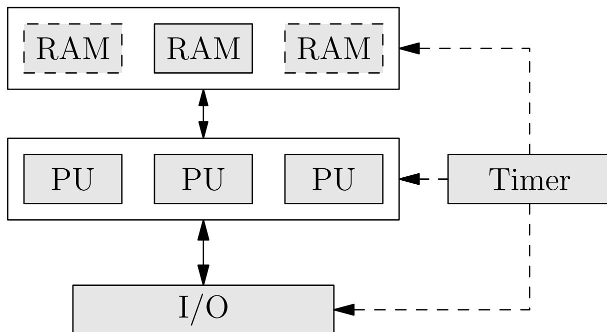
Vektorizace

Paralelní architektury

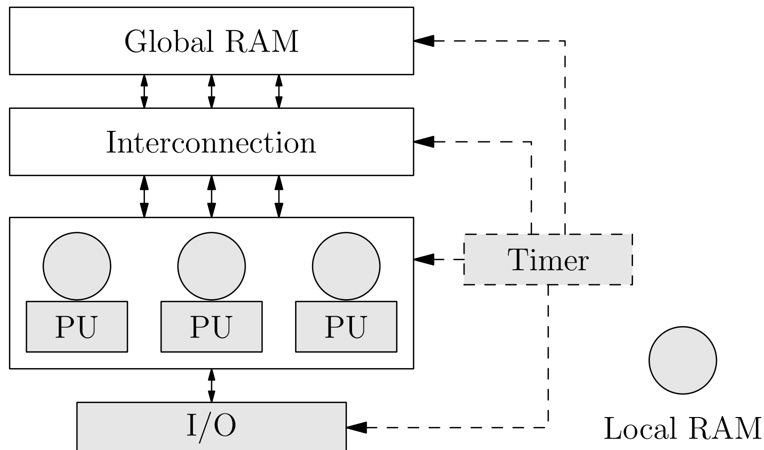
Vektorové počítače

Paralelní architektury

Definice: **Paralelní architektura je taková, která obsahuje více jednotek pro zpracování dat (PU).**



Paralelní architektura II.



Paralelní architektura II.

- **Globální paměť** - zprostředkovává komunikaci mezi výpočetními jednotkami
- **Lokální paměť** - výpočetní jednotky ji používají ve chvíli, kdy provádějí výpočty nezávisle na ostatních
  - lokální paměť je přítomna ve všech paralelních architekturách
  - přístup do globální paměti je vždy pomalý
  - lokální paměť může být:
    - operační paměť výpočetního uzlu v případu klastru nebo gridu
    - cache paměť v případě více procesorových systémů
    - registr v případě vektorových architektur (MMX, GPU)
    - ...
- **Časovač** (Timer)
  - stejně jako u sekvenčních systémů i zda má význam synchronizace
  - některé architektury (klastr,grid) jsou synchronizovány pouze přístupem do globální paměti, časovač zde pak chybí

Vektorové počítače - vector computers

- mají podporu i pro vektorovou aritmetiku
  - např. umožňují v jednom kroku sečíst dva vektory
- patří sem jedny z prvních superpočítačů
  - CDC STAR
  - Cray-1
  - Cyber-205

Vektorové počítače

Cray-1

- byl instalován roku 1976 v Los Alamos National Laboratory

SIMD

- pro tyto architektury se vžilo označeni SIMD
  - *Single Instruction, Multiple Data*
- dnes jsou zastoupeny v každém CPU jako SIMD instrukce

MMX

- MMX zkratka zřejmě znamená *MultiMedia eXtension*
- oficiálně ale žádný význam nemá, protože by jako zkratka nemohla být autorsky chráněna Intelem
- MMX bylo prvně implementováno v Pentiu P5, 1997
- rozšíření využívá 8 64-bitových registrů pro výpočty s pohyblivou desetinnou čárkou, které lze využít pro MMX instrukce
- registry se označují jako `MM0, ..., MM7`
- nelze tedy zároveň provádět výpočty s pohyblivou čárkou a MMX instrukce
- po provedení jakékoliv instrukce je nutné vrátit registry do stavu použitelného pro desetinné výpočty instrukcí `emms`
- do těchto registrů lze načíst
  - jeden 64-bitový quadword
  - dva 32-bitové doublewordy
  - čtyři 16-bitové wordy
  - osm 8-bitových bajtů
- s nimi pak lze provádět paralelní operace

MMX

Příklady některých MMX instrukcí

padd	s	b	sečti dva MMX registry jako 8 bajtů se znaménkem
		w	sečti dva MMX registry jako 4 wordy se znaménkem
		d	sečti dva MMX registry jako 2 doublewordy se znaménkem
	u	b	sečti dva MMX registry jako 8 bajtů bez znaménka
		w	sečti dva MMX registry jako 4 wordy bez znaménka
		d	sečti dva MMX registry jako 2 doublewordy bez znaménka
psub	s	b	odečti dva MMX registry jako 8 bajtů se znaménkem
		w	odečti dva MMX registry jako 4 wordy se znaménkem
		d	odečti dva MMX registry jako 2 doublewordy se znaménkem
	u	b	odečti dva MMX registry jako 8 bajtů bez znaménka
		w	odečti dva MMX registry jako 4 wordy bez znaménka
		d	odečti dva MMX registry jako 2 doublewordy bez znaménka

MMX

MMX instrukce dále umožňují provádět

- bitové operace s obsahem registrů
- bitové posuvy
- porovnávání

MMX nepodporuje dělení.

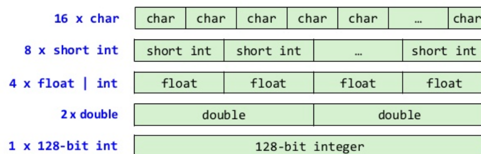
3DNow!

