Arquiteturas Paralelas: máquinas SIMD

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP

SIMD – *Single Instruction & Multiple Data*

- Arquiteturas SIMD considera processadores com
  - Uma Unidade de Controle controlando múltiplas Unidades Funcionais
    - ULA e FPU principalmente
  - Uma mesma instrução executando nas Unidades Funcionais com dados diferentes
  - Processador SIMD considera **execuções síncronas** sobre diferentes dados

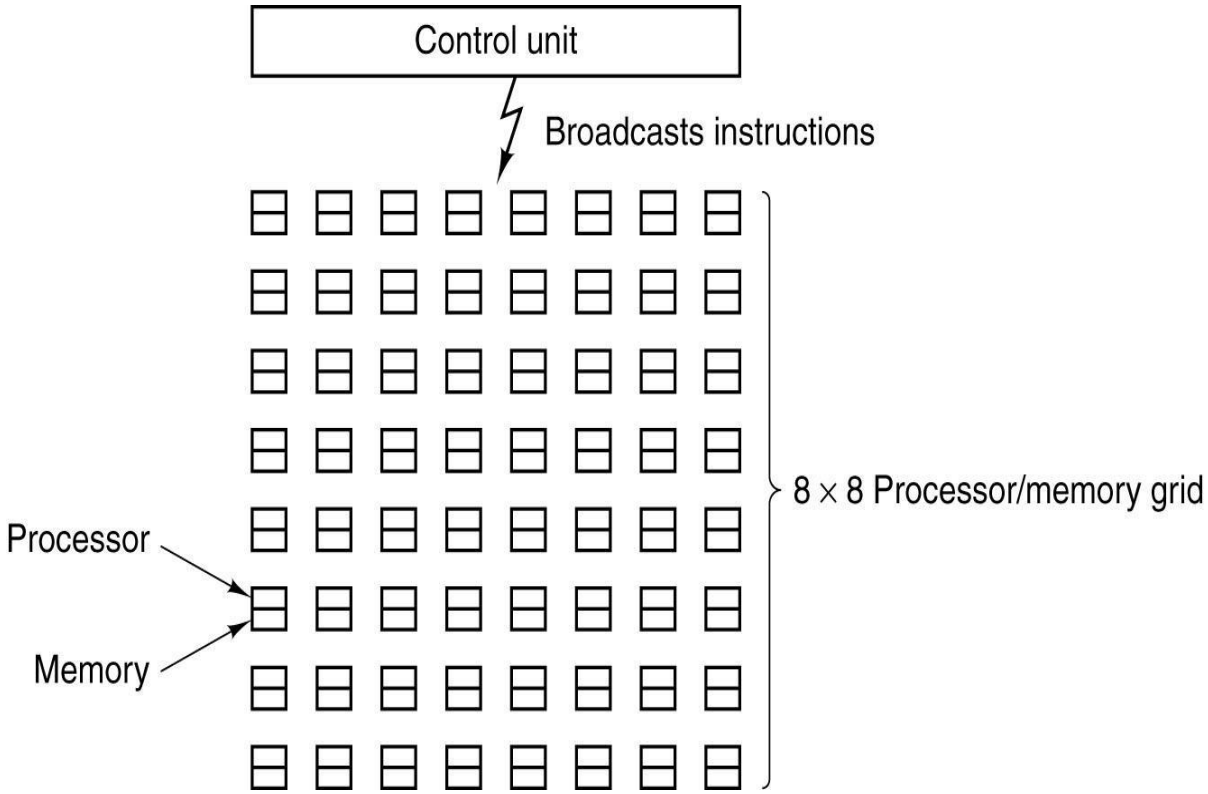
- Alto grau de paralelismo de dados (*supercomputadores*)
  - Operações sobre **vetores e matrizes de dados** (*float* e *double*)
    - **Maior precisão** e **vazão** de operações em ponto-flutuante
  - Voltadas para aplicações que têm essa demanda sobre dados em vetores

- São uma alternativa aos *mainframes*
  - Voltados para multiprogramação e operações intensas de E/S

- Dificuldade para algumas execuções
  if (b ==0) c = a; else c = a/b;
  - Uma única instrução opera sincronamente dados distintos
  - if() acima é executado em 2 passos

SIMD – *Single Instruction & Multiple Data*

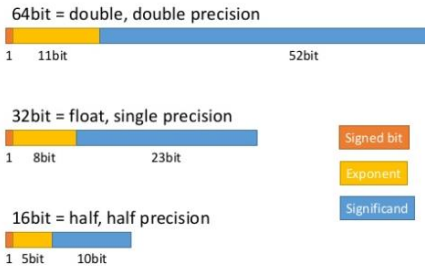
- Um exemplo básico de arquitetura: processadores SIMD do ILLIAC IV
  - Primeiro computador SIMD projetado pela *Universidade de Illinois*
    - Construído pela *Burroughs* em 6 anos, custou USD 40M
  - Primeiro a quebrar a visão sequencial das máquinas SISD em 1972



Tanenbaum (2013)

