Arquiteturas de Computadores

Licenciatura em Engenharia Informática

Arquiteturas vetoriais

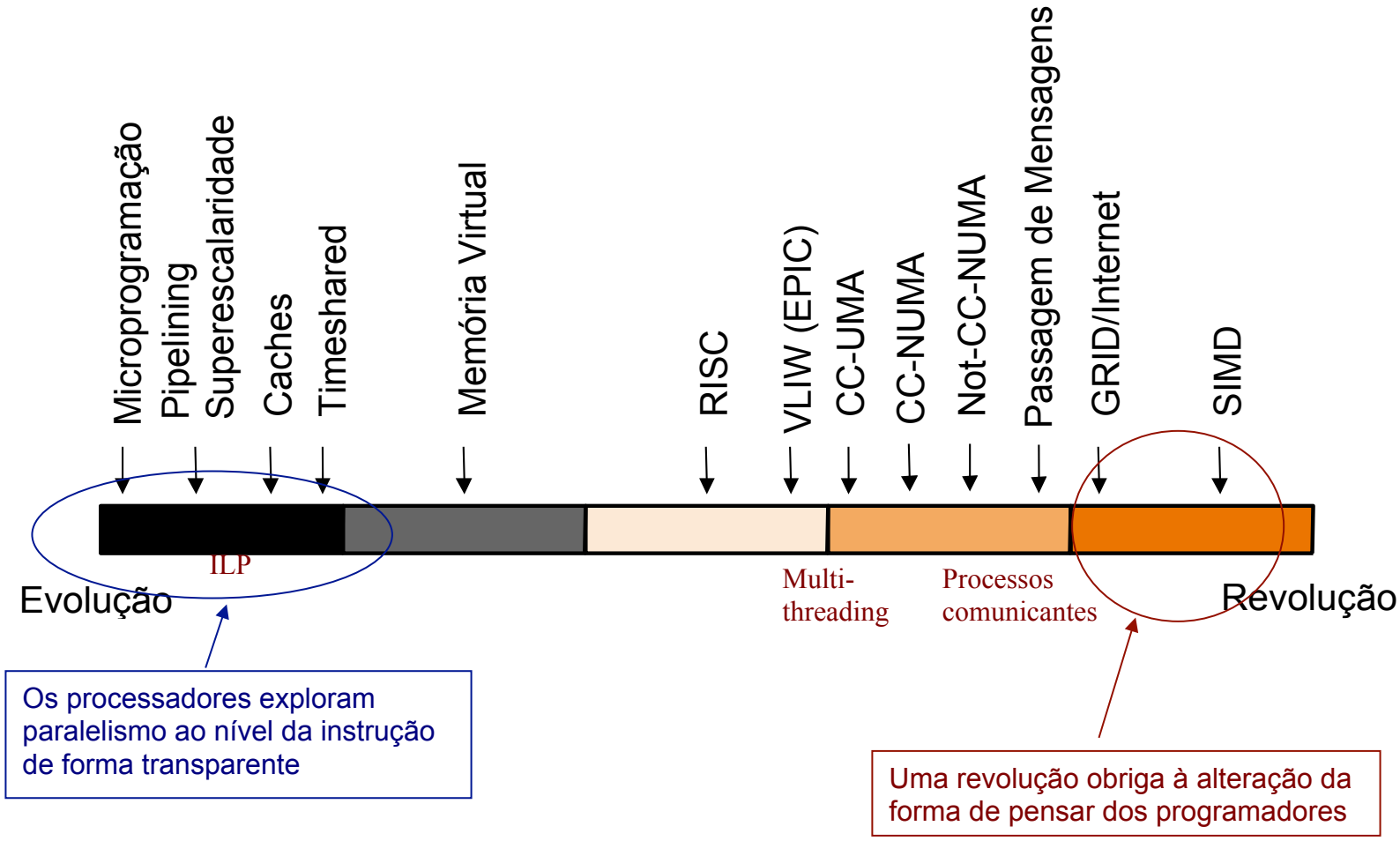
João Luís Ferreira Sobral

jls@...