- v roce 1998 firma AMD rozšířila rozšíření MMX o instrukce pro operaci s typem `float`
  - `pi2fd, pf2id` = konverze mezi `float` a `int`
  - `pfmax, pfmin` = výpočet maxima a minima
  - `pfadd, pfsub, pfmul, pfrcp, pfrsqrt` = přičítání, odčítání, násobení, převrácená hodnota, odmocnina
- dále jsou zde instrukce pro `prefetching` dat do cache
- v roce 1999 vzniklo další rozšíření 3DNow!2, které ale přidalo jen pár instrukcí

SSE

SSE = *Streaming SIMD Extension*

- zavedla firma Intel v roce 1999
- narozdíl od MMX a 3DNow! CPU dostává navíc osm 128-bitových registrů `XMM0, ... XMM7`
- každý registr může obsahovat čtyři čísla typu `float`



SSE

- lze provádět (mimo jiné) tyto operace se čtveřicemi čísel typu `float`
  - `addps` = součet
  - `subps` = rozdíl
  - `mulps` = násobení
  - `divps` = dělení
  - `rcpps` = převrácená hodnota
  - `sqrtps` = odmocnina
  - `rsqrtps` = převrácená hodnota odmocniny
  - `maxps` = maximum
  - `minps` = minimum
  - `cmpXXps` = porovnání, kde
    - `XX = eq, lt, le, ne, nlt, nle`

SSE 2

- jde o další rozšíření od firmy Intel z roku 2000
- hlavní novinkou je podpora pro dvojitou přesnost `double`
- velikost registrů zůstala zachována, tudíž je do jednoho registru možné načíst jen dvě čísla typu `double`
- přibyly nové instrukce pro počítání s dvojitou přesností
- od instrukcí pro jednoduchou přesnost se liší pouze písmenem `d` místo `s` na konci
  - tj. `addpd` místo `addps` apod.

SSE 3 a SSSE 3

- SSE 3 je rozšíření z roku 2004
- přidává jen pár dalších instrukcí
- zejména jde o tzv. horizontální instrukce, např.
  - `haddps` = součet všech čtyř čísel v daném registru
- SSSE3 = *Suplemental SSE3*, 2006
- přidává např. horizontální instrukce pro celočíselnou aritmetiku

SSE 4

- SSE 4.1, 2007
- přidává např. instrukci pro výpočet skalárního součinu nebo součtu absolutních hodnot rozdílu
- SSE 4.2
- přidává instrukce pro práci s řetězci

AVX

AVX = *advanced vector extension*, 2011

- velikost registrů se zvyšuje ze 128-bitů na 256-bitů
- místo `XMM0, ..., XMM7` se jmenují `YMM0,...,YMM7`
- v případě 64-bitových CPU přibudou i registry `YMM8,...,YMM15`
- všechny instrukce z SSE se rozšiřují na tyto větší registry
- instrukce mohou mít i tři operandy
  - SSE umí jen operace typu `a += b`
  - AVX dokáže i `a = b + c`
- tyto instrukce podporují procesory architektury Sandy Bridge, Bulldozer a novější

AVX 2

AVX2, 2013

- jde o další rozšíření AVX
- přibyla sada funkcí FMA3 = *fused multiply-add*
  - jde o instrukce, které spočítají výraz `a + b * c` najednou
- přibyly funkce pro *gather* = načtení čísel, které nejsou v paměti uloženy v souvislém bloku
- tyto instrukce podporují procesory architektury Intel Haswell (2013), AMD Excavator (2015) a novější

AVX 512

AVX 512, 2015

- jde o další zvětšení registrů až na 512-bitů
- umožní tedy operovat s až 8 operandy typu `double`
- prvně se objeví v akcelerátorech Knights Landing Xeon Phi (2015) a následně v procesorech Xeon řady Skylake (2016) a Cannonlake (2017)

Optimalizace pomocí vektorových instrukcí

Jsou celkem tři možnosti, jak využít tyto instrukce v našem kódu:

- využít optimalizace překladače
- využít "vnitřní" instrukce (`intrinsic instructions`) překladače
- (vložit kód v assembleru)
  - nemá snad žádnou výhodu oproti druhé možnosti
- procesorem podporovaná rozšíření lze zjistit v Linuxu příkazem
  - `cat /proc/cpuinfo`

Optimalizace pomocí vektorových instrukcí

- překladače dokáží poměrně dobře využít vektorových instrukcí
- někdy je možné jim práci usnadnit
- hlavní překážky bránící překladači využít vektorových instrukcí jsou
  - správné zarovnání dat v paměti
  - možné překrývání polí
  - nejasný počet opakování smyčky

Optimalizace pomocí vektorových instrukcí

Zarovnání dat v paměti – *memory alignment*

- chceme-li použít SSE instrukce na pole čísel, musí toto pole začínat na adrese dělitelné 16
  - 16 bajtů = 128 bitů = velikost SSE registrů XMM
  - obecně vždy platí, že data se do registru načítají mnohem rychleji, jsou-li v paměti zarovnána na násobek velikosti registru

Optimalizace pomocí vektorových instrukcí

Překryv polí – *array overlap, aliasing*

- například následující smyčku není možné vektorizovat

```
1  float a[ 128 ];
2  ...
3  float* b = &a[ 1 ];
4  for( int i = 0; i < 127; i++ )
5     b[ i ] += a[ i ];
```

Nejasný počet opakování smyčky

- opuštění smyčky může záviset na výpočtu v jejím vnitřku
- pokud např. meze for smyčky není násobek 4, musí se některé prvky zpracovat sekvenčně a to je nutné rozhodnout až za běhu programu

Optimalizace pomocí vektorových instrukcí

- předáme-li překladači `gcc` přepínač `-ftree-vectorizer-verbose=1`, řekne nám, které smyčky vektorizoval nebo rozbalil
  - místo jedničky lze udávat čísla až do šesti pro více informací

Optimalizace pomocí vektorových instrukcí

Příklad: Součet dvou vektorů

```
1   void vectorAddition( const float* v1,
2                        const float* v2,
3                        const int size,
4                        float* sum )
5   {
6      for( int i = 0; i < size; i++ )
7         sum[ i ] = v1[ i ] + v2[ i ];
8   }
```

