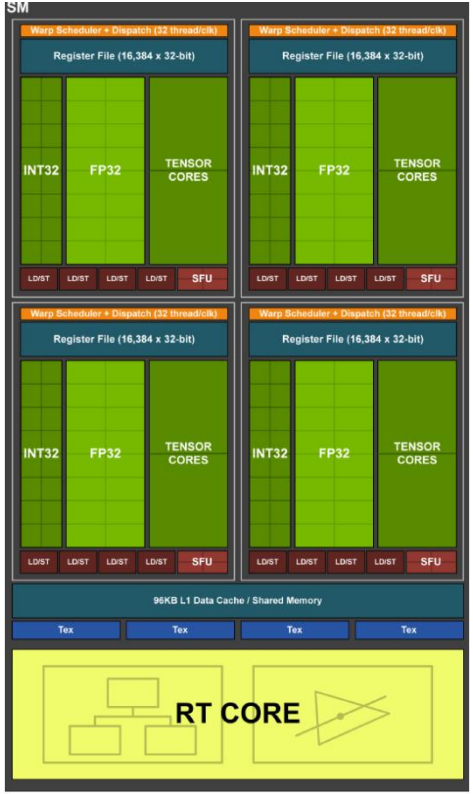
- Výpočet jednotlivých prvků probíhá nezávisle na ostatních
- **Silná stránka:**
  - Zpracování polí, matic, trenzorů, seznamů
- **Slabá stránka:**
  - Podmíněné sekce a příkazy `switch`.
    N alternativ se provádí v podmnožinách procesorů sekvenčně.



- Při mapování algoritmů na tyto architektury je nezbytné změnit způsob uvažování.

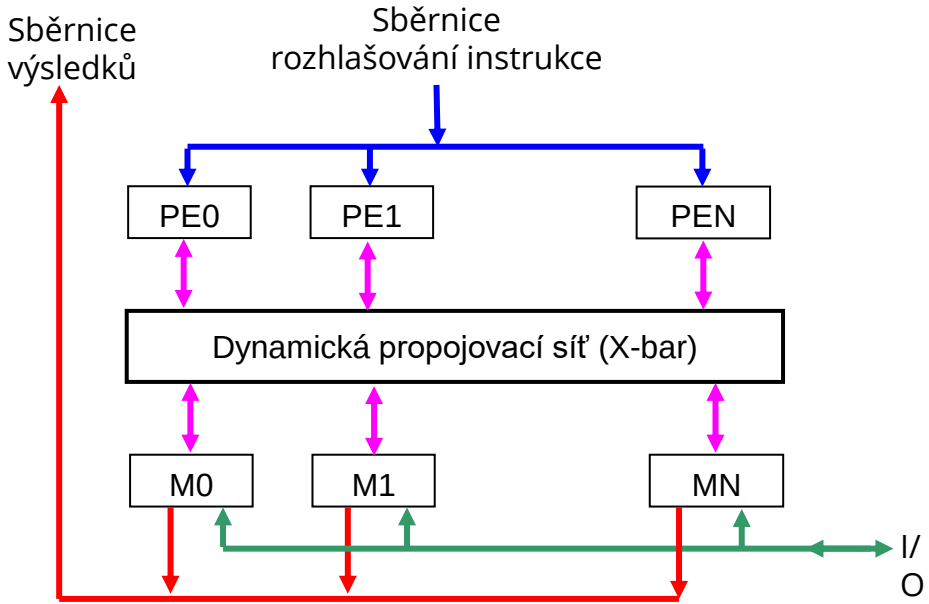
SIMD Architektury

- SIMD architektury obsahují velký počet výpočetních jednotek (Processing Units PE) a řídící procesor.
- Počet PE jednotek na CPU a GPU v rozsahu 4–32.
- SIMD architektury vyžadují méně HW prostředků než MIMD (pouze jedna řídící jednotka).
- Základní problém: Vhodný poměr mezi složitostí PE jednotky a jejich počtem.
- Častý kompromis: Velký počet základních jednotek podporujících operace (ADD, SUB, CMP, MUL, …) doplněných o několik málo specializovaných jednotek realizujících speciální operace (DIV, SIN, LOG, SQRT, …)
- Nejsou považovány za univerzální výpočetní systémy; pro některé testy špičková výkonnost, pro jiné jen nízká.
- **Použití jako HW akcelerátory a koprocesory (GPU, AVX)**



Rozdělení SIMD architektur

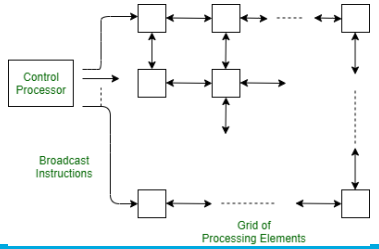
- SIMD se sdílenou pamětí
  - Paměťové moduly sdíleny všemi PE elementy
  - Čtení i zápis dat probíhá skrze dynamickou propojovací síť (např. křížový přepínač)
  - Jednotlivé PE elementy komunikují pouze přes sdílenou paměť
  - Současné architektury AVX jednotek v procesorech a SM procesorů v GPU se nejvíce podobají tomuto modelu

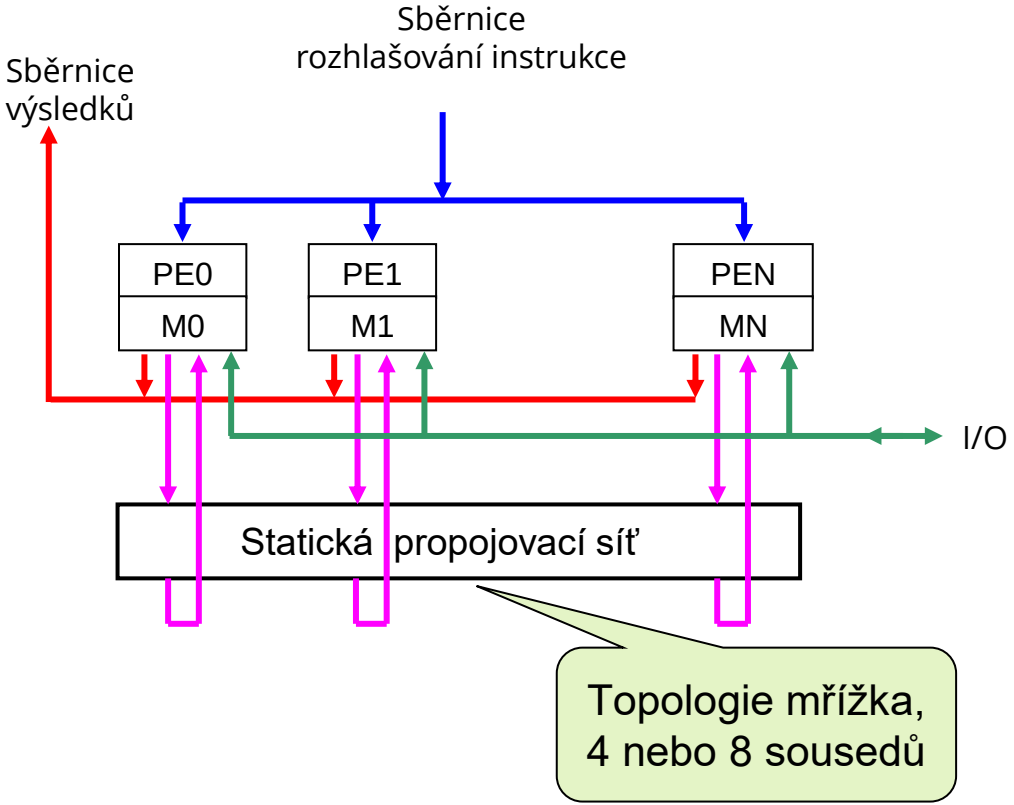


Nejvíce připomíná model CREW PRAM

Rozdělení SIMD architektur

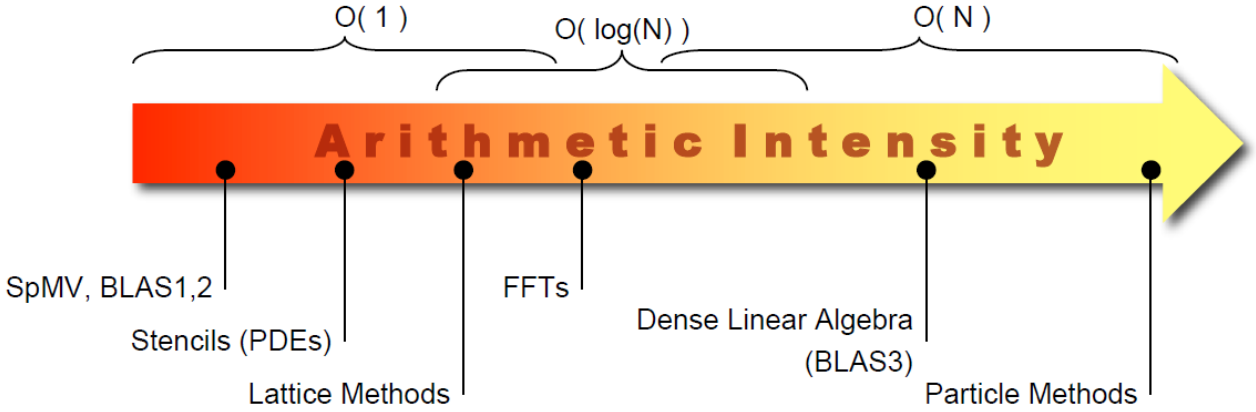
- SIMD s distribuovanou pamětí
  - Každý PE má svůj vlastní paměťový modul – většinu výpočtu provádí nad lokálními daty.
  - Komunikace resp. výměna dat s ostatními PE elementy probíhá skrze statickou propojovací síť (např. mřížka, torus, popř. složitější topologie).
  - Použito např. v IBM Cell procesory (Sony Playstaion 3)





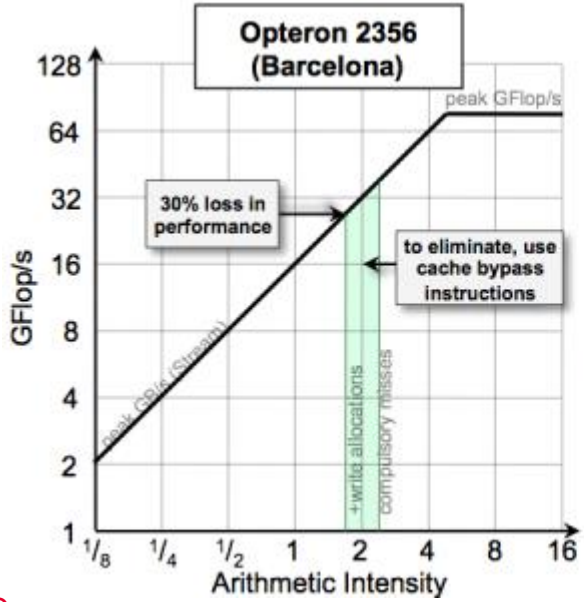
Kdy má smysl přemýšlet o SIMD a vektorizaci?