Arquiteturas de computadores

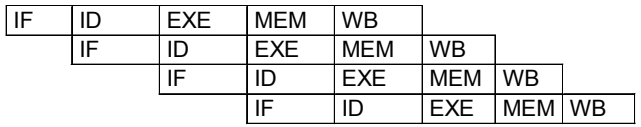
Conteúdos	**4 – Otimização de desempenho**	
	4.3 – Processamento vetorial	
Resultados de Aprendizagem	R4.2 – Identificar oportunidades de exploração de processamento vetorial na execução de programas	

Arquiteturas de computadores

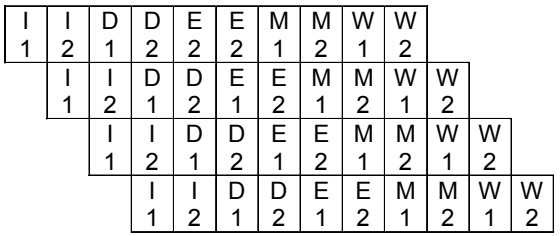


Arquiteturas de computadores

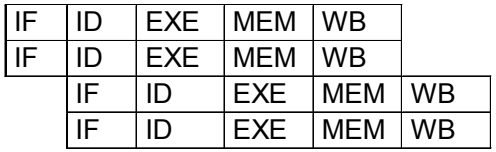
- ***Pipeliling* –** Executar as instruções de forma encadeada



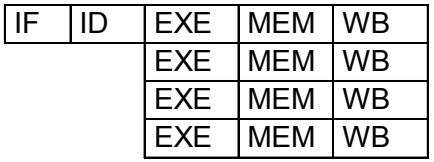
- ***Super-pipelining* –** Aumentar o número de estágios para possibilitar um aumento da frequência interna do relógio



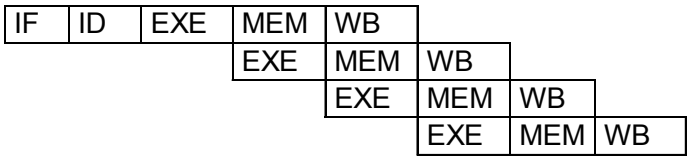
- **Super-escalar –** Iniciar várias instruções por ciclo



- ***Very Long Instruction Word* (VLIW) –** Especificar várias operações por instrução



- **Instruções vetoriais -** Especificar uma série de operações a realizar em vários dados, numa só instrução (SIMD – **S**ingle **I**nstruction **M**ultiple **D**ata)



Arquiteturas de computadores

Processadores Vetoriais

- **Incluem unidades dedicadas à execução de operações sobre vectores.**
  - **Ex. efetuar uma adição de dois vectores de 64 elementos**
    - **Equivalente a iterar sobre os elementos do vector, adicionando um elemento de cada vetor por iteração**

- **Vantagens das operações vetoriais**
  - **Evita anomalias de dados :** o cálculo de um resultado é independente dos anteriores, possibilitando a utilização de *pipeline* com muitos estágios,
  - **Reduz a quantidade de buscas de instruções:** uma só instrução especifica um grande número de operações, equivalente à execução de um ciclo
  - **Amortiza os custos dos acessos à memória:** o carregamento dos elementos do vector em registos pode tirar partido do débito da memória
  - **Reduz as anomalias de controlo:** os ciclos são transformados numa só instrução.

Arquiteturas de computadores

Processadores Vetoriais

- **Exemplo: calcular Y = α X + Y** **Y, X - vectores de 64 elementos**
  α – escalar
  - **Unidade de processamento vectorial e registos de 64 valores**

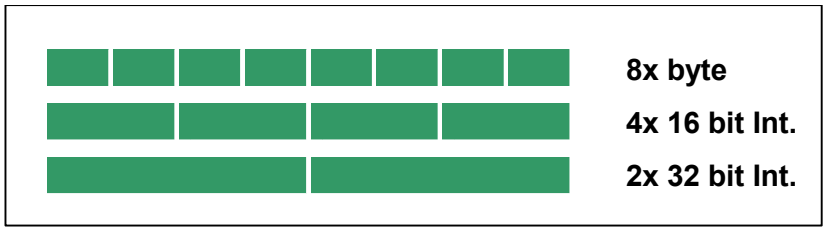
Y86	Y86 vetorial (pseudo avx)