Překladač hlasí úspěšnou vektorizaci:

```
1   Analyzing loop at vectorization/vector-addition.cpp:6
2
3
4   Vectorizing loop at vectorization/vector-addition.cpp:6
5
6   vector-addition.cpp:6: note: create runtime check for data references *_10 ar
7   vector-addition.cpp:6: note: create runtime check for data references *_13 ar
8   vector-addition.cpp:6: note: created 2 versioning for alias checks.
9
10  vector-addition.cpp:6: note: === vect_do_peeling_for_loop_bound ===Setting up
11
12  vector-addition.cpp:6: note: LOOP VECTORIZED.
13  vector-addition.cpp:1: note: vectorized 1 loops in function.
14
15  vector-addition.cpp:6: note: Completely unroll loop 2 times
```

Optimalizace pomocí vektorových instrukcí

Překladač nám hlásí:

- `create runtime check for data references`
  - generuje kód pro kontrolu zarovnání polí za běhu programu
- `created 2 versioning for alias checks`
  - generuje kód pro kontrolu překryvu polí
- `LOOP VECTORIZED`
  - smyčka na řádku 6 byla vektorizována

Následující příklady nám ukážou, co překladač vygeneroval.

```
1  gdb --args vector-addition
2  (gbd) disas vectorAddition
```