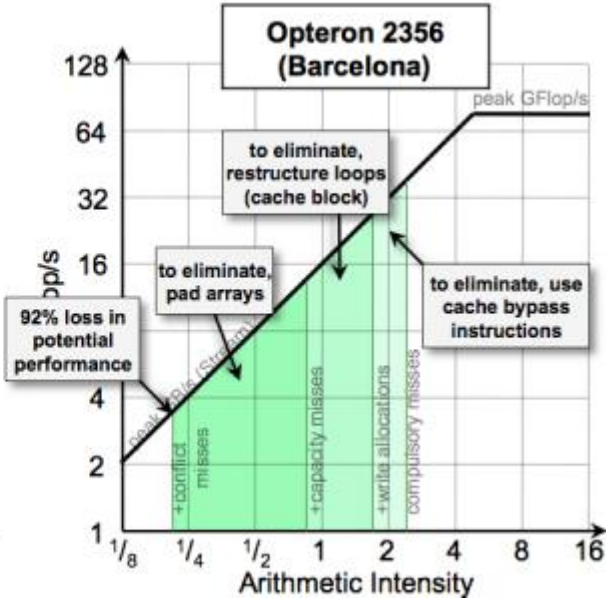
- **Výkonnost každého algoritmu je shora omezena buď propustností (FP) ALU (GFLOP/s) nebo paměti (GB/s).**
- I **L1** je stěží schopna zásobovat žravé AVX jednotky a zabránit vkládání NOP.
- Pokud nedosáhneme **rozumné aritmetické intenzity** (FLOP:B), nemá smysl se do vektorizace vůbec pouštět.
- **SIMD-friendly** algoritmus tedy musí maximálně vyžívat pamětí cache (cache blocking), efektivně přednačítat data z RAM, minimalizovat výpadky v TLB, …



Roofline model výkonnosti

- **Roofline popisuje výkon algoritmu na základě**
  - Aritmetické intenzity
  - Propustnosti paměti
  - Výpočetního výkonu
- **Výpadky v paměti cache mají velký vliv na aritmetickou intenzitu**

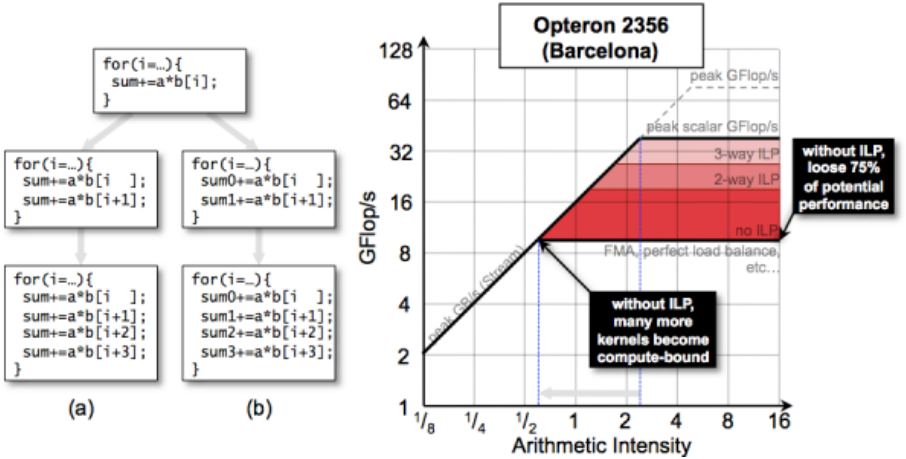




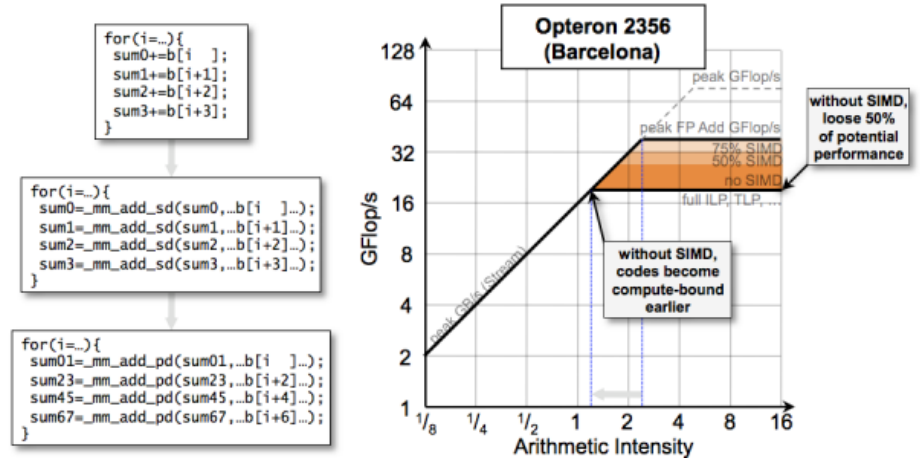
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

Roofline model výkonnosti

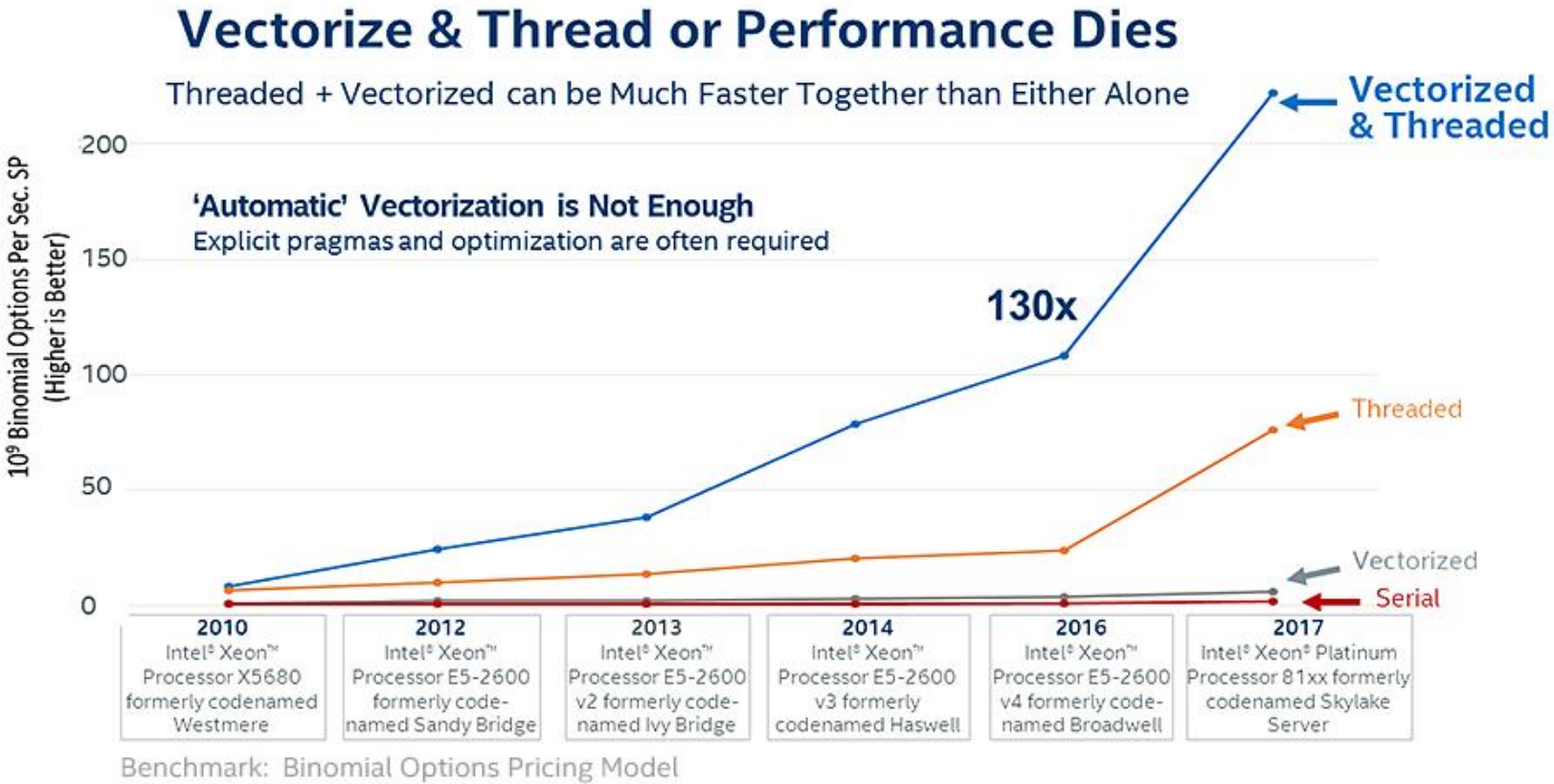
ILP



SIMD



Max výkon procesoru

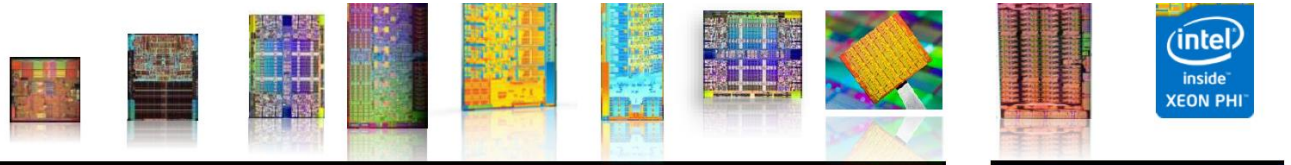


https://www.intel.com/content/www/us/en/developer/videos/vectorize-or-performance-dies-tune-for-the-latest-avx-simd.html

SIMD UVNITŘ REGISTRŮ SWAR

Historie vektorových instrukcí v procesorech x86

1999	2000	2004	2006	2007	2008	2009	2011	2012\2013	2012
SSE	SSE2	SSE3	SSSE3	SSE4.1	SSE4.2	AES-NI	AVX	AVX2	MIC
70 instr Single-Precision Vectors Streaming operations	144 instr Double-precision Vectors 8/16/32 64/128-bit vector integer	13 instr Complex Data	32 instr Decode	47 instr Video Graphics building blocks Advanced vector instr	8 instr String/XML processing POP-Count CRC	7 instr Encryption and Decryption Key Generation	~100 new instr. ~300 legacy sse instr updated 256-bit vector 3 and 4-operand instructions	Int. AVX expands to 256 bit Improved bit manip. fma Vector shifts Gather	512-bit vector