movl alfa(%rip), %ecx
movl $0, %eax
.L2:
movl %ecx, %edx # **α**
imull X(%eax), %edx # **αX[i]**
addl %edx, Y(%eax) # **Y[i]+=αX[i]**
addl $4, %eax
cmpl $256, %eax
jne .L2
	vbroadcastss alfa(%rip), %xmm0 # **α**
	vpmulld X(%rip), %xmm0, %xmm1 # **αX**
	vpaddd Y(%rip), %xmm1, %xmm1 # **αX + Y**
	vmovdqa %xmm1, Y(%rip) # **Y = …**

Arquitecturas de computadores

Vetoriais para multimédia: instruções MMX, SSE de IA-32

- **MMX – 57 instruções operando sobre registos de 64 bits**
  - **Os registos podem representar vectores com 2 valores inteiros de 32 bits, 4 valores inteiros de 16 bits ou 8 valores inteiros de 8 bits:**



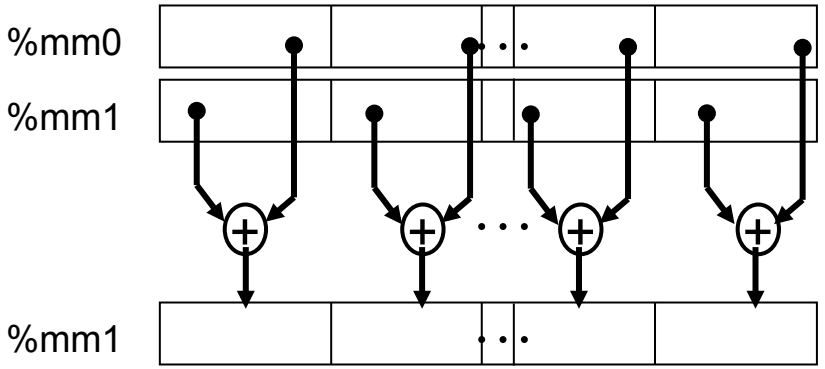
  - **Reutiliza os 8 registos FP (Não há mistura entre MMX e FP)**
- **Contém instruções para ler e escrever vectores na memória, aritméticas e lógicas entre vectores e conversão entre tipos de vectores:**
  - ***Load* e *store* de 32 ou 64 bytes**
  - ***Add*, *sub* em paralelo: 8 x 8 bits, 4 x 16 bits ou 2 x 32 bits**
  - **Deslocamentos (*sll*, *srl*), *And*, *And Not*, *Or*, *Xor* em paralelo**
  - **Multiplicação e *mult-add* em paralelo**
  - **Comparações em paralelo (=,>)**
  - ***Pack* – conversão entre tipos 32b <->16b, 16b <->8b**

Arquitecturas de computadores

Vectoriais para multimédia: instruções MMX, SSE e SSE2 de IA-32

- **MMX – Exemplo** **SIMD** – *Single Instruction Multiple Data*
  - **PADDB mm1, mm0**
    - **adiciona dois registos mmx, onde cada registo contém 8 operandos de 1 byte**
    - **relativamente a um add normal não existe *carry* entre grupos de 8 bits**

```
mm1 [7..0]   = mm0[7..0]     + mm1[7..0]
mm1 [8..15]  = mm0[8..15]    + mm1[8..15]
mm1 [16..23] = mm0[16..23] + mm1[16..23]
…
mm1 [63..56] = mm0[63..56] + mm1[63..56]
```