Optimalizace pomocí vektorových instrukcí

```
Dump of assembler code for function vectorAddition(float const*, float const*, int, float*):
   0x0000000000401300 <+0>:     test   %edx,%edx
   0x0000000000401302 <+2>:     jle    0x40147d <vectorAddition(float const*, float const*, int, float*)+381>
   0x0000000000401308 <+8>:     lea    0x10(%rcx),%rax
   0x000000000040130c <+12>:    lea    0x10(%rdi),%r9
   0x0000000000401310 <+16>:    cmp    %rax,%rdi
   0x0000000000401313 <+19>:    setae  %r8b
   0x0000000000401317 <+23>:    cmp    %r9,%rcx
   0x000000000040131a <+26>:    setae  %r9b
   0x000000000040131e <+30>:    or     %r9d,%r8d
   0x0000000000401321 <+33>:    cmp    %rax,%rsi
   0x0000000000401324 <+36>:    lea    0x10(%rsi),%r9
   0x0000000000401328 <+40>:    setae  %al
   0x000000000040132b <+43>:    cmp    %r9,%rcx
   0x000000000040132e <+46>:    setae  %r9b
   0x0000000000401332 <+50>:    or     %r9d,%eax
   0x0000000000401335 <+53>:    test   %al,%r8b
   0x0000000000401338 <+56>:    je     0x401480 <vectorAddition(float const*, float const*, int, float*)+384>
   0x000000000040133e <+62>:    cmp    $0xb,%edx
   0x0000000000401341 <+65>:    jbe    0x401480 <vectorAddition(float const*, float const*, int, float*)+384>
   0x0000000000401347 <+71>:    push   %r12
   0x0000000000401349 <+73>:    mov    %edx,%r10d
   0x000000000040134c <+76>:    shr    $0x2,%r10d
   0x0000000000401350 <+80>:    push   %rbp
   0x0000000000401351 <+81>:    cmp    $0x4,%r10d
   0x0000000000401355 <+85>:    push   %rbx
   0x0000000000401356 <+86>:    lea    0x0(,%r10,4),%ebx
   0x000000000040135e <+94>:    jbe    0x401620 <vectorAddition(float const*, float const*, int, float*)+800>
   0x0000000000401364 <+100>:   lea    -0x5(%r10),%eax
   0x0000000000401368 <+104>:   lea    0x40(%rdi),%r8
   0x000000000040136c <+108>:   mov    %rsi,%r9
   0x000000000040136f <+111>:   shr    $0x2,%eax
   0x0000000000401372 <+114>:   xor    %r11d,%r11d
   0x0000000000401375 <+117>:   shl    $0x6,%rax
   0x0000000000401379 <+121>:   lea    0x80(%rdi,%rax,1),%r12
   0x0000000000401381 <+129>:   mov    %rcx,%rax
   0x0000000000401384 <+132>:   movups (%r9),%xmm1
   0x0000000000401388 <+136>:   movups -0x40(%r8),%xmm0
   0x000000000040138d <+141>:   prefetcht0 0x100(%r8)
   0x0000000000401395 <+149>:   prefetcht0 0x140(%r9)
   0x000000000040139d <+157>:   mov    %r8,%rbp
   0x00000000004013a0 <+160>:   add    $0x40,%r8
   0x00000000004013a4 <+164>:   minps  %xmm1,%xmm0
   0x00000000004013a7 <+167>:   prefetchw 0x140(%rax)
   0x00000000004013ae <+174>:   add    $0x40,%r9
   0x00000000004013b2 <+178>:   add    $0x40,%rax
   0x00000000004013b6 <+182>:   add    $0x4,%r11d
   0x00000000004013ba <+186>:   movlps %xmm0,-0x40(%rax)
   0x00000000004013be <+190>:   movhps %xmm0,-0x38(%rax)
   0x00000000004013c2 <+194>:   movups -0x30(%r9),%xmm1
   0x00000000004013c7 <+199>:   movups -0x70(%r8),%xmm0
   0x00000000004013cc <+204>:   minps  %xmm1,%xmm0
   0x00000000004013cf <+207>:   movlps %xmm0,-0x30(%rax)
   0x00000000004013d3 <+211>:   movhps %xmm0,-0x28(%rax)
   0x00000000004013d7 <+215>:   movups -0x20(%r9),%xmm1
   0x00000000004013dc <+220>:   movups -0x60(%r8),%xmm0
   0x00000000004013e1 <+225>:   minps  %xmm1,%xmm0
   0x00000000004013e4 <+228>:   movlps %xmm0,-0x20(%rax)
   0x00000000004013e8 <+232>:   movhps %xmm0,-0x18(%rax)
   0x00000000004013ec <+236>:   movups -0x50(%r8),%xmm0
   0x00000000004013f1 <+241>:   movups -0x10(%r9),%xmm1
   0x00000000004013f6 <+246>:   minps  %xmm1,%xmm0
   0x00000000004013f9 <+249>:   movlps %xmm0,-0x10(%rax)
   0x00000000004013fd <+253>:   movhps %xmm0,-0x8(%rax)
   0x0000000000401401 <+257>:   cmp    %r12,%r8
   0x0000000000401404 <+260>:   jne    0x401384 <vectorAddition(float const*, float const*, int, float*)+132>
   0x000000000040140a <+266>:   xor    %r8d,%r8d
   0x000000000040140d <+269>:   movups 0x0(%rbp,%r8,1),%xmm0
   0x0000000000401413 <+275>:   movups (%r9,%r8,1),%xmm1
   0x0000000000401418 <+280>:   inc    %r11d
   0x000000000040141b <+283>:   minps  %xmm1,%xmm0
   0x000000000040141e <+286>:   movlps %xmm0,(%rax,%r8,1)
   0x0000000000401423 <+291>:   movhps %xmm0,0x8(%rax,%r8,1)
   0x0000000000401429 <+297>:   add    $0x10,%r8
   0x000000000040142d <+301>:   cmp    %r11d,%r10d
   0x0000000000401430 <+304>:   ja     0x40140d <vectorAddition(float const*, float const*, int, float*)+269>
   0x0000000000401432 <+306>:   cmp    %ebx,%edx
   0x0000000000401434 <+308>:   je     0x401479 <vectorAddition(float const*, float const*, int, float*)+377>
   0x0000000000401436 <+310>:   movslq %ebx,%rax
   0x0000000000401439 <+313>:   movss  (%rdi,%rax,4),%xmm0
   0x000000000040143e <+318>:   minss  (%rsi,%rax,4),%xmm0
   0x0000000000401443 <+323>:   movss  %xmm0,(%rcx,%rax,4)
   0x0000000000401448 <+328>:   lea    0x1(%rbx),%eax
   0x000000000040144b <+331>:   cmp    %edx,%eax
   0x000000000040144d <+333>:   jge    0x401479 <vectorAddition(float const*, float const*, int, float*)+377>
   0x000000000040144f <+335>:   cltq
   0x0000000000401451 <+337>:   add    $0x2,%ebx
   0x0000000000401454 <+340>:   movss  (%rdi,%rax,4),%xmm0
   0x0000000000401459 <+345>:   cmp    %ebx,%edx
   0x000000000040145b <+347>:   minss  (%rsi,%rax,4),%xmm0
   0x0000000000401460 <+352>:   movss  %xmm0,(%rcx,%rax,4)
   0x0000000000401465 <+357>:   jle    0x401479 <vectorAddition(float const*, float const*, int, float*)+377>
   0x0000000000401467 <+359>:   movslq %ebx,%rbx
   0x000000000040146a <+362>:   movss  (%rdi,%rbx,4),%xmm0
   0x000000000040146f <+367>:   minss  (%rsi,%rbx,4),%xmm0
   0x0000000000401474 <+372>:   movss  %xmm0,(%rcx,%rbx,4)
   0x0000000000401479 <+377>:   pop    %rbx
   0x000000000040147a <+378>:   pop    %rbp
   0x000000000040147b <+379>:   pop    %r12
   0x000000000040147d <+381>:   repz retq
   0x000000000040147f <+383>:   nop
   0x0000000000401480 <+384>:   cmp    $0x10,%edx
   0x0000000000401483 <+387>:   jle    0x401631 <vectorAddition(float const*, float const*, int, float*)+817>
   0x0000000000401489 <+393>:   lea    -0x11(%rdx),%r10d
   0x000000000040148d <+397>:   mov    %rdi,%r9
   0x0000000000401490 <+400>:   mov    %rsi,%r8
   0x0000000000401493 <+403>:   and    $0xfffffff0,%r10d
   0x0000000000401497 <+407>:   mov    %rcx,%rax
   0x000000000040149a <+410>:   lea    0x10(%r10),%r11d
   0x000000000040149e <+414>:   xor    %r10d,%r10d
   0x00000000004014a1 <+417>:   nopl   0x0(%rax)
   0x00000000004014a8 <+424>:   movss  (%r9),%xmm0
   0x00000000004014ad <+429>:   prefetcht0 0x4c(%r9)
   0x00000000004014b2 <+434>:   prefetcht0 0x4c(%r8)
   0x00000000004014b7 <+439>:   prefetchw 0x4c(%rax)
   0x00000000004014bb <+443>:   add    $0x10,%r10d
   0x00000000004014bf <+447>:   add    $0x40,%r9
   0x00000000004014c3 <+451>:   minss  (%r8),%xmm0
   0x00000000004014c8 <+456>:   add    $0x40,%rax
   0x00000000004014cc <+460>:   add    $0x40,%r8
   0x00000000004014d0 <+464>:   movss  %xmm0,-0x40(%rax)
   0x00000000004014d5 <+469>:   movss  -0x3c(%r9),%xmm0
   0x00000000004014db <+475>:   minss  -0x3c(%r8),%xmm0
   0x00000000004014e1 <+481>:   movss  %xmm0,-0x3c(%rax)
   0x00000000004014e6 <+486>:   movss  -0x38(%r9),%xmm0
   0x00000000004014ec <+492>:   minss  -0x38(%r8),%xmm0
   0x00000000004014f2 <+498>:   movss  %xmm0,-0x38(%rax)
   0x00000000004014f7 <+503>:   movss  -0x34(%r9),%xmm0
   0x00000000004014fd <+509>:   minss  -0x34(%r8),%xmm0
   0x0000000000401503 <+515>:   movss  %xmm0,-0x34(%rax)
   0x0000000000401508 <+520>:   movss  -0x30(%r9),%xmm0
   0x000000000040150e <+526>:   minss  -0x30(%r8),%xmm0
   0x0000000000401514 <+532>:   movss  %xmm0,-0x30(%rax)
   0x0000000000401519 <+537>:   movss  -0x2c(%r9),%xmm0
   0x000000000040151f <+543>:   minss  -0x2c(%r8),%xmm0
   0x0000000000401525 <+549>:   movss  %xmm0,-0x2c(%rax)
   0x000000000040152a <+554>:   movss  -0x28(%r9),%xmm0
   0x0000000000401530 <+560>:   minss  -0x28(%r8),%xmm0
$
```