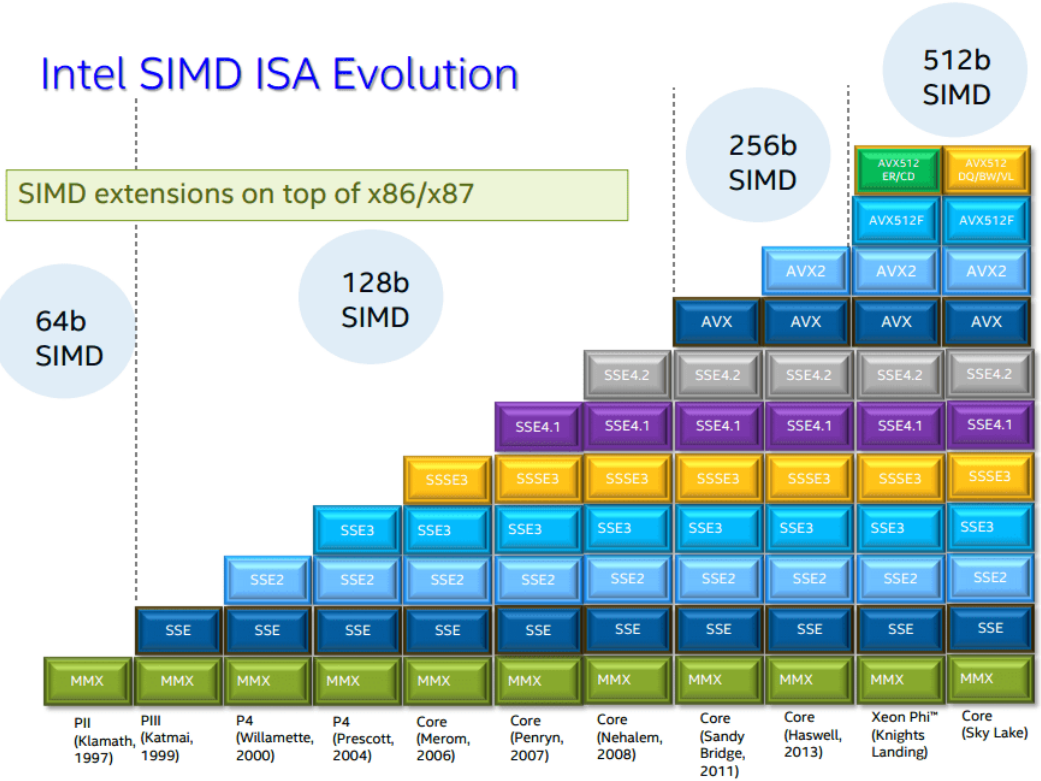


	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EP	Intel® Xeon Processor codenamed Skylake EP	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing[1]
Core(s)	1	2	4	6	8	12	18	28	61	70+
Threads	2	2	8	12	16	24	36	56	244	280+
SIMD Width	128	128	128	128	256	256	256	512	512	512

Současnost vektorových instrukcí

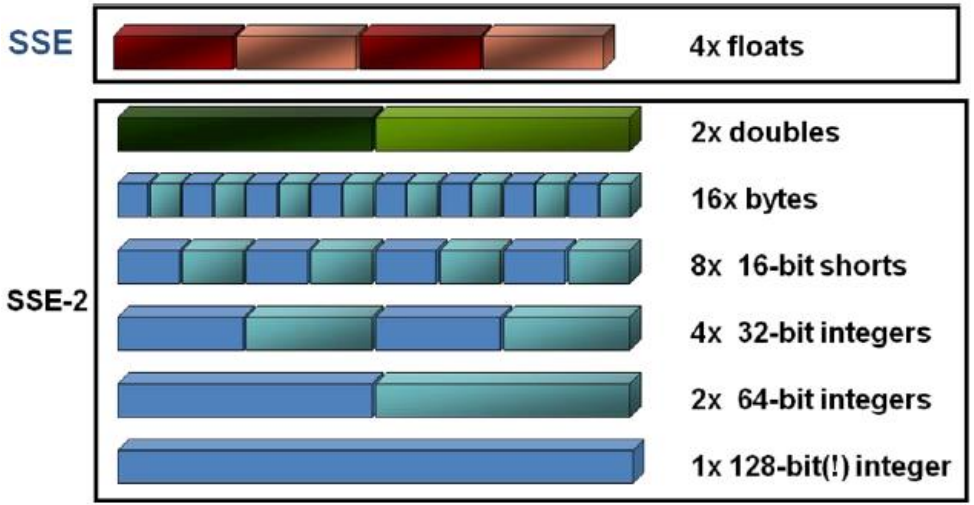
AVX-512 instrukce jsou rozdělenu do několika skupin

- **Podpora na CPU i MIC (Xeon Phi)**
  - Foundation Instructions (AVX512-F)
  - Conflict Detection Instructions (AVX512-CD)
- **Podpora pouze na MIC**
  - Exponential and Reciprocal Instruction (AVX512-ER)
  - Prefetch Instructions (AVX512-PF)
- **Podpora pouze na CPU**
  - Byte (char/int8) and Word (short/int16) Instructions (AVX512-BW)
  - Double-word (int32/int) and Quad-word (int64/long) Instructions (AVX512-DQ)
  - Vector Length Extensions (AVX512-VL)

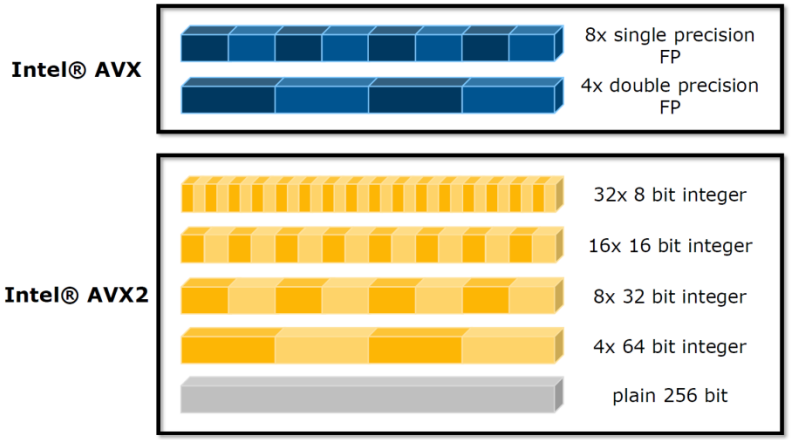


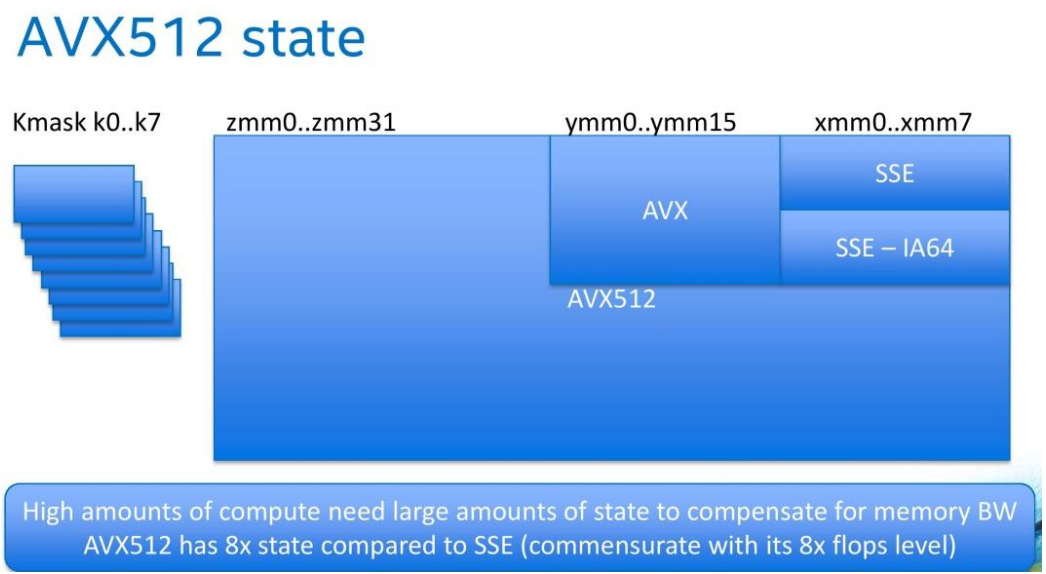
Registry SSE

- Registry SSE již nejsou mapovány na registry FPU jako registry MMX.
- SSE obsahuje 8 x 128b registrů (XMM0–XMM7)
- Jelikož tyto registry nejsou v původní sadě x86, OS nemá o jejich existenci tušení => nelze uložit jejich stav.
- Proto Intel modifikoval chráněný režim procesoru a vytvořil tzv. rozšířený mód procesoru (Enhanced mode), kde jsou registry SSE již viditelné pro OS.
- Pokud je OS schopen pracovat s registry SSE, musí přepnout procesor do rozšířeného módu.

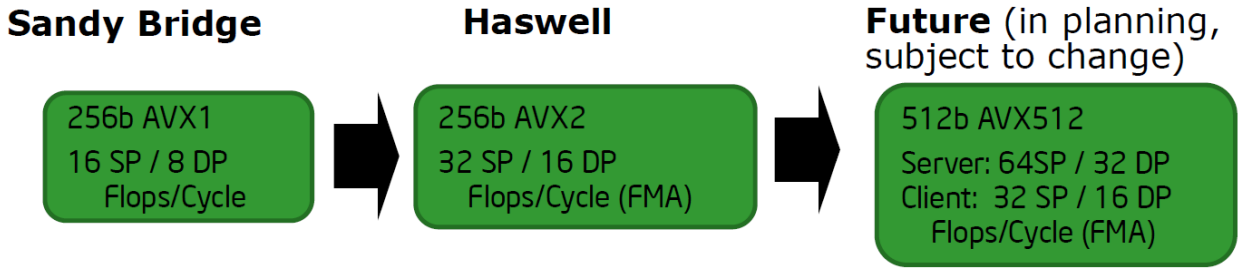


Registry AVX a AVX-512





Instrukční sady AVX a AVX-512



AVX	AVX2
256-bit basic FP	Float16 (IVB 2012)
16 registers	256-bit FP FMA
NDS (and AVX128)	256-bit integer
Improved blend	PERMD
MASKMOV	Gather
Implicit unaligned	

AVX512
512-bit FP/Integer
32 registers
8 mask registers
Embedded rounding
Embedded broadcast
Scalar/SSE/AVX “promotions”
Native media additions
HPC additions
Transcendental support
Gather/Scatter