Computação Vetorial

- Supercomputadores vs Processadores Matriciais
  - Ambos executam operações sobre matrizes ou vetores de números de ponto flutuante (padrão IEEE 754)
  - Exemplos: $c_{i,i} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j}$ ou



Stallings (2013)

$$\begin{bmatrix} 1.5 \\ 7.1 \\ 6.9 \\ 100.5 \\ 0 \\ 59.7 \end{bmatrix} + \begin{bmatrix} 2.0 \\ 39.7 \\ 1000.003 \\ 11 \\ 21.1 \\ 19.7 \end{bmatrix} = \begin{bmatrix} 3.5 \\ 46.8 \\ 1006.093 \\ 111.5 \\ 21.1 \\ 79.4 \end{bmatrix}$$

$A \quad + \quad B \quad = \quad C$

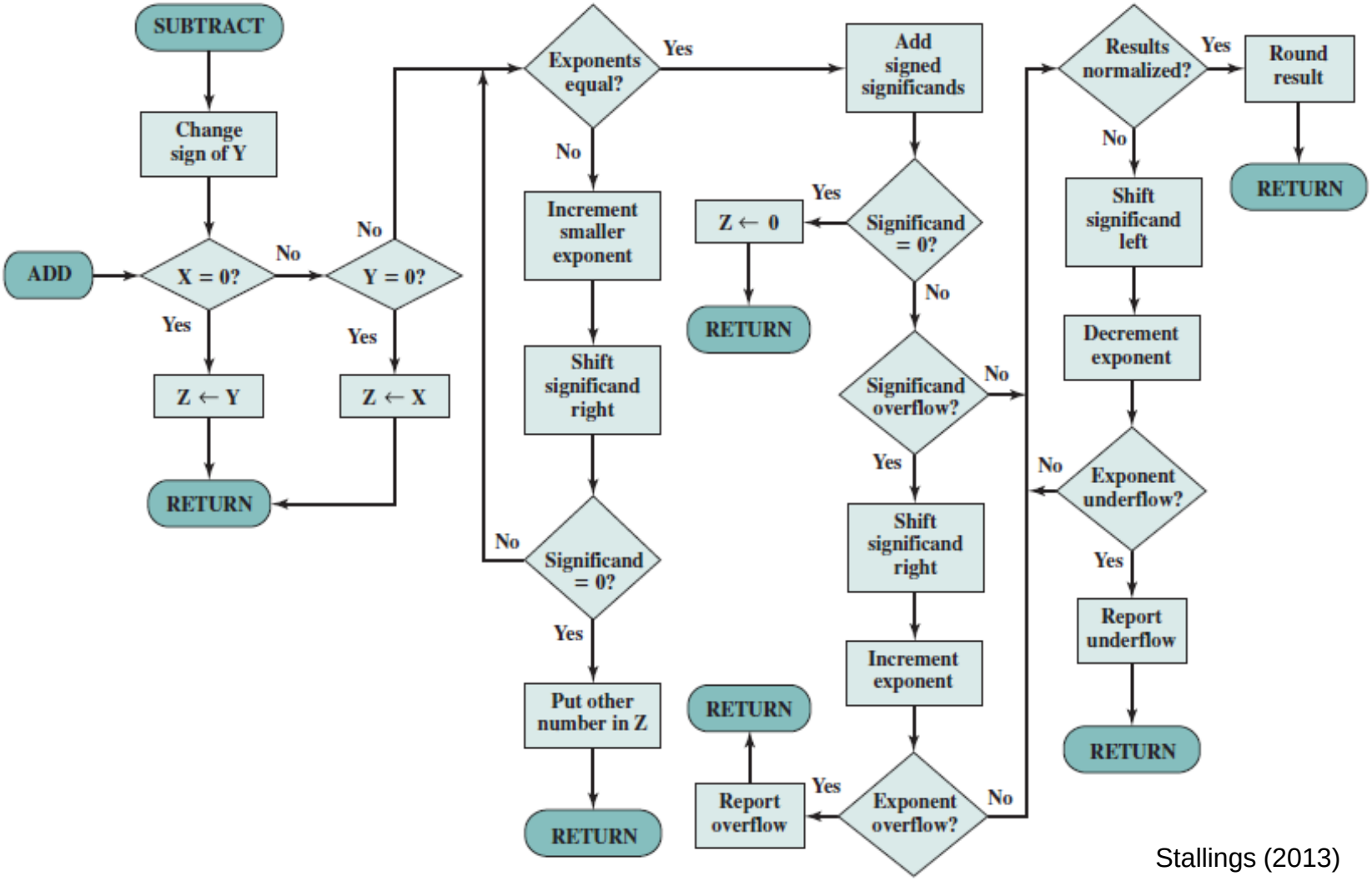
Figure 17.13 Example of Vector Addition

- **Operações com ponto-flutuante são mais complexas**
  - Exemplo (base 10): $1*10^0 + 1*10^3 = 1001*10^0$ ou $1{,}001*10^3$ ou ...
    - 04 passos básicos (visão simplificada)
      - 1º passo: compara expoentes => 0 é diferente de 3?
      - 2º passo: desloca expoentes => $0{,}001*10^3 + 1*10^3$
      - 3º passo: executa a operação => $1{,}001*10^3$
      - 4º passo: arredonda/normaliza => $1{,}001*10^3$ (já estava norm.)