Optimalizace pomocí vektorových instrukcí

- funkce má velikost 560 bajtů
- ukážeme si, jak překladači práci usnadnit
- nejprve je potřeba mít všechna pole v paměti správně zarovnána
- toho lze dosáhnout pomocí funkce `memalign`

```
1  #include <malloc.h>
2
3     float* aligned_v1 = ( float* ) memalign( 16, size * sizeof( float ) );
4     float* aligned_v2 = ( float* ) memalign( 16, size * sizeof( float ) );
5     float* aligned_sum = ( float* ) memalign( 16, size * sizeof( float ) );
```

- první parameter, který je tu navíc, udává, na kolik bajtů se má zarovnání provést
- zarovnání musí být mocnina dvou

- dále to musíme překladači oznámit ve funkci samotné

```
1  void alignedVectorAddition( const float* v1,
2                              const float* v2,
3                              const int size,
4                              float*  sum )
5  {
6     float* _sum = ( float* ) __builtin_assume_aligned( sum, 16);
7     const float* _v1 = ( const float* ) __builtin_assume_aligned( v1, 16);
8     const float* _v2 = ( const float* ) __builtin_assume_aligned( &v2[ i ], 16);
9     for( int i = 0; i < size; i ++ )
10    {
11       _sum[ i ] = _v1[ i ] + _v2[ i ];
12    }
13 }
```

Optimalizace pomocí vektorových instrukcí

```
1  Dump of assembler code for function alignedVectorAddition(float const*, float const*, int, float*):
2      0x0000000000401080 <+0>:    test   %edx,%edx
3      0x0000000000401082 <+2>:    jle    0x40115f <alignedVectorAddition(float const*, float const*, int, float*)+223>
4      0x0000000000401088 <+8>:    lea    0x10(%rdi),%r8
5      0x000000000040108c <+12>:   lea    0x10(%rcx),%rax
6      0x0000000000401090 <+16>:   cmp    %r8,%rcx
7      0x0000000000401093 <+19>:   setae  %r8b
8      0x0000000000401097 <+23>:   cmp    %rax,%rdi
9      0x000000000040109a <+26>:   setae  %r9b
10     0x000000000040109e <+30>:   or     %r9d,%r8d
11     0x00000000004010a1 <+33>:   lea    0x10(%rsi),%r9
12     0x00000000004010a5 <+37>:   cmp    %rax,%rsi
13     0x00000000004010a8 <+40>:   setae  %al
14     0x00000000004010ab <+43>:   cmp    %r9,%rcx
15     0x00000000004010ae <+46>:   setae  %r9b
16     0x00000000004010b2 <+50>:   or     %r9d,%eax
17     0x00000000004010b5 <+53>:   test   %al,%r8b
18     0x00000000004010b8 <+56>:   je     0x401140 <alignedVectorAddition(float const*, float const*, int, float*)+192>
19     0x00000000004010be <+62>:   cmp    $0x5,%edx
20     0x00000000004010c1 <+65>:   jbe    0x401140 <alignedVectorAddition(float const*, float const*, int, float*)+192>
21     0x00000000004010c3 <+67>:   mov    %edx,%r10d
22     0x00000000004010c6 <+70>:   xor    %eax,%eax
23     0x00000000004010c8 <+72>:   xor    %r8d,%r8d
24     0x00000000004010cb <+75>:   shr    $0x2,%r10d
25     0x00000000004010cf <+79>:   lea    0x0(,%r10,4),%r9d
26     0x00000000004010d7 <+87>:   movaps (%rdi,%rax,1),%xmm0
27     0x00000000004010db <+91>:   add    $0x1,%r8d
28     0x00000000004010df <+95>:   addps  (%rsi,%rax,1),%xmm0
29     0x00000000004010e3 <+99>:   movaps %xmm0,(%rcx,%rax,1)
30     0x00000000004010e7 <+103>:  add    $0x10,%rax
31     0x00000000004010eb <+107>:  cmp    %r10d,%r8d
32     0x00000000004010ee <+110>:  jb     0x4010d7 <alignedVectorAddition(float const*, float const*, int, float*)+87>
33     0x00000000004010f0 <+112>:  cmp    %r9d,%edx
34     0x00000000004010f3 <+115>:  je     0x40115f <alignedVectorAddition(float const*, float const*, int, float*)+223>
35     0x00000000004010f5 <+117>:  movslq %r9d,%rax
36     0x00000000004010f8 <+120>:  movss  (%rdi,%rax,4),%xmm0
37     0x00000000004010fd <+125>:  addss  (%rsi,%rax,4),%xmm0
38     0x0000000000401102 <+130>:  movss  %xmm0,(%rcx,%rax,4)
39     0x0000000000401107 <+135>:  lea    0x1(%r9),%eax
40     0x000000000040110b <+139>:  cmp    %eax,%edx
41     0x000000000040110d <+141>:  jle    0x40115f <alignedVectorAddition(float const*, float const*, int, float*)+223>
42     0x000000000040110f <+143>:  cltq
43     0x0000000000401111 <+145>:  add    $0x2,%r9d
44     0x0000000000401115 <+149>:  movss  (%rdi,%rax,4),%xmm0
45     0x000000000040111a <+154>:  cmp    %r9d,%edx
46     0x000000000040111d <+157>:  addss  (%rsi,%rax,4),%xmm0
47     0x0000000000401122 <+162>:  movss  %xmm0,(%rcx,%rax,4)
48     0x0000000000401127 <+167>:  jle    0x401168 <alignedVectorAddition(float const*, float const*, int, float*)+232>
49     0x0000000000401129 <+169>:  movslq %r9d,%r9
50     0x000000000040112c <+172>:  movss  (%rdi,%r9,4),%xmm0
51     0x0000000000401132 <+178>:  addss  (%rsi,%r9,4),%xmm0
52     0x0000000000401138 <+184>:  movss  %xmm0,(%rcx,%r9,4)
53     0x000000000040113e <+190>:  retq
54     0x000000000040113f <+191>:  nop
55     0x0000000000401140 <+192>:  xor    %eax,%eax
56     0x0000000000401142 <+194>:  nopw   0x0(%rax,%rax,1)
57     0x0000000000401148 <+200>:  movss  (%rdi,%rax,4),%xmm0
58     0x000000000040114d <+205>:  addss  (%rsi,%rax,4),%xmm0
59     0x0000000000401152 <+210>:  movss  %xmm0,(%rcx,%rax,4)
60     0x0000000000401157 <+215>:  add    $0x1,%rax
61     0x000000000040115b <+219>:  cmp    %eax,%edx
62     0x000000000040115d <+221>:  jg     0x401148 <alignedVectorAddition(float const*, float const*, int, float*)+200>
63     0x000000000040115f <+223>:  repz retq
64     0x0000000000401161 <+225>:  nopl   0x0(%rax)
65     0x0000000000401168 <+232>:  repz retq
```