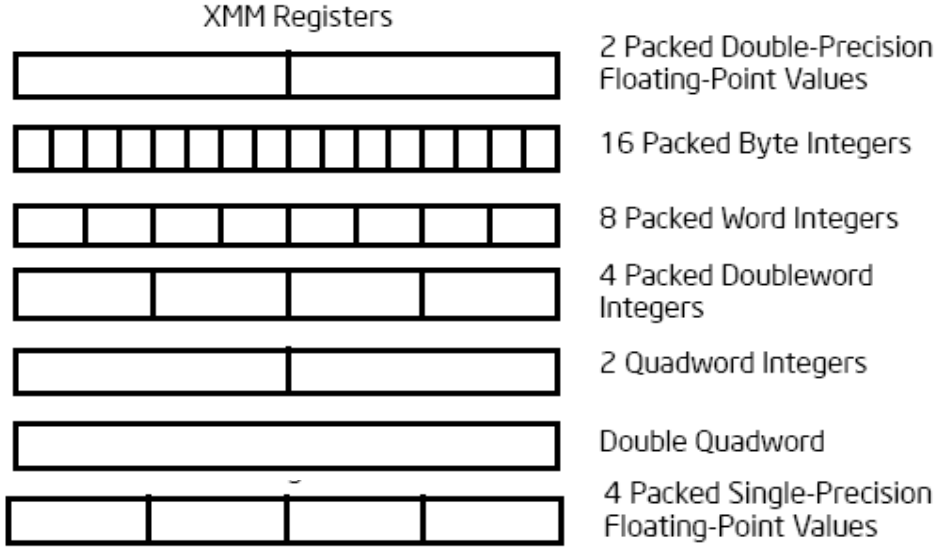
  - **PADDSB - adição com saturação**
    - **quando existe *carry* o byte correspondente é colocado com o valor máximo**
    - **útil, por exemplo, para efetuar a adição de duas imagens, para que os pixels que excedem o valor máximo fiquem com esse valor**

Arquitecturas de computadores

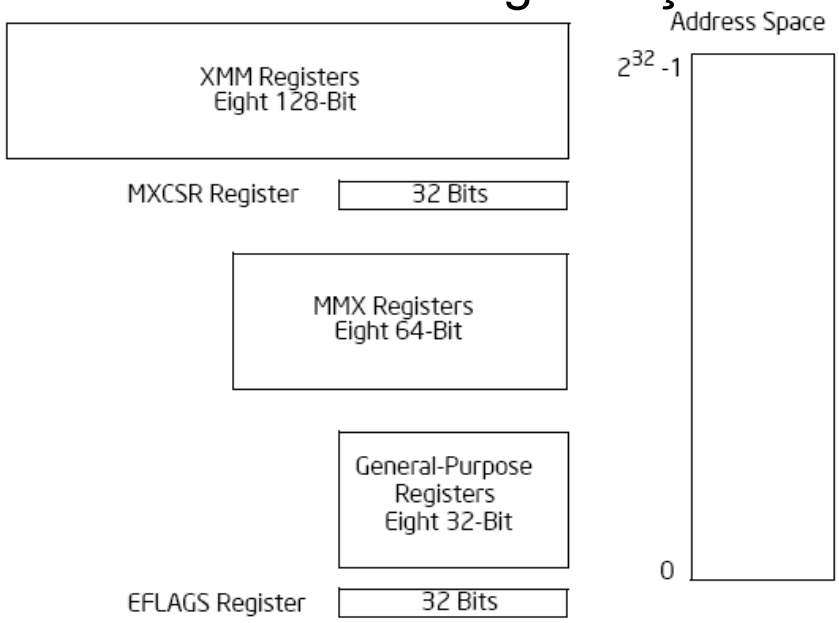
Vectoriais para multimédia: instruções MMX, SSE, AVX de IA-32

- SSE (Streaming SIMD Extensions) - introduz registos de 128 bits (xmm0..xmm7) e suporte a operandos em vírgula flutuante.
- AVX (Advanced Vector eXtensions) – registos de 256 bits (ymm0..ymm7)

Tipos de Dados



Ambiente de Programação



Instruções: Transferência de Dados

Instruções	**Operandos** orig, dest	**Obs.**
MOVQ	mm/m64, mm mm, mm/m64	Mover palavra quádrupla (8 bytes) de memória para registo mmx ou de registo mmx para memória (Apenas para inteiros)
MOVDQA MOVDQU	xmm/m128, xmm xmm, xmm/m128	Mover 2 palavras quádruplas (2*8 bytes) Apenas para inteiros A - addr alinhado M16; U – addr não alinhado
MOVAP[S|D] MOVUP[S|D]	xmm/m128, xmm xmm, xmm/m128	Mover 4 FP precisão simples ou 2 FP precisão dupla A – addr alinhado M16 U – addr não alinhado