SNB-2011 **HSW-2013** **Future Processor (Knight Landing & Skylake Xeon)**

SIMD zpracování – výhled do budoucnosti

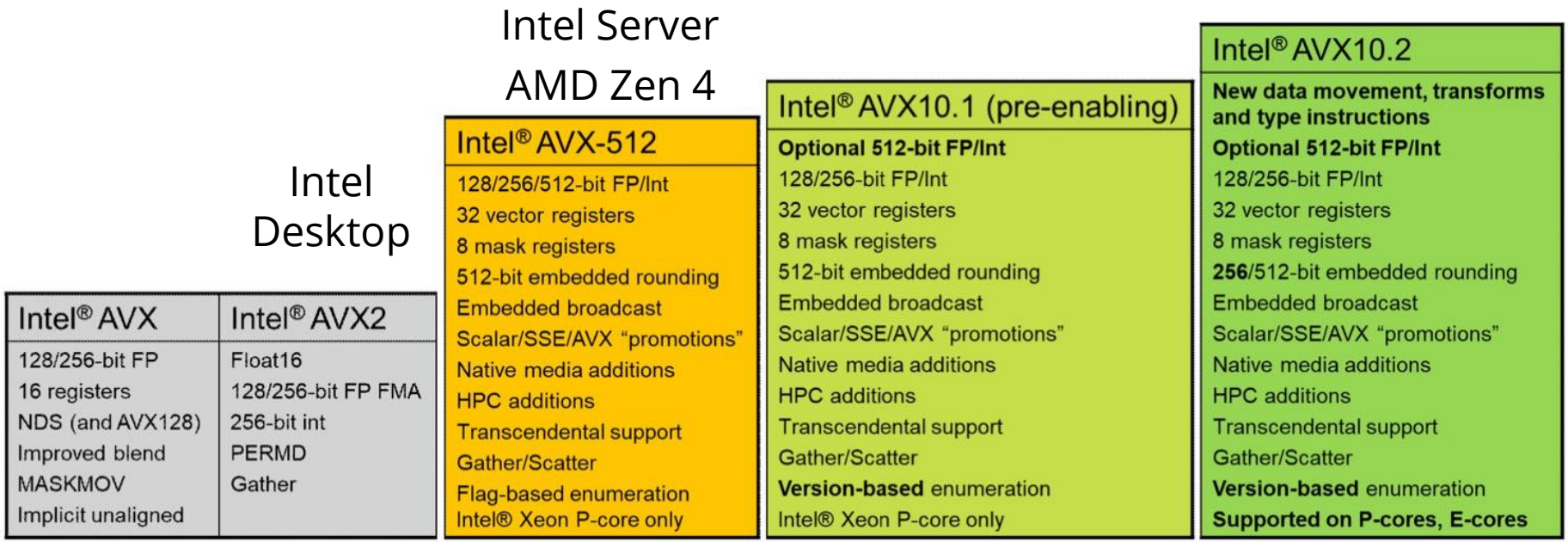


Figure 1-2. Intel® ISA Families and Features

AVX-512: Podpora nových instrukcí

- VADDPS ZMM0 {k1}, ZMM3, [mem]
  - Mask bits used to:
  1. *Suppress individual elements read from memory*
     - hence not signaling any memory fault
  2. *Avoid actual independent operations within an instruction happening*
     - and hence not signaling any FP fault
  3. *Avoid the individual destination elements being updated,*
     - or alternatively, force them to zero (zeroing)

```
for (I in vector length)
{
   if (no_masking or mask[I]) {
      dest[I] = OP(src2, src3)
   } else {
      if (zeroing_masking)
         dest[I]  = 0
      else
         // dest[I] is preserved
   }
}
```

Some instructions do no suppress memory exceptions as mask aligned to "out

AVX512 Masking

Gather & Scatter

D/Q/SP/DP element types
D/Q indices
Instruction can partially execute
k-reg Mask used as completion mask

```
for(j=0, i=0; i<N; i++)
{
    B[R[i]] = A[Q[i]];
}
```

```
VMOVDQU64 zmm1, Q[rsi]
VMOVDQU64 zmm2, R[rsi]
VGATHERQQ zmm0 {k2}, [rax+zmm1*8]
VSCATTERQQ [rax+zmm2*8] {k3}, zmm0
```

G/S implementation attempts to 'max out' DCU BW
Performance gains come from vectorizing REST of algorithm
Algorithm shown could get some gain (24 load dispatches → 10 per 8 elements)

Počet FP operací za takt u Haswelu

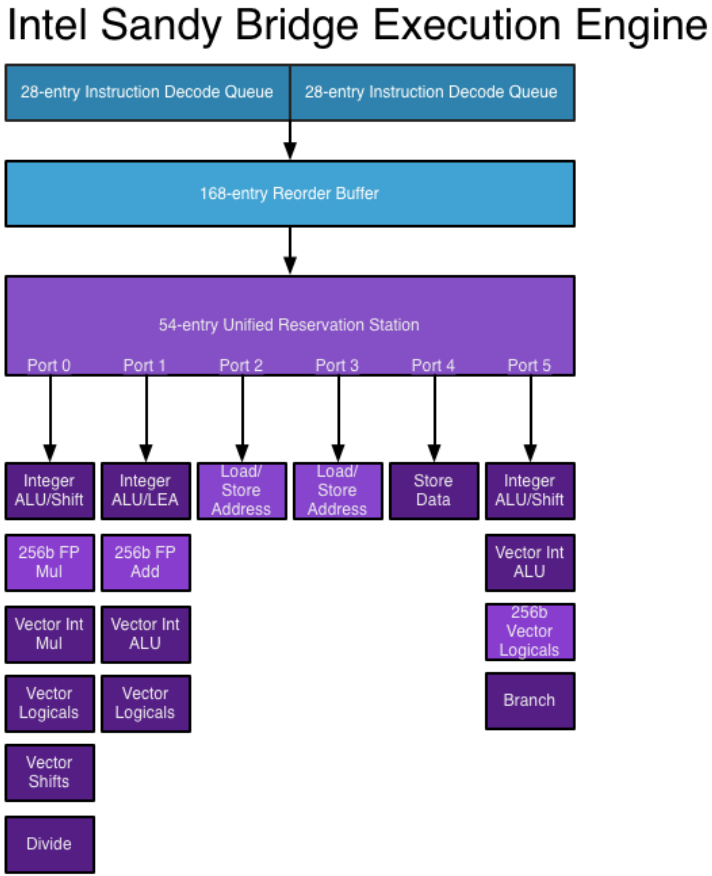
- Intel® Advanced Vector Extensions 2 (Intel® AVX2)
  - Includes
    - 256-bit Integer vectors
    - FMA: Fused Multiply-Add
    - Full-width element permutes
    - Gather
  - Benefits
    - High performance computing
    - Audio & Video
    - Games
- New Integer Instructions
  - Indexing and hashing
  - Cryptography
  - Endian conversion – MOVBE

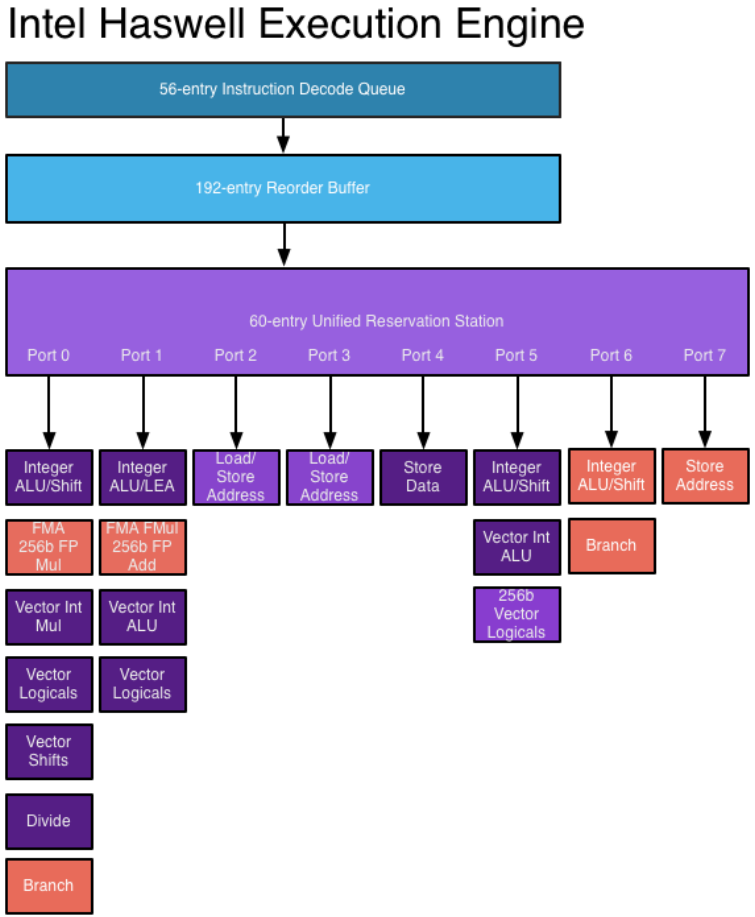
	Instruction Set	SP FLOPs per cycle	DP FLOPs per cycle
Nehalem	SSE (128-bits)	8	4
Sandy Bridge	AVX (256-bits)	16	8
Haswell	AVX2 & FMA	32	16

Group	Instructions
Bit Field Pack/Extract	BZHI, SHLX, SHRX, SARX, BEXTR
Variable Bit Length Stream Decode	LZCNT, TZCNT, BLSR, BLSMSK, BLSI, ANDN
Bit Gather/Scatter	PDEP, PEXT
Arbitrary Precision Arithmetic & Hashing	MULX, RORX

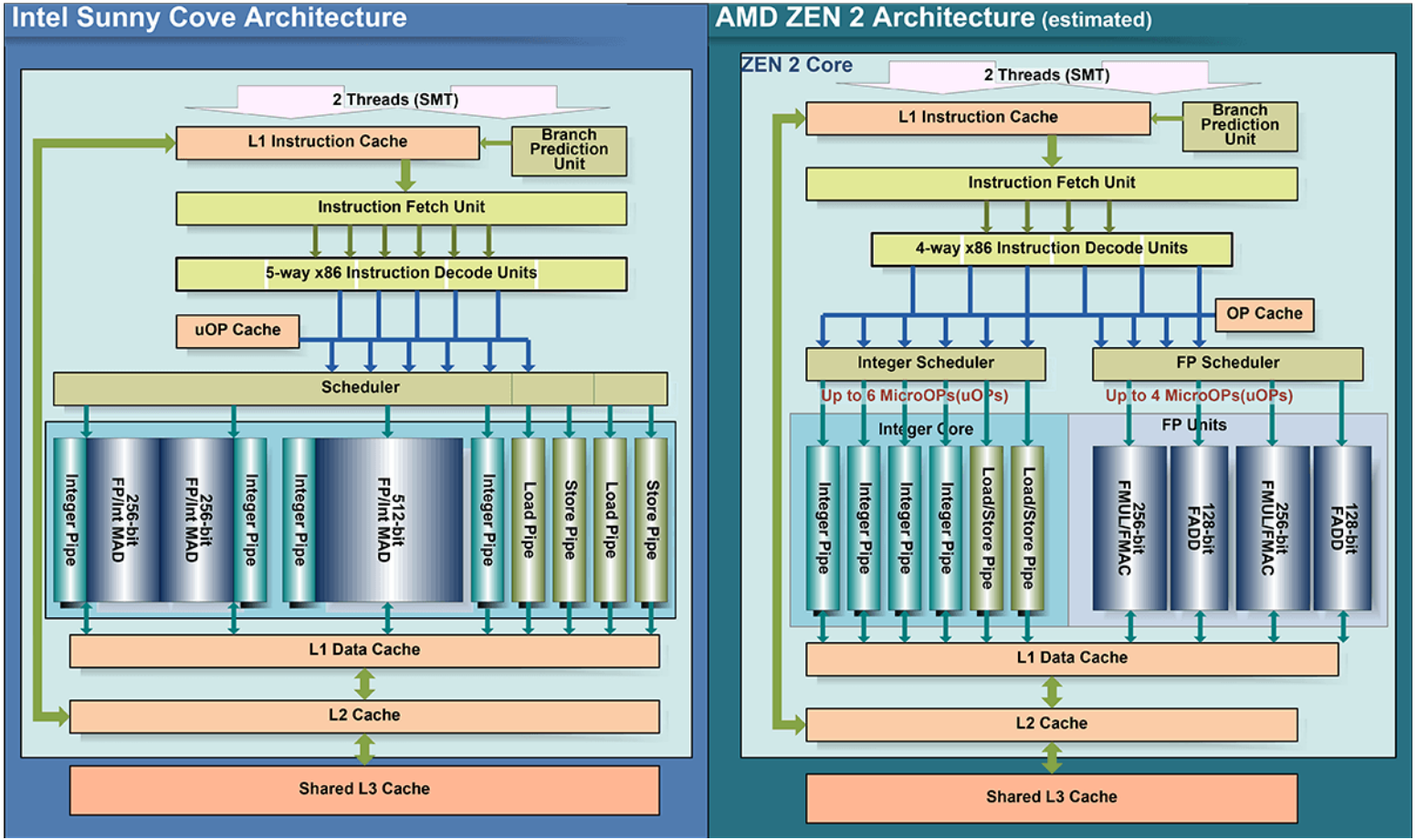
- Full Instruction Specification Available at: http://software.intel.com/en-us/avx/