- Há aplicações que operam muito sobre grandes vetores de *float/double*
  - **Otimizar essas operações produz ganhos expressivos de desempenho**

Computação Vetorial

- Aritmética com ponto-flutuante: alguns detalhes a mais.



Stallings (2013)

Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

Computação Vetorial

- Processadores Vetoriais
  - **Normalmente**
    são ULAs com pipeline
- Processadores Matriciais
  - **Normalmente**
    são ULAs paralelas
- Mas... a terminologia é confusa
  - Depende do autor... :-\
  - Todos permitem computação otimizada sobre vetores de dados numéricos de ponto flutuante

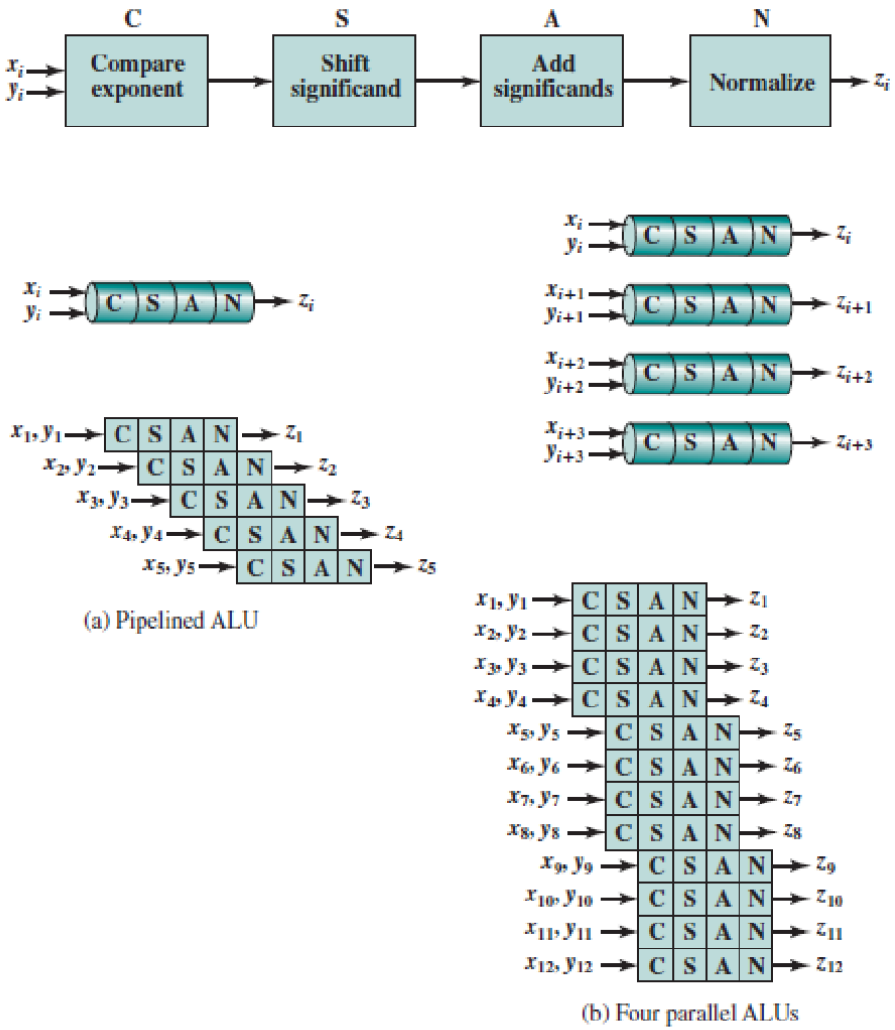
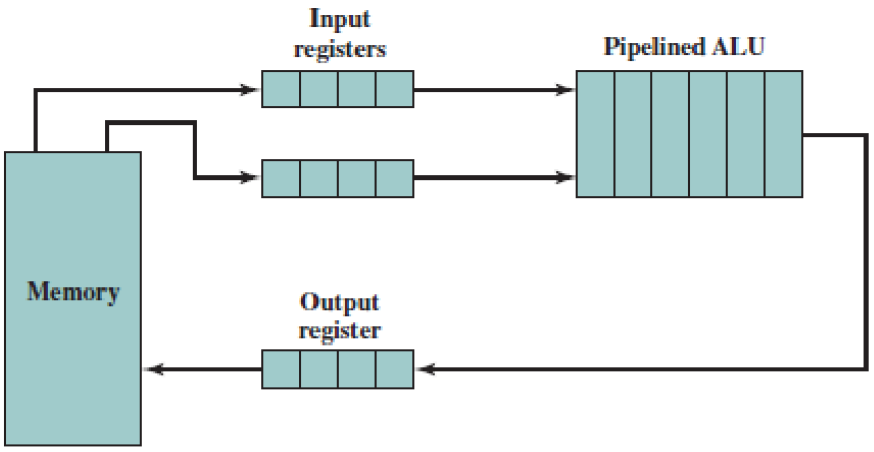