Optimalizace pomocí vektorových instrukcí

- kód se zmenšil na 232 bajtů
- dále překladači řekneme, že pole se nepřekrývají pomocí atributu `__restrict__`

```
1  void alignedVectorAddition( const float* __restrict__  v1,
2                              const float* __restrict__ v2,
3                              const int size,
4                              float* __restrict__ sum )
5  {
6     float* _sum = ( float* ) __builtin_assume_aligned( sum, 16);
7     const float* _v1 = ( const float* ) __builtin_assume_aligned( v1, 16);
8     const float* _v2 = ( const float* ) __builtin_assume_aligned( &v2[ i ], 16
9     for( int i = 0; i < size; i ++ )
10    {
11       _sum[ i ] = _v1[ i ] + _v2[ i ];
12    }
13 }
```

Optimalizace pomocí vektorových instrukcí

```
1  Dump of assembler code for function alignedVectorAddition(float const*, float const*, int, float*):
2     0x0000000000401080 <+0>:    test   %edx,%edx
3     0x0000000000401082 <+2>:    jle    0x401120 <alignedVectorAddition(float const*, float const*, int, float*)+160>
4     0x0000000000401088 <+8>:    mov    %edx,%r9d
5     0x000000000040108b <+11>:   shr    $0x2,%r9d
6     0x000000000040108f <+15>:   lea    0x0(,%r9,4),%eax
7     0x0000000000401097 <+23>:   test   %eax,%eax
8     0x0000000000401099 <+25>:   je     0x401128 <alignedVectorAddition(float const*, float const*, int, float*)+168>
9     0x000000000040109f <+31>:   cmp    $0x3,%edx
10    0x00000000004010a2 <+34>:   jbe    0x401128 <alignedVectorAddition(float const*, float const*, int, float*)+168>
11    0x00000000004010a8 <+40>:   xor    %r8d,%r8d
12    0x00000000004010ab <+43>:   xor    %r10d,%r10d
13    0x00000000004010ae <+46>:   movaps (%rdi,%r8,1),%xmm0
14    0x00000000004010b3 <+51>:   add    $0x1,%r10d
15    0x00000000004010b7 <+55>:   addps  (%rsi,%r8,1),%xmm0
16    0x00000000004010bc <+60>:   movaps %xmm0,(%rcx,%r8,1)
17    0x00000000004010c1 <+65>:   add    $0x10,%r8
18    0x00000000004010c5 <+69>:   cmp    %r10d,%r9d
19    0x00000000004010c8 <+72>:   ja     0x4010ae <alignedVectorAddition(float const*, float const*, int, float*)+46>
20    0x00000000004010ca <+74>:   cmp    %edx,%eax
21    0x00000000004010cc <+76>:   je     0x401130 <alignedVectorAddition(float const*, float const*, int, float*)+176>
22    0x00000000004010ce <+78>:   movslq %eax,%r8
23    0x00000000004010d1 <+81>:   movss  (%rdi,%r8,4),%xmm0
24    0x00000000004010d7 <+87>:   addss  (%rsi,%r8,4),%xmm0
25    0x00000000004010dd <+93>:   movss  %xmm0,(%rcx,%r8,4)
26    0x00000000004010e3 <+99>:   lea    0x1(%rax),%r8d
27    0x00000000004010e7 <+103>:  cmp    %r8d,%edx
28    0x00000000004010ea <+106>:  jle    0x401120 <alignedVectorAddition(float const*, float const*, int, float*)+160>
29    0x00000000004010ec <+108>:  movslq %r8d,%r8
30    0x00000000004010ef <+111>:  add    $0x2,%eax
31    0x00000000004010f2 <+114>:  movss  (%rdi,%r8,4),%xmm0
32    0x00000000004010f8 <+120>:  cmp    %eax,%edx
33    0x00000000004010fa <+122>:  addss  (%rsi,%r8,4),%xmm0
34    0x0000000000401100 <+128>:  movss  %xmm0,(%rcx,%r8,4)
35    0x0000000000401106 <+134>:  jle    0x401120 <alignedVectorAddition(float const*, float const*, int, float*)+160>
36    0x0000000000401108 <+136>:  cltq
37    0x000000000040110a <+138>:  movss  (%rdi,%rax,4),%xmm0
38    0x000000000040110f <+143>:  addss  (%rsi,%rax,4),%xmm0
39    0x0000000000401114 <+148>:  movss  %xmm0,(%rcx,%rax,4)
40    0x0000000000401119 <+153>:  retq
41    0x000000000040111a <+154>:  nopw   0x0(%rax,%rax,1)
42    0x0000000000401120 <+160>:  repz retq
43    0x0000000000401122 <+162>:  nopw   0x0(%rax,%rax,1)
44    0x0000000000401128 <+168>:  xor    %eax,%eax
45    0x000000000040112a <+170>:  jmp    0x4010ce <alignedVectorAddition(float const*, float const*, int, float*)+78>
46    0x000000000040112c <+172>:  nopl   0x0(%rax)
47    0x0000000000401130 <+176>:  repz retq
48 $
```