Instruções: Operações Inteiras

Instruções	**Operandos** orig, dest	**Obs.**

`PADD?`
`PSUB?`
`PAND?`
`POR?`
	`mm/m64, mm`
	`xmm/m128, xmm`
		Adição, subtracção, conjunção ou disjunção do tipo de dados indicado
		Operação realizada sobre o número de elementos determinado pelo registo+tipo de dados
		Endereços em memória alinhados
		O destino tem que ser um registo

? = B | W | D | Q

B – byte W – 2 bytes

D – 4 bytes Q – 8 bytes

Instruções: Operações FP

Instruções	**Operandos** orig, dest	**Obs.**

`ADDP?`
`SUBP?`
`MULP?`
`DIVP?`
`SQRTP?`
`MAXP?`
`MINP?`
`ANDP?`
`ORP?`
	`xmm/m128, xmm`	Operação sobre o tipo de dados indicado
		
		Operação realizada sobre o número de elementos determinado pelo tipo de dados (S = 4 ; D = 2)
		
		Endereços em memória alinhados
		
		O destino tem que ser um registo

? = S | D
S – precisão simples
D – dupla precisão

Exemplo MMX

```
int a[100], b[100], r[100];

func (int n, int *a, int *b, int *r) {
  int i;
  for (i=0 ; i<n ; i++)
    r[i] = a[i] + b[i];
}
```

```
func:
  …  // %ecx=i, %edx =n
     // %eax=a, %ebx=b, %esi=r
  movl $0, %ecx
ciclo:
  movq (%eax, %ecx, 4), %mm0
  paddd (%ebx, %ecx, 4), %mm0
  movq %mm0, (%esi, %ecx, 4)
  addl $2, %ecx
  cmpl %edx, %ecx
  jle ciclo
  …
```

Registo mm0 (64 bits)

Processa dois elementos por iteração

Exemplo SSE

```
__MM_ALIGN16 float a[100], b[100], r[100];

func (int n, float *a, float *b, float *r) {
  int i;
  for (i=0 ; i<n ; i++)
    r[i] = a[i] * b[i];
}
```

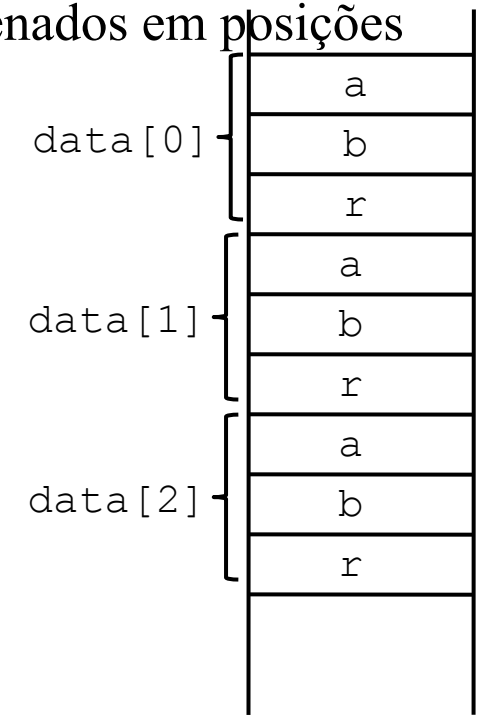
```
func:
  …
  movl $0, %ecx
ciclo:
  movaps (%eax, %ecx, 4), %xmm0
  mulps (%ebx, %ecx, 4), %xmm0
  movaps %xmm0, (%esi, %ecx, 4)
  addl $4, %ecx
  cmpl %edx, %ecx
  jle ciclo
  …
```

Registo xmm0 (128 bits)

Data layout – AoS versus SoA

Exemplo anterior: `movaps (%eax, %ecx, 4), %xmm0`

- Carrega 4 elementos do vector `b` para `%xmm0`
- Isto requer que estes 4 elementos estejam armazenados em posições consecutivas de memória