Figure 17.16 Pipelined Processing of Floating-Point Operations

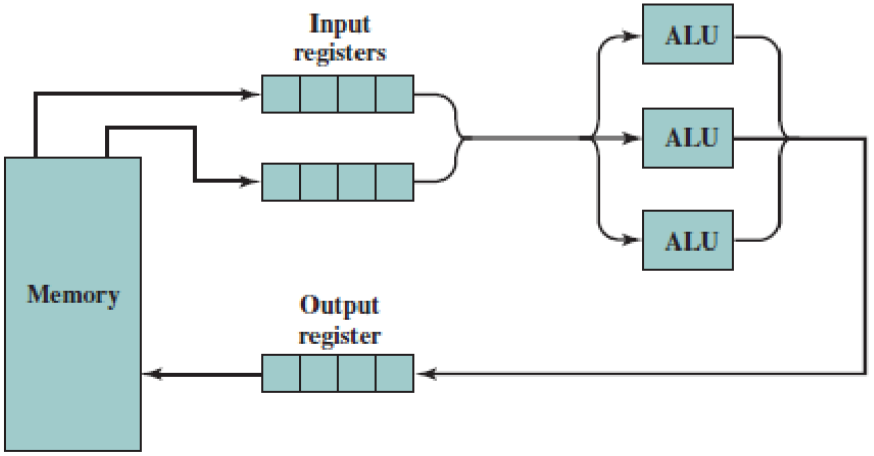
Stallings (2013)

Processadores para Computação Vetorial

- Processadores para processamento vetorial



(a) Pipelined ALU



(b) Parallel ALUs

Figure 17.15 Approaches to Vector Computation

Stallings (2013)

Exemplo do IBM 3090
Registradores vetoriais
ULAs vetoriais

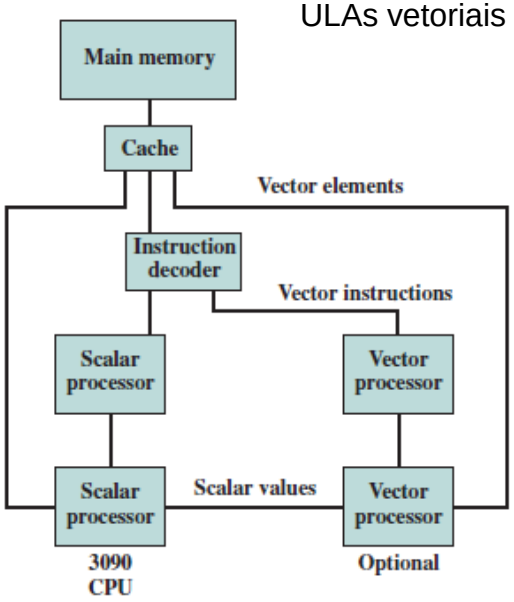


Figure 17.18 IBM 3090 with Vector Facility

Stallings (2013)

GPUs

- *Graphics Processing Unit* (GPU) desenvolvida pela NVIDIA
  - Coprocessador otimizado para
    - Processamento gráfico 2D e 3D
    - Computação visual
      - Interação visual em tempo real com gráficos, imagens e vídeos

- Arquitetura permite
  - Processamento gráfico programável
  - Processamento paralelo escalável

- PCs e consoles de jogos combinam CPUs e GPUs
  - Formam sistemas com processadores heterogêneos
  - Aceleradores gráficos complementam uma CPU escalar

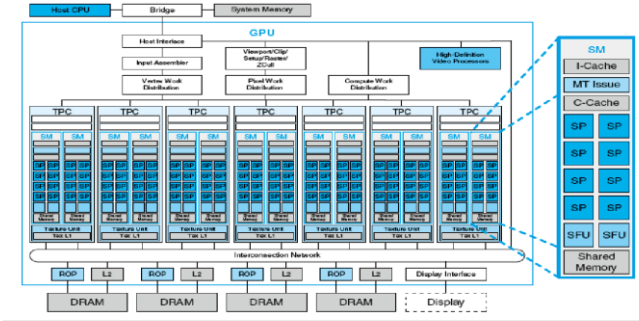
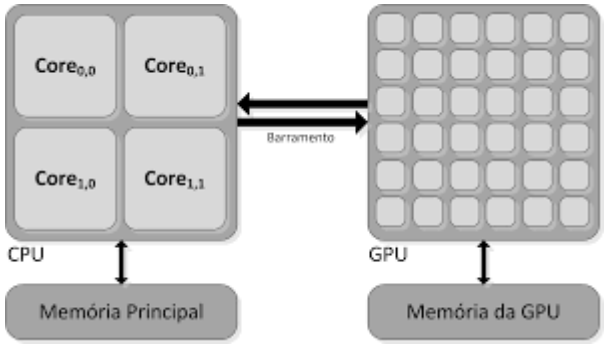