Optimalizace pomocí vektorových instrukcí

- dostali jsme se na 176 bajtů
- dalšího zjednodušení lze dosáhnout, pokud můžeme předpokládat fixní počet opakování dělitelný čtyřmi

```
1   const int size = 16;
2   void alignedVectorAddition( const float* __restrict__  v1,
3                               const float* __restrict__ v2,
4                               float* __restrict__ sum )
5   {
6      float* _sum = ( float* ) __builtin_assume_aligned( sum, 16);
7      const float* _v1 = ( const float* ) __builtin_assume_aligned( v1, 16);
8      const float* _v2 = ( const float* ) __builtin_assume_aligned( &v2[ i ], 16
9      for( int i = 0; i < size; i ++ )
10     {
11        _sum[ i ] = _v1[ i ] + _v2[ i ];
12     }
13  }
```

Optimalizace pomocí vektorových instrukcí

```
1  Dump of assembler code for function alignedVectorAddition(float const*, float const*, float*):
2     0x00000000004010a0 <+0>:     xor    %eax,%eax
3     0x00000000004010a2 <+2>:     nopw   0x0(%rax,%rax,1)
4     0x00000000004010a8 <+8>:     movaps (%rdi,%rax,1),%xmm0
5     0x00000000004010ac <+12>:    addps  (%rsi,%rax,1),%xmm0
6     0x00000000004010b0 <+16>:    movaps %xmm0,(%rdx,%rax,1)
7     0x00000000004010b4 <+20>:    add    $0x10,%rax
8     0x00000000004010b8 <+24>:    cmp    $0x40000,%rax
9     0x00000000004010be <+30>:    jne    0x4010a8 <alignedVectorAddition(float const*, float const*, float*)+8>
10    0x00000000004010c0 <+32>:    repz retq
```

- dostáváme 32 bajtů kódu, tj. osmkrát méně, než na počátku
- jak se to projeví na efektivitě kódu?

Optimalizace pomocí vektorových instrukcí

- překvapivě se to neprojeví nijak
- v každé verzi trvá zpracování jednoho elementu cca. 4 takty
- vidíme tedy, že překladač dokáže jednoduché smyčky vektorizovat velmi efektivně
- výsledný kód je efektivní i pro pole, která nejsou v paměti správně zarovnána
- naše optimalizace ale eliminovaly více než půl kilobajtu kódu
- to by mohlo být užitečné v situaci, kdy chceme optimalizovat využtití instrukční L1 cache
- ta má velikost cca. 16kB
- podívejme se ještě, co vše lze vektorizovat

Optimalizace pomocí vektorových instrukcí

Vektorizace podmínek:

```
1  void alignedVectorMin( const float* __restrict__  v1,
2                         float* __restrict__ sum )
3  {
4     float* _sum = ( float* ) __builtin_assume_aligned( sum, 16);
5     const float* _v1 = ( const float* ) __builtin_assume_aligned( v1, 16);
6     for( int i = 0; i < size; i ++ )
7     {
8        if( _sum[ i ] < _v1[ i ] )
9           _sum[ i ] = _v1[ i ];
10    }
11 }
```

- v tomto případě nám překladač žádnou vektorizaci nehlasí
- a ve výsledném kódu také žádná vektorová instrukce použita není

```
1  Dump of assembler code for function alignedVectorAddition(float const*, float const*, float*):
2     0x0000000000401080 <+0>:     xor     %eax,%eax
3     0x0000000000401082 <+2>:     nopw    0x0(%rax,%rax,1)
4     0x0000000000401088 <+8>:     movss   (%rdi,%rax,1),%xmm0
5     0x000000000040108d <+13>:    ucomiss (%rdx,%rax,1),%xmm0
6     0x0000000000401091 <+17>:    jbe     0x401098 <alignedVectorAddition(float const*, float const*, float*)+24>
7     0x0000000000401093 <+19>:    movss   %xmm0,(%rdx,%rax,1)
8     0x0000000000401098 <+24>:    add     $0x4,%rax
9     0x000000000040109c <+28>:    cmp     $0x40000,%rax
10    0x00000000004010a2 <+34>:    jne     0x401088 <alignedVectorAddition(float const*, float const*, float*)+8>
11    0x00000000004010a4 <+36>:    repz retq
```

- důvodem je, že pokud není splněna podmínka `if( _sum[ i ] < _v1[ i ] )`, překladač nemá nic dělat
- vektorové instrukce neumí deaktivovat část XMM registru
- bylo by sice možné generovat kód `_sum[ i ] = _sum[ i ]`, ale to překladač nemůže
- obecně by to nebylo bezpečné v případu běhu více vláken

Optimalizace pomocí vektorových instrukcí

- úprava kódu na tvar

```
1   for( i = 0; i < size; i ++ )
2   {
3       if( _sum[ i ] < _v1[ i ] )
4           _sum[ i ] = _v1[ i ];
5       else
6           _sum[ i ] = _sum[ i ];
7   }
```

- ale nepomůže
- překladač zřejmě vidí příkaz `_sum[ i ] = _sum[ i ]` jako zbytečný a eliminuje ho ještě před analýzou možné vektorizace
- použijeme operátor `?`

Optimalizace pomocí vektorových instrukcí

```
1  void alignedVectorMin( const float* __restrict__  v1,
2                         float* __restrict__ sum )
3  {
4     float* _sum = ( float* ) __builtin_assume_aligned( sum, 16);
5     const float* _v1 = ( const float* ) __builtin_assume_aligned( v1, 16);
6     for( int i = 0; i < size; i ++ )
7        _sum[ i ] = _sum[ i ] < _v1[ i ] ? _v1[ i ] : _sum[ i ];
8  }
```

- výsledkem je již kód, který obsahuje instrukci `maxps`

```
1  Dump of assembler code for function alignedVectorAddition(float const*, float const*, float*):
2     0x0000000000401080 <+0>:    xor    %eax,%eax
3     0x0000000000401082 <+2>:    nopw   0x0(%rax,%rax,1)
4     0x0000000000401088 <+8>:    movaps (%rdi,%rax,1),%xmm0
5     0x000000000040108c <+12>:   maxps  (%rdx,%rax,1),%xmm0
6     0x0000000000401090 <+16>:   movaps %xmm0,(%rdx,%rax,1)
7     0x0000000000401094 <+20>:   add    $0x10,%rax
8     0x0000000000401098 <+24>:   cmp    $0x40000,%rax
9     0x000000000040109e <+30>:   jne    0x401088 <alignedVectorAddition(float const*, float const*, floa
10    0x00000000004010a0 <+32>:   repz retq
```