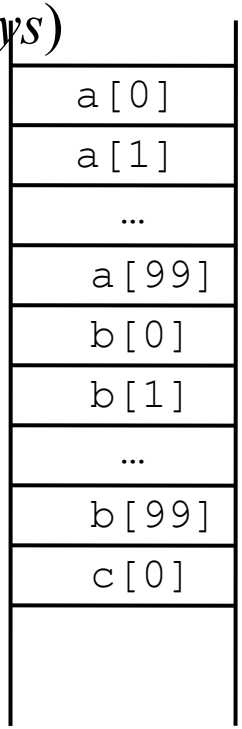
- O modelo habitual de programação usa vectores de estruturas (AoS) que resulta na dispersão dos elementos do mesmo vector:

```
struct {
  float a, b, r;
} data[100];
```

Data layout – AoS versus SoA

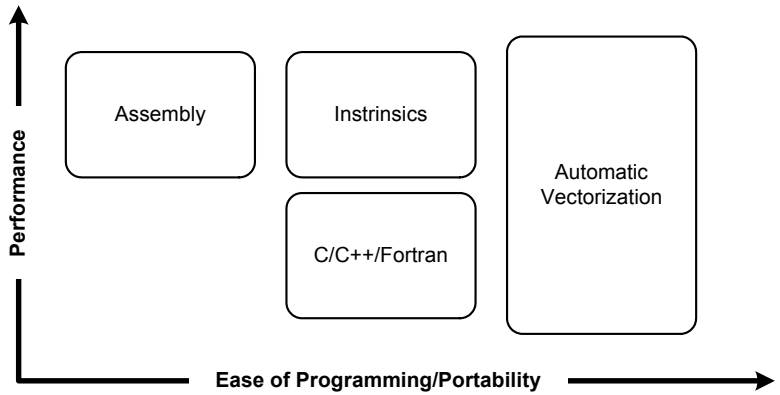
- Para que os vários elementos do mesmo campo (ou vector) sejam armazenados consecutivamente em mémória é necessário usar uma codificação do tipo (SoA – *Structure of Arrays*)

```
struct {
  float a[100], b[100], r[100];
} data;
```



Instruções vetoriais

Como gerar programas usando instruções vetoriais?



- Utilizar um compilador com suporte a "vetorização" automática
  - Opção –mavx no gcc
  - Apenas útil em programas simples
- Intrinsics
  - Estilo de codificação mais próximo do C
- Embutir *assembly* no programa

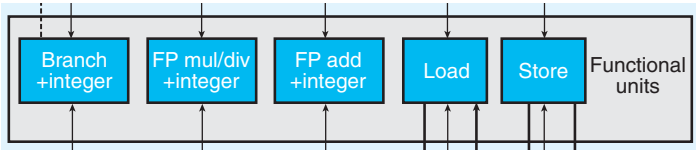
```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
   c[i] = a[i] + b[i];
 }
}
```

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

(pico de) Desempenho

Arquiteturas super-escalares

- i7 (Sandy Brigde)– execução (sustentada) de 4 instruções por ciclo (máximo de 5)

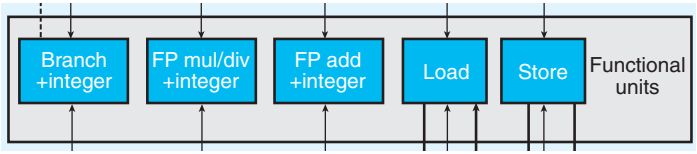


- Frequentemente caracteriza-se o desempenho (de pico) de uma arquitetura como o número de operações em vírgula flutuante por segundo (e.g., GFlops)
  - Um processador i7 tem a capacidade de executar, em cada ciclo:
    - 1 load + 1 store
    - 3 operações sobre inteiros (uma pode ser salto e/ou FP add e/ou FP mult)
  - Exemplo: 3.0 GHz => pico de 6,0 GFlops (1 *FP mul* + 1 *FP add* por ciclo)

- Principais fatores que impedem que o pico seja atingido numa aplicação:
  - *stalls* devido a *misses* de dados
  - *stalls* originados por dependências de controlo ou de dados
  - inexistência de balanço entre FP mul/div, FP add, Load/store, etc...