FIGURE C.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

Patterson & Hennessy (2014)



GPUs

- Conceitos gerais envolvidos
  - Oferecem suporte ao ganho de desempenho com *manycores*
  - Não usa múltiplos níveis de cache para esconder a latência de memória
    - Usa *mutithreading* (milhares de *threads*)
  - Foco no aumento da vazão; não na latência
    - Faz mais rápido um bloco de processamento de dados

  - Perdem tempo na transferência de dados da memória do computador hospedeiro para a memória da GPU
    - Princípios de localidade dos dados são fundamentais

- Modelos de programação:
  - GPU *Computing*
    - Uso da GPU como um co-processador para acelerar CPUs na computação de aplicações de propósito geral que envolve dados em vetores e matrizes

  - GPGPU
    - Uso da API gráfica da GPU para programar aplicações de propósito geral

  - CUDA
    - Modelo de programação paralela escalar em uma linguagem baseada em C/C++.
    - Plataforma de programação paralela para aceleradores e CPUs *multicore*

GPUs

- Uma estrutura genérica de GPU da NVIDIA

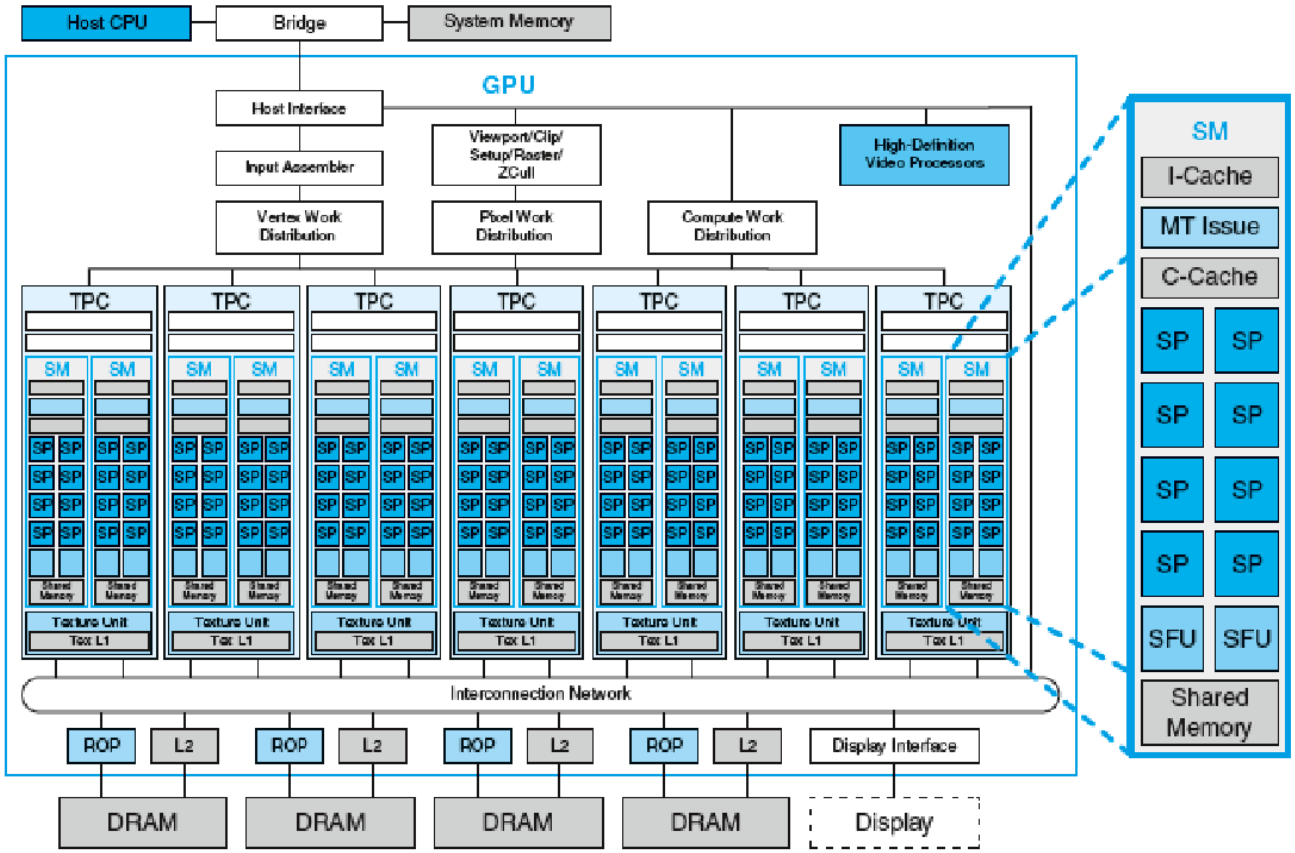
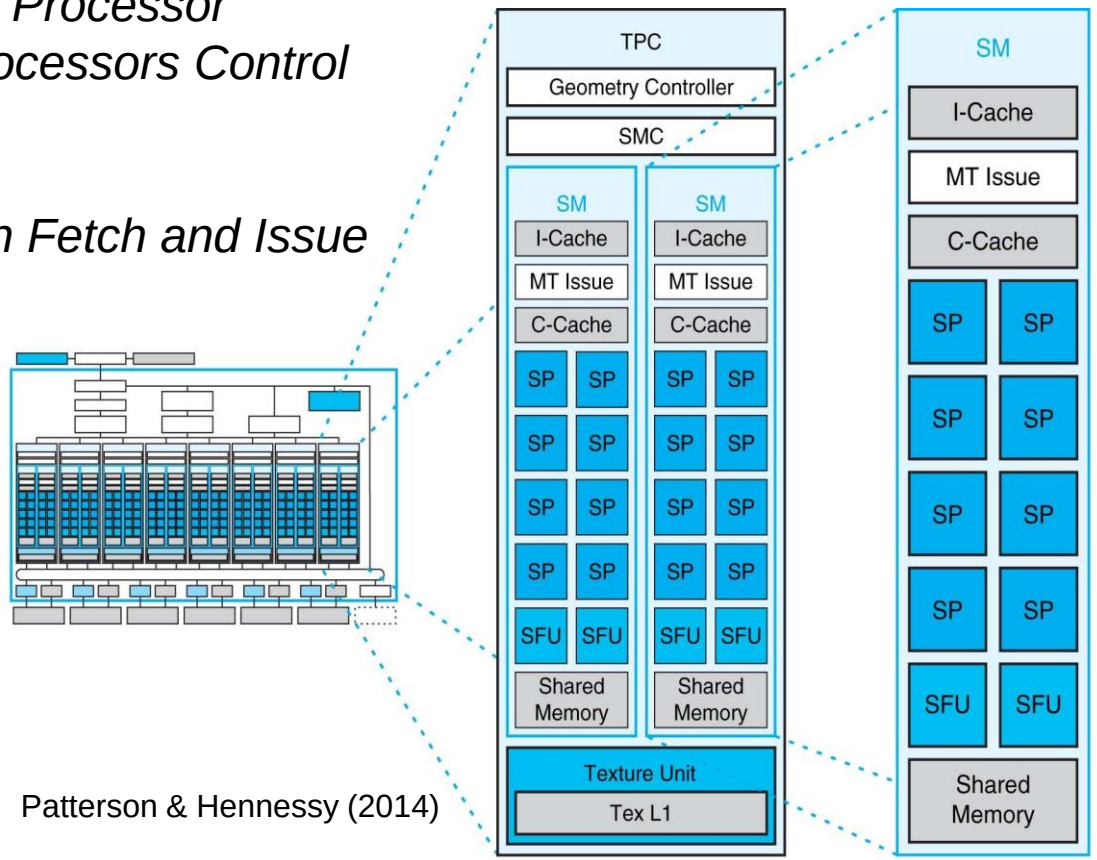