Optimalizace pomocí vektorových instrukcí

- nyní zkusíme tento příklad:

```
1    for( i = 0; i < size; i ++ )
2       _sum[i] = ((_sum[i] > _v1[i]) ? _sum[i] + _v1[i] : _sum[i]);
```

- ukáže se, že toto překladač nedokáže vektorizovat
- když ale zápis upravíme na tento tvar

```
1    for( i = 0; i < size; i ++ )
2       _sum[i] += ((_sum[i] > _v1[i]) ? _v1[i] : 0 );
```

- vektorizace se již provede

```
1  Dump of assembler code for function alignedVectorAddition(float const*, float const*, float*):
2     0x0000000000401080 <+0>:    xor     %eax,%eax
3     0x0000000000401082 <+2>:    nopw    0x0(%rax,%rax,1)
4     0x0000000000401088 <+8>:    movaps  (%rdi,%rax,1),%xmm2
5     0x000000000040108c <+12>:   movaps  (%rdx,%rax,1),%xmm1
6     0x0000000000401090 <+16>:   movaps  %xmm2,%xmm0
7     0x0000000000401093 <+19>:   cmpltps %xmm1,%xmm0
8     0x0000000000401097 <+23>:   andps   %xmm2,%xmm0
9     0x000000000040109a <+26>:   addps   %xmm1,%xmm0
10    0x000000000040109d <+29>:   movaps  %xmm0,(%rdx,%rax,1)
11    0x00000000004010a1 <+33>:   add     $0x10,%rax
12    0x00000000004010a5 <+37>:   cmp     $0x40000,%rax
13    0x00000000004010ab <+43>:   jne     0x401088 <alignedVectorAddition(float const*, float const*, float*)+8>
14    0x00000000004010ad <+45>:   repz retq
```

Optimalizace pomocí vektorových instrukcí

Výpočet sumy:

```
1  float alignedVectorSum( const float* v1 )
2  {
3     const float* _v1 = ( const float* )
4        __builtin_assume_aligned( v1, 16);
5     float y( 0.0 );
6     for( int i = 0; i < size; i ++ )
7        y += _v1[ i ];
8     return y;
9  }
```

- toto překladač vektorizovat neumí
- důvodem je, že by nedokázal dodržet přesně stejné pořadí, v němž se prvky vektoru sčítají a tím pádem by mohl narušit přesnost výpočtu
- řešením je použití přepínače `-ffast-math`

Optimalizace pomocí vektorových instrukcí

- ukazuje se, že pro úspěšnou vektorizaci kódu je podstatný tvar těla smyčky
- pokud překladač nedokáže vektorizaci provést, je dobré se pokusit zapsat výraz jinak
- můžeme se ale dostat do situace, kdy překladač nedokáže kód vektorizovat vůbec
- pak musíme vektorizaci provést sami

Optimalizace pomocí vektorových instrukcí

- pro vektorizaci svépomocí lze využít tzv. *intrinsic instructions*
- jsou definovány v následujících hlavičkových souborech

- `<mmintrin.h>` – MMX
- `<xmmintrin.h>` – SSE
- `<emmintrin.h>` – SSE2
- `<pmmintrin.h>` –SSE3
- `<tmmintrin.h>` – SSSE3
- `<smmintrin.h>` – SSE4.1
- `<nmmintrin.h>` – SSE4.2
- `<ammintrin.h>` – SSE4A
- `<avxintrin.h>` – AVX
- `<avx2intrin.h>` – AVX2
- `<immintrin.h>` – obecný

Optimalizace pomocí vektorových instrukcí

- tyto instrukce pracují s typy tvaru `__mXXX[ ,i,d]`
- kde `XXX` udává velikost registru, se kterým chceme pracovat
  - `XXX = 64,128,256,512`
- koncovka udává číselný typ
  - nic → `float`
  - `i` → `int`
  - `d` → `double`

Optimalizace pomocí vektorových instrukcí

```
1   float alignedSum( float* v,
2                     const int size )
3   {
4       __m128 s1p = _mm_setzero_ps();
5       for( int i = 0; i < size; i += 4 )
6       {
7           __m128 v1p = _mm_load_ps( &v[ i ] );
8           s1p = _mm_add_ps( s1p, v1p );
9       }
10      float res[ 4 ] __attribute__( ( aligned( 16 ) ) );
11      _mm_store_ps( res, s1p );
12      return ( res[ 0 ] + res[ 1 ] ) + ( res[ 2 ] + res[ 3 ] );
13  }
```

- proměnné `s1p` a `v1p` jsou uloženy v XMM registrech a každá obsahuje čtyři hodnoty typu `float`
- instrukce `_mm_setzero_ps()` nuluje XMM registr
- instrukce `_mm_load_ps( float* )` načte do příslušného registru čtyři po sobě jdoucí čísla typu `float` a adresa prvního musí být dělitelná 16
- instrukce `_mm_add_ps(_mm128 xmm0, _mm128 xmm1)` přičte do registru `xmm0` hodnotu z registru `xmm1`
- instrukce `_mm_store_ps( float*, _m128 xmm0 )` uloží na danou adresu obsah registru `xmm0`

Vektorizace - shrnutí

- současné překladače dokáží využít vektorové instrukce dobře
- někdy je ale potřeba kód vhodně modifikovat
- to často umožní vektorizaci tam, kde by ji překladač nezvládl provést
- někdy to může vést k výrazně menšímu strojovému kódu, což může zlepšit efektivitu práce s instrukční cache
  - kód je ovšem méně robustní a náchylnější k chybám
- v některých případech můžeme být nuceni provést vektorizaci sami pomocí *intrinsic instructions*
- vektorizace je velmi důležitá při programování MIC akcelerátorů Xeon Phi