Intel Sandy Bridge a Haswell backend





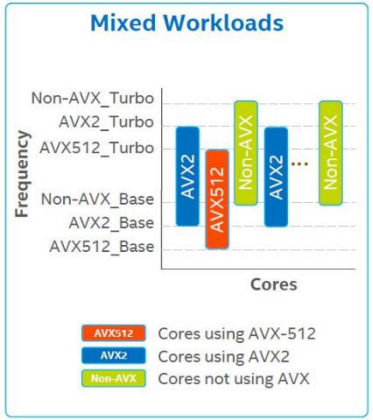
Intel vs AMD backend

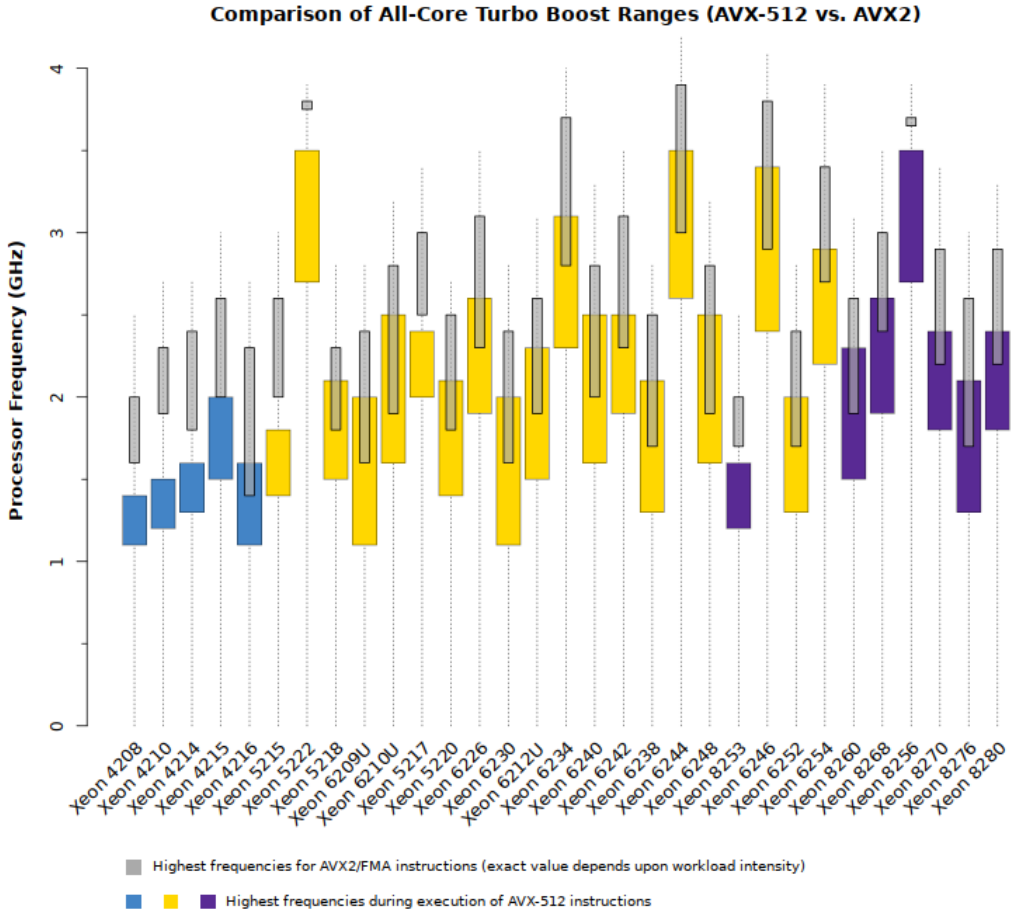


Snižování frekvence při použití AVX-512

- Cores running non-AVX, Intel® AVX2 light/heavy, and Intel® AVX-512 light/heavy code have different turbo frequency limits
- Frequency of each core is determined independently based on workload demand

Code Type	All Core Frequency Limit
SSE AVX2-Light (without FP & int-mul)	Non-AVX All Core Turbo
AVX2-Heavy (FP & int-mul) AVX512-Light (without FP & int-mul)	AVX2 All Core Turbo
AVX512-Heavy (FP & int-mul)	AVX512 All Core Turbo





VEKTORIZACE KÓDU

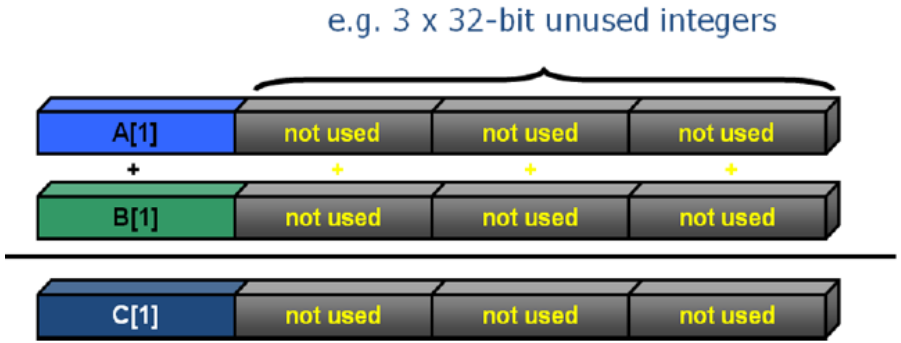
Jednoduchý příklad využití registrů SSE

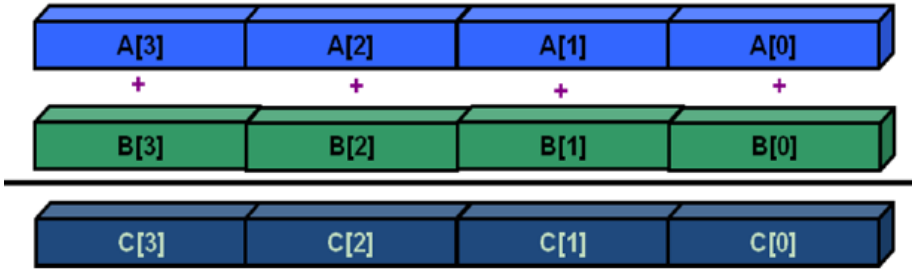
Bez vektorizace

```
for (i = 0; i < MAX; i++)
{
  c[i] = a[i] + b[i];
}
```

Vektorizováno

```
for (i = 0; i < MAX; i+=4)
{
  c[i:4] = a[i:4] + b[i:4];
}
```





Jak lze vektorizovat kód

Vektorizované knihovny

- Intel MKL
- Atlas, FFTW, TensorFlow, ...

Automatická vektorizace kompilátorem

- -O3 -vec

Pragma hinty kompilátoru

- OpenMP 4.0+
- Intel

Vektorové intrinsic funkce

- _mm_malloc
- __mm_add_ps

ASM kód

- addps

Automatická vektorizace kompilátorem

Rozbalení smyčky je spojeno s generováním SIMD instrukcí

```
static double A[1000], B[1000],
              C[1000];
void add() {
  int i;
  for (i=0; i<1000; i++)
    if (A[i]>0)
      A[i] += B[i];
    else
      A[i] += C[i];
}
```

```
.B1.2::
  movaps    xmm2, A[rdx*8]
  xorps     xmm0, xmm0
  cmpltpd   xmm0, xmm2
  movaps    xmm1, B[rdx*8]
  andps     xmm1, xmm0
  andnps    xmm0, C[rdx*8]
  orps      xmm1, xmm0
  addpd     xmm2, xmm1
  movaps    A[rdx*8], xmm2
  add       rdx, 2
  cmp       rdx, 1000
  jl        .B1.2
```

SSE2

```
.B1.2::
  vmovaps    ymm3, A[rdx*8]
  vmovaps    ymm1, C[rdx*8]
  vcmpgtpd   ymm2, ymm3, ymm0
  vblendvpd  ymm4, ymm1,B[rdx*8], ymm2
  vaddpd     ymm5, ymm3, ymm4
  vmovaps    A[rdx*8], ymm5
  add        rdx, 4
  cmp        rdx, 1000
  jl         .B1.2
```

AVX

```
.B1.2::
  movaps    xmm2, A[rdx*8]
  xorps     xmm0, xmm0
  cmpltpd   xmm0, xmm2
  movaps    xmm1, C[rdx*8]
  blendvpd  xmm1, B[rdx*8], xmm0
  addpd     xmm2, xmm1
  movaps    A[rdx*8], xmm2
  add       rdx, 2
  cmp       rdx, 1000
  jl        .B1.2
```