FIGURE C.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

Patterson & Hennessy (2014)

GPUs

- Uma estrutura genérica de GPU da NVIDIA
  - TPC – *Texture/Processor Cluster*
  - SMs – *Streaming Multiprocessors*
  - SP (ou SPA) – *Streaming Processors Array*
  - SFU – *Special Function Unit*
  - ROP – *Raster Operations Processor*
  - SMC – *Streaming Multiprocessors Control*
  - *Geometry Controller*
  - MT Issue –
    Multithreaded Instruction Fetch and Issue



Patterson & Hennessy (2014)

GPUs

- SIMT (*Single Instruction Multiple Threads*)
  - Este modelo flexibiliza a visão SIMD de execução com múltiplas threads executando uma mesma instrução
  - SIMD tradicional tem uma thread que executa a mesma instrução com dados diferentes

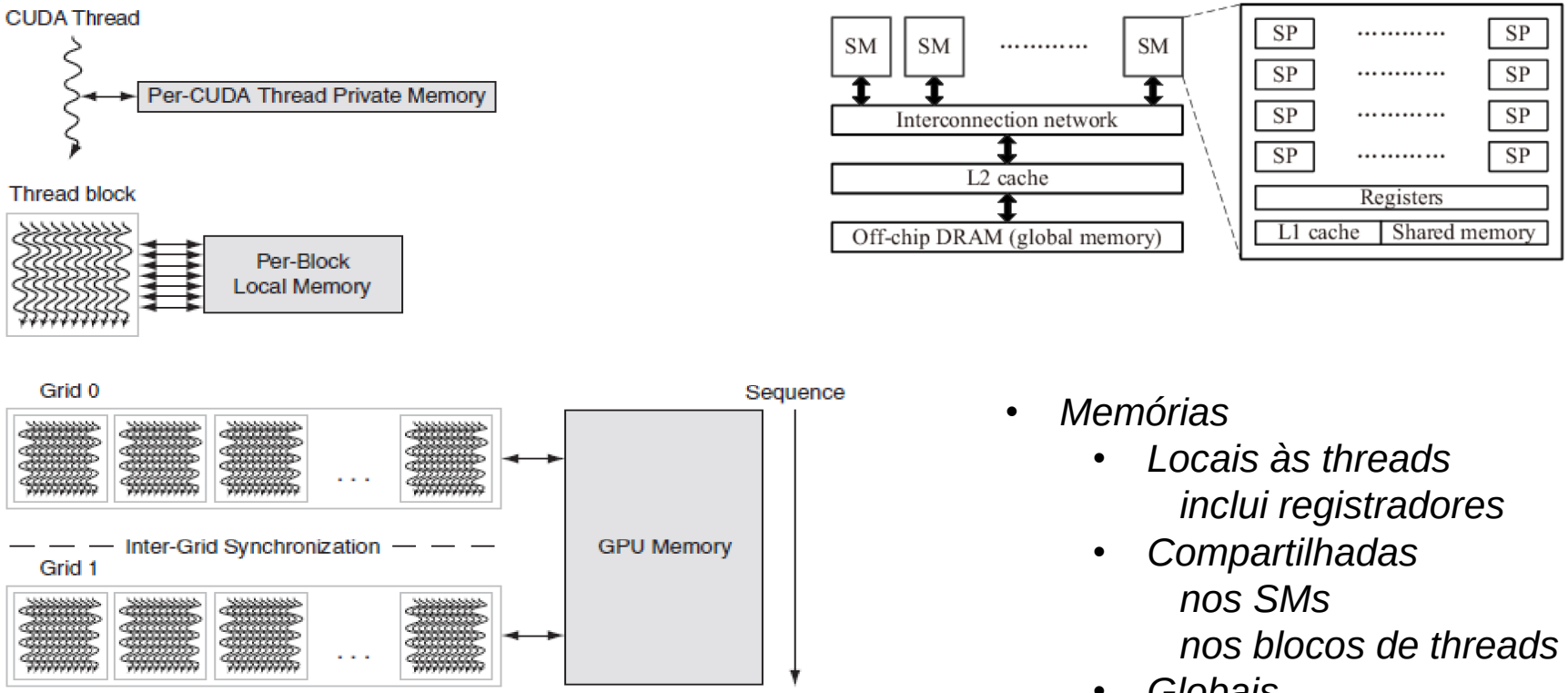


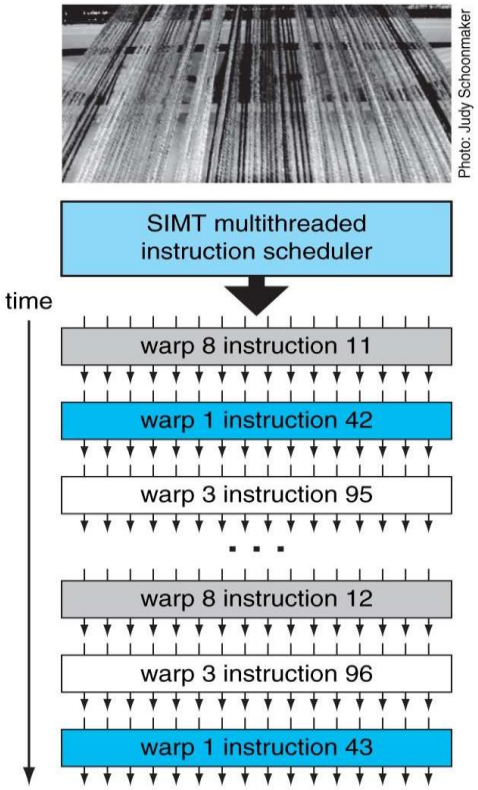
FIGURE 6.10 GPU Memory structures. GPU Memory is shared by the vectorized loops. All threads of SIMD instructions within a thread block share Local Memory.

Patterson & Hennessy (2014)

- *Memórias*
  - *Locais às threads*
    inclui registradores
  - *Compartilhadas*
    nos SMs
    nos blocos de threads
  - *Globais*
    à GPU
    às Grades

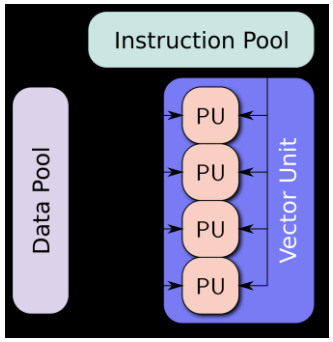
GPUs

- A GPU cria, controla, escalona e executa *threads* simultâneas em grupos chamados *warps*
  - Um ***warp*** é um conjunto de threads que executam a mesma instrução em diferentes elementos de dados
  - Caminhos distintos são serializados e o desempenho cai



Patterson & Hennessy (2014)

Extensões SIMD/Vetoriais nas CPUs



- CPUs modernas possuem extensões SIMD usualmente
  - Entender as extensões e o modelo paralelo SIMD
    - Permite que os processadores atinjam alto desempenho
    - Economiza energia para executá-los com alto desempenho