(pico de) Desempenho

Arquitetura vetorial



- Operações AVX
  - *Vmult* e *Vadd* partilham unidades funcionais com FP mult e FP add
  - O pico teórico é 8x superior à arquitetura sem instruções vetoriais (considerando vetores de 8 elementos, e.g., *floats*)
  - Exemplo:
    - 3.0 GHz => pico de 48,0 GFlops (1 *V mul* + 1 *V add* por ciclo)

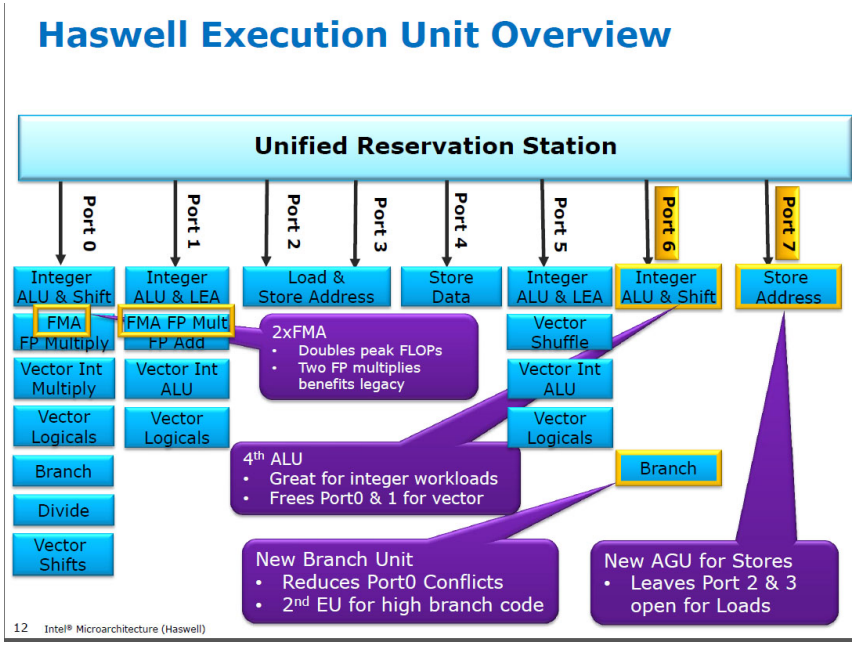
- Principais fatores que impedem que o pico seja atingido numa aplicação:
  - *stalls* devido a *misses* de dados (AVX pode implicar 8x mais dados)
  - *stalls* originados por dependências de controlo ou de dados
  - inexistência de balanço entre Vmul/div, Vadd, Load/store, etc...

  - Basicamente: todas as penalizações podem atingir uma grandeza superior

(pico de) Desempenho

Arquitetura vetorial

- Operações AVX2
  - introduzidas na microarquitetura Haswell
  - novas instruções de *Fused Mult/Add (FMA)*
    - VFMADD132PDy ymm$2, ymm$1, ymm$0
    - $0 = $0×$2 + $1
  - o pico teórico é 16x superior à arquitetura sem instruções vetoriais (2x Sandy Bridge)
  - Exemplo:
    - 3.0 GHz => pico de 96,0 GFlops (2 *VFMA* por ciclo)



- Principais fatores que impedem que o pico seja atingido numa aplicação:
  - *stalls* devido a *misses* de dados (AVX2 pode implicar 16x mais dados)
    - arquitetura Haswell duplica o largura de banda para leitura de dados da cache L1 (relativamente a Sandy Bridge)
  - *stalls* originados por dependências de controlo ou de dados
    - Apesar de continuar uma arquitetura de 4 vias, Haswell introduziu mais flexibilidade nas combinações possíveis de instruções (4 operações sobre inteiros, 2 *load*+1 *store*, 2ª porta para execução de saltos)