SSE4.1

- ✓ SSE2-SSE4.2
  - No native masked operations
  - "Masks" in vector registers for AND/OR blending
  - Memory operations and unsafe FP operations speculated or emulated
    - Via scalarization for memory
    - Blend w/ safe value for FP
  - Remainder vectorized unmasked
- ✓ AVX
  - vmaskmov is introduced
- ✓ AVX2
  - vgather is introduced
  - Peel/remainder vectorized unmasked
- ✓ Intel® MIC Architecture
  - Native masking in dedicated registers
  - For all operations
  - May be unsafe if masked-out memory is not paged
    - Except gathers/scatters
    - Safety Ifs inserted for empty masks
    - Some operations done through gather
  - Peel/remainder/low-trip vectorized in masked mode

https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

Podpora SIMD rozšíření Intel kompilátoru -xSADA

SIMD Sada	Popis
COMMON-AVX512	May generate Intel AVX-512 Foundation instructions, Intel AVX-512 Conflict Detection instructions, AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2 and SSE instructions for Intel processors.
MIC-AVX512	May generate Intel AVX-512 Foundation instructions, AVX-512 Conflict Detection instructions, AVX-512 Prefetch instructions, AVX-512 Exponential and Reciprocal instructions, AVX2, AVX, SSE4.2 - SSE instructions for Intel processors.
CORE-AVX512	May generate AVX-512 Foundation instructions, AVX-512 Conflict Detection instructions, AVX-512 Doubleword and Quadword instructions, AVX-512 Byte and Word instructions, AVX-512 Vector Length extensions, AVX2, AVX, SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors.
CORE-AVX2	May generate Intel AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2 and SSE instructions for Intel® processors.
CORE-AVX-I	May generate Intel AVX, SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors,
AVX	May generate Intel AVX, SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel
SSE4.2	May generate Intel SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors.
ATOM_SSE4.2	May generate Intel SSE4.2, SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors.
SSE4.1	May generate Intel SSE4.1, SSE3, SSE3, SSE2 and SSE instructions for Intel processors.
SSSE3	May generate Intel SSE3, SSE3, SSE2 and SSE instructions for Intel processors.
ATOM_SSSE3	May generate SSE3, SSE3, SSE2 and SSE instructions for Intel processors.
SSE3	May generate Intel SSE3, SSE2 and SSE instructions.
SSE2	May generate Intel SSE2 and SSE instructions.
HOST	May generate instructions from any of the above instruction sets that are supported by the compilation host processor.

https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations

Základní přepínače pro vektorizaci (Intel)

- **Linux, MacOS X: -x<feature>, Windows: /Qx<feature>**
  - Může zapnout specifické optimalizace pro procesory Intel.
  - Do hlavní funkce (main) je přidán test, který vypíše chybu v případě, že daný procesor nepodporuje některé funkce.
- **Linux, MacOS X: -m<feature>, Windows: /arch:<feature>**
  - Test na intel procesory vypnut, stejně tak specifické Intel optimalizace.
  - Lze tedy použít i pro procesory AMD.
  - Pokud procesor nepodporuje nějakou funkci, vyhodí výjimku Illegal instruction.
- **Linux, MacOS X: -ax<feature>, Windows: /Qax:<feature>**
  - Přeloží se několik variant: baseline a procesorově specifická.
  - Takto lze kompilovat pro různé instrukční sady, např. –axSSE2,AVX, CORE-AVX2
  - Baseline varianta je dnes stále –msse2 (/arch:sse2). Prakticky nemá smysl používat nic staršího než AVX.
- **Linux, MacOS X: -aHost, Windows: /QxHost**
  - Přeloží a optimalizuje pro procesor na kterém se překládá.

Optimalizační tipy od Intel kompilátoru (icpc)

- **Nastavení úrovně detailů, které generuje Intel kompilátor**
  - `/Qopt-report[0|1|2|3]` (Windows)
  - `-opt-report[0|1|2|3]` (Linux, MacOS)
- **Nastavení fází, které nás zajímají**
  - `/Qopt-report-phase[:phase]` (Windows)
  - `-opt-report-phase[=phase]` (Linux, MacOS)
    - `ipo_inl` – Interprocedural Optimization Inlining Rep
    - `ilo` – Intermediate language Scalar Optimizatio
    - `hpo` – High Performance Optimization
    - `hlo` – High-level Optimization
    - `vec` – Vectorization
    - `all` – All optimizations
- **Uložení reportu do souboru**
  - `/Qopt-report-file:[file]` (Windows)
  - `-opt-report-file=[file]` (Linux, MacOS)

```
% icc -O3 -opt-report-phase=hlo -opt-report-phase=hpo
```

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
…
Loop at line 8 blocked by 128
Loop at line 9 blocked by 128
Loop at line 10 blocked by 128
…
Loop at line 10 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
…
…(10)…  loop was not vectorized: not inner loop.
…(8)…   loop was not vectorized: not inner loop.
…(9)…   PERMUTED LOOP WAS VECTORIZED
…
```

Co dokáže kompilátor auto vektorizovat?

- **Počitatelné smyčky**

```
typedef struct{ float* data; size_t size;} vec_t;

void vec_elwise_product(vec_t* a, vec_t* b, const vec_t* c)
{
  for (auto i = 0; i < a->size; i++}
    c->data[i] = a->data[i] * b->data[i];
}
```

- **Smyčky s jedním vstupem a výstupem**

```
while (i < 100) {
   a[i] = b[i] * c[i];
   if (a[i] < 0.0) break; // data-dependent exit condition:
   i++;
} // loop not vectorized
```

Co dokáže kompilátor auto vektorizovat?

- **Přímý kód** (žádné **switch**, **if** pouze s maskováním)

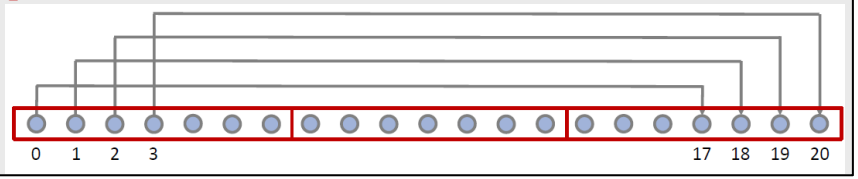
```
for (int i = 0; i < length; i++) {
   float s = b[i] * b[i] - 4 * a[i] * c[i];
   if (s >= 0) x[i] = sqrt(s);
   else        x[i] = 0.;
} // loop vectorized (because of masking)
```

- **Pouze nejvnitřnější smyčky** (kompilátor může smyčky kolabovat a přehazovat)
- **Bez volání funkcí až na**
  - Intrinsic matematiku (sin, log, …)
  - Inline funkce
  - Elementární funkce `__attribute__((vector))`
  - OMP SIMD funkce `#omp declare simd`
  - **Pozor na operátor [], hlídání mezí polí…**

Na čem vektorizace nejčastěji zhavaruje

1. **Nejednotkový rozestup** – Lze načítat pouze datové elementy uložené v paměti za sebou. Pokud je rozestup fixní, lze použít operace gather/scatter
2. **Nezarovnané datové struktury** – Je nutné vydávat více instrukcí load/store (gather, scatter, shuffle)
3. **Datové závislosti mezi iteracemi** – (RAW, WAR, WAW), některé lze ošetřit redukcí.
4. **Pointer aliasing** – překrývající se paměťové oblasti (je nutné dělat testy za chodu a volit různé varianty provedení – memcpy vs. memmove)

```
void add(float* a, float* b, int n)
{
    for (int i = 0; i < n - 17; i++)
    {
        a[i] += b[i + 17];
    }
}
```



Překážky vektorizaci – shrnutí

- Existence závislostí mezi iteracemi
- Nejednotkový rozestup (gather / scatter)
- Míchání datových typů
- Příliš složité podmínky
- Podmínka může strážit výjimku
- Příliš malý trip count (počet iterací)

- Složité indexování
- Nepodporovaná struktura smyček
- Existence nevektorizovatelného příkazu
- Mimo vnitřní smyčku
- Vektorizace možná, ale dle heuristiky kompilátoru neefektivní
- Operátor nevhodný pro vektorizaci

https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/vectorization/automatic-vectorization/programming-guidelines-for-vectorization.html

Problémy s C++ STL knihovnou

- STL knihovna může být použita, ale musí si s ní kompilátor rozumět

```
std::vector<double> A, B;

void foo(int iters, int x, int y)
{
  for (int i = 0; i < iters; i++)
    A[x + i] += B[y + i];
}
```

```
$ icc -vec-report3 -c code.cpp
   code.cpp(5): (col. 3) remark: loop was not vectorized:
      existence of vector dependence.
   code.cpp(6): (col. 7) remark: vector dependence: assumed
      FLOW dependence
   between _M_start line 6 and _M_start line 6.
```

- Problémy se závislostmi v těle smyčky (dáno implementací STL)
- Kompilátor také nemusí rozpoznat, že A a B jsou globální invarianty.

Problémy s C++ STL knihovnou: Řešení

```
#include <vector>

std::vector<double> A, B;

void foo(int iters, int x, int y)
{
  double       *a = A.data();
  const double *b = B.data();

 #pragma omp simd
 for (int i = 0; i < iters; i++)
 {
    a[x + i] += b[y + i];
 }
}
```

```
$ icc -vec-report3 -c code.cpp
code.cpp(9): (col. 3) remark: LOOP WAS VECTORIZED.
```

Zarovnání dat v paměti

- Nesprávné zarovnání dat (na **16/32/64 bytů** pro **SSE, AVX** a **AVX-512**) je jeden z hlavních důvodů špatného výkonu.
- Proto kompilátor vkládá do kódu test na zarovnání.
  - Pokud nejsou data zarovnaná, generuje tzv. peel, body a reminder.
- Zarovnání dat jen nutné explicitně zmínit když
  - Alokujeme data
  - Deklarujeme nový pointer
  - Deklarujeme funkci, které má na vstupu pointer.
- **Penalizace:**
  - Nezarovnaný přístup vs. zarovnaný (ale stále v rámci stejné cache line) 40 % výkonu dolů.
  - Nezarovnaný přístup vs. zarovnaný (ale přes více cache line) až o 500 % horší.

Zdroj: http://software.intel.com/en-us/articles/reducing-the-impact-of-misaligned-memory-accesses/

Jak zarovnat data v paměti

- Pro nová statická pole zarovnaná na N byte:
  - `__declspec(align(N)) type name[size];` `// Intel`
  - `type name[size] __attribute__((aligned(N)));` `// GNU`
  - `alignas(N) type name[bounds];` `// C++-11`
- Nově alokovanou dynamickou paměť:
  - `_mm_malloc(size, N);` `// X86 CPU linux`
  - `_mm_free(p);`
  - `posix_memalign(void **memptr, size_t N, size_t size);` `// POSIX`
  - `aligned_alloc(size_t N, size_t size)` `// C++-11`
  - Lze rovněž přetížit alokátory new a delete
- Argumenty funkcí:
  - `__assume_aligned(name, N);`
- Pro specifické smyčky:
  - `#pragma omp simd align(a:64)`

Pokud je zarovnání porušeno, může nastat výjimka!

Zarovnání vícerozměrných struktur (Intel Compiler)

```
void matvec(double a[][COLWIDTH], double b[], double c[])
{
  int i, j;
  for(i = 0; i < size1; i++) {
    b[i] = 0;
#pragma vector aligned
    for(j = 0; j < size2; j++)
      b[i] += a[i][j] * c[j];
  }
}
```

- Let's assume `a`, `b` and `c` are be declared 16 byte aligned in calling routine
- **Question:** Would this be correct when compiled for Intel® SSE2?
- **Answer:** It depends on `COLWIDTH`!
  - In case `COLWIDTH` is even: Yes
  - In case `COLWIDTH` is odd: No, vectorized code would fail by alignment error after first row!
- **Solution:**
  Instead of pragma use `__assume_aligned(<array>, base)` here as this refers to the start address only. It wouldn't allow vectorization, nevertheless!

Vliv zarovnání vygenerovaný kód

Compiled both cases using **-xAVX:**

```
void mult(double* a, double* b, double* c)
{
  int i;
#pragma vector aligned
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
..B2.2:
  vmovupd    (%rdi,%rax,8), %ymm0
  vmulpd     (%rsi,%rax,8), %ymm0, %ymm1
  vmovntpd   %ymm1, (%rdx,%rax,8)
  addq       $4, %rax
  cmpq       $1000000, %rax
  jb         ..B2.2
```

```
void mult(double* a, double* b, double* c)
{
  int i;
#pragma vector unaligned
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
..B2.2:
  vmovupd    (%rdi,%rax,8), %xmm0
  vmovupd    (%rsi,%rax,8), %xmm1
  vinsertf128 $1, 16(%rsi,%rax,8), %ymm1, %ymm3
  vinsertf128 $1, 16(%rdi,%rax,8), %ymm0, %ymm2
  vmulpd     %ymm3, %ymm2, %ymm4
  vmovupd    %xmm4, (%rdx,%rax,8)
  vextractf128 $1, %ymm4, 16(%rdx,%rax,8)
  addq       $4, %rax
  cmpq       $1000000, %rax
  jb         ..B2.2
```

Compiler can create more efficient code if alignment can be guaranteed!

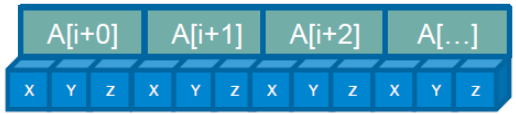
Nejednotkový rozestup

- Non-consecutive memory locations are being accessed in the loop
- Vectorization works best with contiguous memory accesses
- Vectorization might still be possible in cases of non-contiguous memory access but…
  - Data arrangement operations might be too expensive (e.g. access pattern linear/regular)
  - Vector report issued when too expensive:
    `Loop was not vectorized: vectorization possible but seems inefficient`
- Examples:

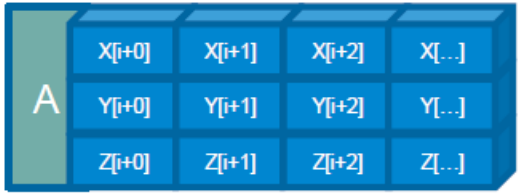
```
for(i = 0; i <= MAX; i++) {
  for(j = 0; j <= MAX; j++) {
    D[i][j] += 1;                    // Unit stride
    D[j][i] += 1;                    // Non-unit stride but linear
    A[j * j] += 1;                   // Non-unit stride
    A[B[j]] += 1;                    // Non-unit stride (scatter)
    if(A[MAX - j]) == 1) last = j; // Non-unit stride
  }
}
```

Datové struktury – AoS nebo SoA?

```
struct Point;
{
  int x;
  int y;
  int z;
};
Point A[100]; //AoS
```




```
struct Points;
{
  int x[100];
  int y[100];
  int z[100];
};
Elements A; //SoA
```



- Pole struktur (AoS) vedou na špatné zarovnání v paměti a nejednotkový rozestup. Proto se špatně vektorizují.
- Struktury polí (SoA) mohou být snadno zarovnány, ale porušují OOP.
- Podpora speciálních knihoven, např. Intel SDLT
  - https://www.intel.com/content/dam/www/public/us/en/documents/presentation/improving-vectorization-efficiency.pdf

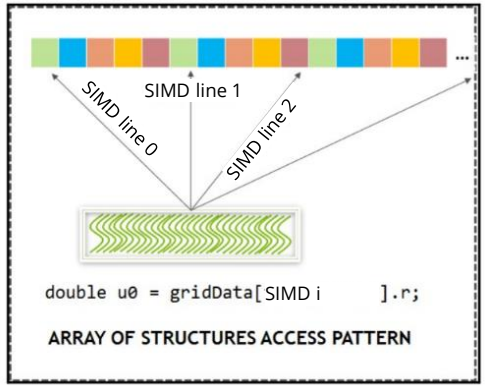
Struktury a objekty v SIMD

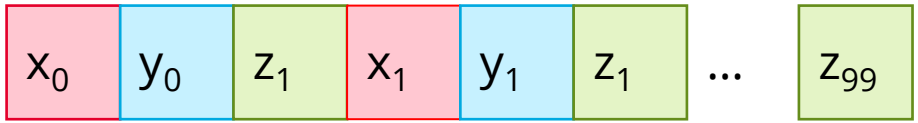
```
struct Point{
  int x;
  int y;
  int z;
};

struct Point points[100];

for (int i=0; i<100; i++){
  points[i].x += 10;
  points[i].y += 20;
  points[i].z += 30;
}
```

Scalar – OK
Rozestup 1

SIMD – Problém
Rozestup 3

```
struct Coefficients_SOA
 int r;
 int b;
 int g;
 int hue;
 int saturation;
 int maxVal;
 int minVal;
 int finalVal;
};
```





- Všechna vlákna v SIMD čtou nejprve komponentu x, poté y a nakonec z
- Řešením je přeuspořádání struktury

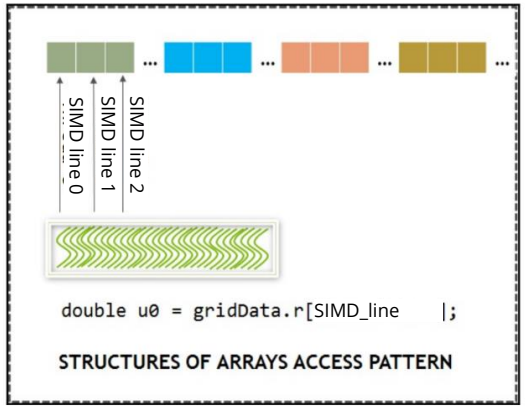
Struktura polí v SIMD

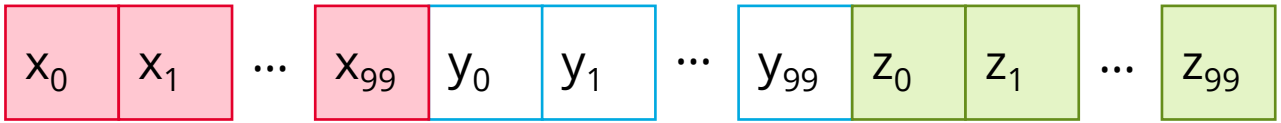
```
struct Points{
  int x[4];
  int y[4];
  int y[4];
}

struct Points points;

for (int i=0; i<100; i++){
  points.x[i] += 10;
  points.y[i] += 20;
  points.z[i] += 30;
}
```

Scalar – Problém
Rozestup 3

SIMD – OK
Rozestup 1