Arquitecturas de computadores

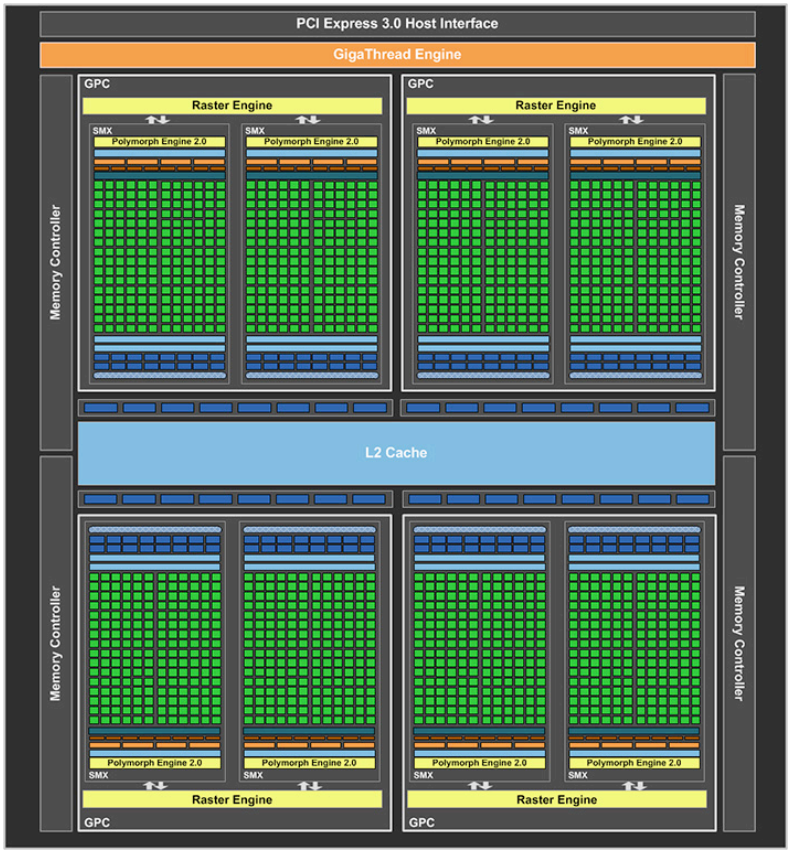
GPU (graphics processing units)

- Arquiteturas inicialmente projetadas para o processamento gráfico
  - E.g., geração de imagens (3D)
- Têm evoluído em termos de flexibilidade de programação, competindo, hoje, com arquiteturas vetoriais clássicas
  - Apresentam, tal como as arquiteturas vetoriais clássicas, picos de desempenho muito elevados
- São baseadas no modelo SIMD, introduzindo alguma flexibilidade adicional
  - Blocos de "cores" que podem executar diferentes fluxos de instruções
- Possuem um débito da memória extremamente elevado
  - Não possuem uma hierarquia de memória "clássica"

Arquitecturas de computadores

GPU (graphics processing units)

- Exemplo GTX 680
  - 1536 “cores” (a 1GHz):
    - 4 Graphics Processing Clusters (GPCs)
    - 8 Streaming Multiprocessor-X (SMX)
    - 192 “cores” por SMX
      - Executam a mesma instrução
      - Partilham 65K registos de 32 bits
      - Partilham uma memória local de 64 KB
    - Pico de desempenho:
      - 3 072 GFlops (quase 100x i7 c/AVX)
  - 192 GB/s de largura de banda memória



Arquitecturas de computadores

GPU (graphics processing units)

- Exemplo de programação

  (CUDA, extensão ao C da NVIDIA para GPUs)

  - **Calcular $Y = \alpha X + Y$**
    - Todos os cores executam o mesmo código
    - Cada "core" irá somar um valor
      - Valor que cada "core" soma é dado pelo seu id global (cada core terá um id único)

Versão CPU

```
void axpy(const int a,  const int *x, int *y, const int N){
   int idx;
   for (idx = 0; idx < N; idx++)
      y[idx] += a * x[idx];
}
```

Versão GPU

```
__global__ void *axpy(const int a,  const int *x, int *y, const int N){
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    y[idx]  += a * x[idx];
}
```

- Principais fatores que impedem que o pico seja atingido numa aplicação:
  - partes do algoritmo que não são intrinsecamente vetoriais
  - largura de banda da memória insuficiente
  - acessos à memória não alinhados
  - divergência nos saltos condicionais (porque é uma arquitetura SIMD)