```
struct Coefficients_AOS {
 int* r;
 int* b;
 int* g;
 int* hue;
 int* saturation;
 int* maxVal;
 int* minVal;
 int* finalVal;
};
```

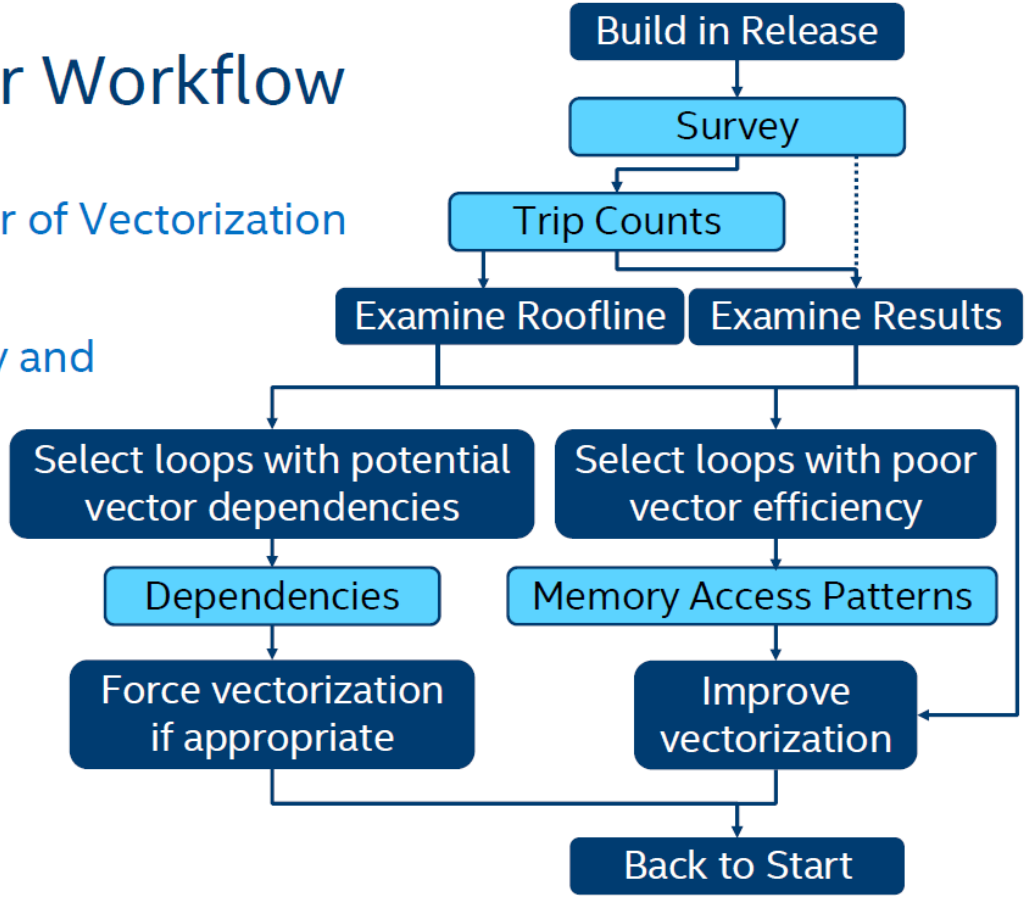




- Pro skalární a SIMD implementaci je nutné mít data uspořádaná jinak!
- Struktura polí rozbíjí Objektově orientovaný model

Vectorization Advisor Workflow

- **Survey** is the bread and butter of Vectorization Advisor! All else builds on it!
- **Trip Counts** adds onto Survey and enables the **Roofline.**
- **Dependencies** determines whether it's safe to force a scalar loop to vectorize.
- **Memory Access Patterns** diagnoses vectorization inefficiency caused by poor memory striding.



Survey
Vectorization Advisor

Tip:
For vectorization, you generally only care about loops. Set the type dropdown to “Loops”.

Function/Loop Icons
- Scalar Function
- Vector Function
- Scalar Loop
- Vector Loop

Vectorizing a loop is usually best done on innermost loops. Since it effectively divides duration by vector length, you want to target loops with high self time.

Efficiency is important!

$$\text{Efficiency} = 100\% \frac{\text{Speedup}}{\text{Vec. Length}}$$

The black arrow is 1x. Gray means you got less than that. Gold means you got more. You want to get this value as high as possible!

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops			
						Vect...	Efficiency	Gain...	VL
[loop in main at example.cpp:38]	1 Assumed depend...	0.391s	0.391s	Scalar	vector depen...				
[loop in main at example.cpp:64]	1 Possible inefficien...	0.297s	0.297s	Vector...		AVX2	2%	0.37x	16
[loop in main at example.cpp:51]	1 Possible inefficien...	0.094s	0.094s	Vector...	1 vectorizatio...	AVX2	8%	1.23x	16
[loop in main at example.cpp:26]		0.030s	0.030s	Vector...		AVX2	100%	7.98x	8
[loop in main at example.cpp:14]	3 Assumed depend...	0.000s	0.000s	Scalar	vector depen...				
[loop in main at example.cpp:23]		0.000s	0.030s	Scalar	inner lo... w...				

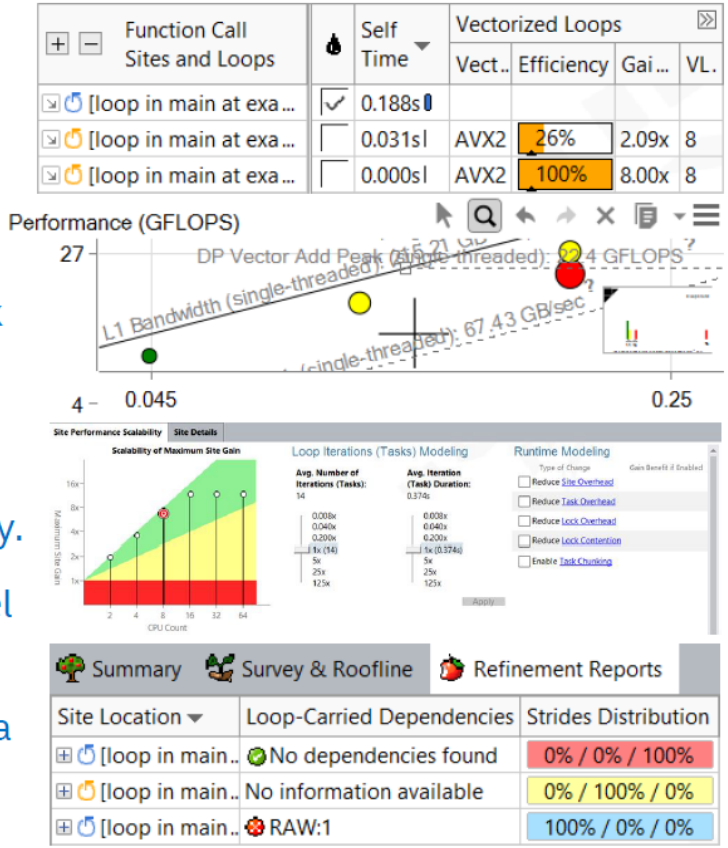
Expand a vectorized loop to see it split into body, peel, and remainder (if applicable).

Advisor *advises* you on potential vector issues. This is often your cue to run MAP or Dependencies. Click the icon to see an explanation in the bottom pane.

The Intel Compiler embeds extra information that Advisor can report in addition to its sampled data, such as why loops failed to vectorize.

Summary

- V T **Survey** – Find the most promising sites for threading, see the meat of the vectorization information, and get recommendations from Advisor.
- V T **Trip Counts & FLOPS** – Add to your Survey report to help fine-tune vector efficiency and capability, as well as unlock the powerful **Roofline** to visualize your bottlenecks and help direct your efforts.
- T **Suitability** – Predict how well your proposed threading model will scale under certain conditions quickly and easily.
- V T **Dependencies** – Prove or disprove the existence of parallel dependencies and learn how to fix them.
- V **Memory Access Patterns** – See how you traverse your data and how it affects your vector efficiency and cache bandwidth usage.



KNIHOVNA OPENMP

Nové pragmy v OpenMP 4.0 (červenec 2013)

- **#pragma omp simd [clause[[,] clause] ...]**

  for ();
- **#pragma omp declare simd [clause[[,] clause] ...]**

  function();
- **pragma omp for simd [clause[[,] clause] ...]**

  for (…)
- **Clause**: safelen(length), linear(list[:linear-step]), aligned(list[:alignment]), private(list), lastprivate(list), reduction(reduction-identifier:list), collapse(n), simdlen(length), linear(argument-list[:constant-linear-step]), aligned(argument-list[:alignment]), uniform(argument-list), inbranch, notinbranch
- **Podporováno od gcc-4.9**
- Více info na http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Smyčky v OpenMP

- Bez direktivy SIMD se kompilátoru smyčku nepodaří vektorizovat kvůli řešení pointer aliasingu
- Pragma SIMD kompilátoru říká, že vektorizace je možná. Kompilátor tedy vypne všechny heuristiky a provede vektorizaci.
- Odpovědnost za dodržení pravidel vektorizace přebírá programátor.

```
void add(float* a, float* b, float* c,
         float* d, float* e, int n)
{
   #pragma omp simd
   for (int i=0; i<n; i++)
      a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Smyčky v OpenMP 4.0: Redukce

```
#pragma omp simd reduction(+:sum)
for(i = 0; i < *p; i++)
{
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

- **Programátor se zaručuje, že**:
  - `*p` je loop invariant, tedy hodnota na adrese `p` je konstanta.
  - `sum` není aliasovaná s `B[]` nebo `C[]`
  - `A[]` se nepřekrývá s `B[]` nebo `C[]`
  - Na `sum` se má pohlížet jako na redukovanou proměnnou

  - Kompilátor může přeházet pořadí operací pro lepší výkon
  - Kód bude vektorizován i když interní heuristiky kompilátoru říkají, že to zhorší výkon.

OpenMP SIMD dovětky

- **`safelen (lenght)`**
  - Maximální počet iterací, které se mohou vykonávat současně bez porušení závislostí (`a[i] += a[i - 10]`)
  - V praxi je to maximální délka vektorového registru
- **`simdlen (lenght)`**
  - Délka preferovaného vektorového registru
- **`linear (list[:linear-step])`**
  - Hodnota proměnné je ve vztahu k číslu iterace
  - $x_i = x_{orig} + i * linear\_step$
- **`aligned (list[:alignment])`**
  - Dané proměnné jsou zarovnány na daný počet bytů
  - Defaultní hodnota je dána architekturou.
- **`colapse (n)`**
  - Kolik smyček pod sebou se má zkolabovat do jedné.

SIMD-Enabled funkce

- SIMD-enabled funkce umožnují definovat funkce, které lze volat v rámci vektorizovaného kódu (smyčky).
- Direktiva se vkládá nad deklaraci (prototyp funkce do hlavičkového souboru).
- **Zápis funkce je stejný jako pro skalární variantu (jeden element).**
- Programátor:
  - Napíše standardní funkci, která pracuje se skalárními parametry
  - Anotuje funkce pomocí pragmy a dovětků `#pragma omp declare simd`
  - Dále se nestará o to, jestli je funkce použita ve skalární nebo vektorové smyčce.
- Kompilátor:
  - Generuje skalární i vektorovou verzi/verze.
  - Volá správnou variantu funkce

Ukázka SIMD-Enabled funkcí

```
#pragma omp declare simd
float min(float a, float b){
  return a < b ? a : b;
}
```

```
vec8 min_v(vec8 a, vec8 b) {
  return a < b ? a : b;
}
```

```
#pragma omp declare simd
float distsq(float x, float y){
  return (x - y) * (x - y);
}
```

```
vec8 distsq_v(vec8 x, vec8 y){
  return (x - y) * (x - y);
}
```

```
void example() {
  #pragma omp for simd
  for (i=0; i < N; i++) {
   d[i] = min(distsq(a[i], b[i]), c[i]);
  }
}
```

```
vd = min_v(distsq_v(
                va, vb),
            vc)
```

Dovětky pro SIMD-Enabled funkce

- **`simdlen (lenght)`**
  - Generuje funkci s podporou dané délky registru. Lze takto generovat specifické funkce pro SSE, AVX a AVX512
- **`linear (list[:linear-step])`**
  - Hodnota proměnné je ve vztahu k číslu iterace
  - $x_i = x_{orig} + i * linear\_step$
- **`uniform (list[:linear-step])`**
  - Hodnota proměnné je konstantní ve všech iteracích
- **`aligned (list[:alignment])`**
  - Dané proměné jsou zarovány na daný počet bytů
- **`inbranch`**
  - Funkce je vždy volána zevnitř podmínky
- **`notinbranch`**
  - Funkce není nikdy volána zevnitř podmínky

Ukázka inbranch a notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
  /* do something */
  return x * 2.0;
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
  /* do something */
  vmulpd x{m}, 2.0, tmp
  return tmp;
}
```

```
void example() {
  #pragma omp simd
  for (int i = 0; i < N; i++)
    if (a[i] < 0.0)
      b[i] = do_stuff(a[i]);
}
```

```
for (int i = 0; i < N; i+=8) {
  vcmp_lt &a[i], 0.0, mask
  b[i] = do_stuff_v(&a[i], mask);
}
```

SIMD-Enabled funkce: modifikátory linear a uniform

- **Proč je potřebujeme?**
- **Protože bez nich je každý parametr funkce brán jako vektor**

```
#pragma omp declare simd uniform(a) linear(i:1)
void foo(float* a, int i):
   a is a pointer
   i is a sequence of integers [i, i+1, i+2, …]
   a[i] is a unit-stride load/store ([v]movups)
```

```
#pragma omp declare simd
void foo(float* a, int i):
   a is a vector of pointers
   i is a vector of integers
   a[i] becomes gather/scatter.
```

- Reference: http://software.intel.com/en-us/articles/usage-of-linear-and-uniform-clause-in-elemental-function-simd-enabled-function-clause

Intrinsické datové typy

Datový typ	Obsah	SSE extension	SSE2 extension	SSE4 extension
`__m128`	4 x float	Available	Available	Available
`__m128d`	2 x double	Not available	Available	Available
`__m128i`	16 x char 8 x short 4 x int	Not available	Available	Available

SSE intrinsics: Pojmenování

- Most intrinsic names use the following notational convention:

```
_mm_<intrin_op>_<suffix>
```

- **<intrin_op>** indicates the basic operation of the intrinsic; for example, add for addition and `sub` for subtraction.
- **<suffix>** denotes the type of data the instruction operates on.
  - The first one or two letters of each suffix denote whether the data is
    - p packed
    - ep extended packed
    - s scalar
  - The remaining letters and numbers denote the type, with notation as follows:
    - s single-precision floating point
    - d double-precision floating point
    - i32 signed 32-bit integer
    - u32 unsigned 32-bit integer

Pokračování příště