- Uso de instruções SIMD é um desafio para linguagens de alto nível
  - Linguagens de programação usuais não têm uma notação para usá-las
  - Compiladores tentam preencher essa lacuna e vetorizam código
    - Mas há sérias limitações devido à visão estática dos compiladores
      - Dependências de dados, dependências de controle, fluxo de dados, diferentes plataformas, ...
  - Alternativas
    - Programação *Assembly*
    - Compiladores específicos focados em recursos SIMD
    - Compiladores tradicionais que vetorizam código automaticamente
      - Esta é conhecida como autovetorização

- Alguns padrões tentam minimizar essa lacuna
  - OpenMP é um exemplo com o uso de diretivas de compilação
    - Indicam ao compilador e bibliotecas onde usar instruções SIMD

Extensões SIMD/Vetoriais nas CPUs

- Código vetorizado
  - Possui uma seção do código que faz uso eficiente de instruções SIMD
  - Múltiplas partes de dados são “empacotadas” juntas em um único e grande tipo de dado
    - Vetores e matrizes
    - Limites dos vetores/matrizes conhecidos e determinados pelo tipo do dado
  - Instruções SIMD atuam sobre este vetor de dados empacotados, modificando o vetor todo com uma única instrução do processador

- Instruções SIMD usam registradores com mais bits: 128, 256, 512, ...
  - Chamados de registradores vetoriais
  - Processadores x86 têm: ZMM, YMM, XMM e MM
  - Processadores ARM e PowerPC também têm seus registradores vetoriais
  - Conjunto de instruções vetoriais determinam como os regs são usados

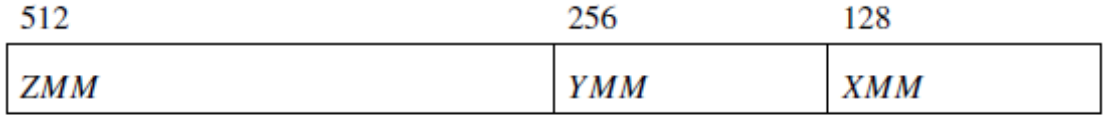


Figure 1. Layout of x86 vector registers

Joseph et al. (2017)

Extensões SIMD/Vetoriais nas CPUs

- Conjuntos de instruções vetoriais disponíveis nas CPUs x86

Instruction Set	Name	Functionality	Year
MMX	Multimedia Extensions	64 bit MMX for packed integers	1997
SSE	Streaming SIMD Extensions	128 bit XMM for floating point.	1999
SSE2	Steaming SIMD Extensions 2	XMM supports doubles and integers	2001
SSE3	Streaming SIMD Extensions 3	Horizontal operations added	2004
SSSE3	Supplemental SSE3	Horizontal and data movement	2006
SSE4.1	Streaming SIMD Extensions 4.1	Extra functionality	2007
SSE4.2	Streaming SIMD Extensions 4.2	Vector string instructions	2008
AVX	Advanced Vector Extensions	256 bit YMM for floating point.	2011
FMA	Fused Multiply Add	Fused multiply add instructions	2011
AVX2	Advanced Vector Extensions 2	YMM supports packed integers	2013
AVX512	Advanced Vector Extensions 512	512 bit ZMM registers	2016

Table 1. Available x86 vector instruction sets

Joseph et al. (2017)

- Quantia de dados que podem ser operados simultaneamente dependem do
  - Tamanho dos dados
  - Tamanho do registrador vetorial usado

- Comprimento do vetor (***vector lenght***)
  - Número de elementos que podem ser usados em um registrador vetorial

Register	long double	double	float	long	int	short	char
64 bit MM	–	–	–	1	2	4	8
128 bit XMM	–	2	4	2	4	8	16
256 bit YMM	–	4	8	4	8	16	32
512 bit ZMM	–	8	16	8	16	32	64

Table 2. Vector lengths for different vector registers and data types

Joseph et al. (2017)

Extensões SIMD/Vetoriais nas CPUs

- Vetorização em um loop é o mesmo que *unrolling* (desenrolar/sequencializar) este loop

- No exemplo abaixo, se os registradores vetoriais XMM de 128 bits são usados
  - 04 *floats* podem ser operados ao mesmo tempo
  - Loop pode ser desenrolado por um fator de 4
    - Loop extra necessita operar posições finais caso o número de iterações não for divisível pelo fator 4
  - As quatro operações da iteração podem ser vetorizadas, reduzindo o tempo de resposta
    - Instruções de carga e escrita são feitas em blocos com 04 valores
  - *Potencial Speedup* de 4

```
float A[SIZE], B[SIZE];
for (int i = 0; i < SIZE; i++){
    A[i] += B[i];
}
```

```
float A[SIZE], B[SIZE];
int i;
for (i = 0; i < SIZE - 4; i += 4)
    A[i] += B[i];
    A[i+1] += B[i+1];
    A[i+2] += B[i+2];
    A[i+3] += B[i+3];
}
for (i = i; i < SIZE; i++){
    A[i] += B[i]
}
```

Joseph et al. (2017)

Extensões SIMD/Vetoriais nas CPUs

- Maiores comprimentos de vetor (***vector lenght***) permitem *speedups* maiores
  - No exemplo anterior com *float*: YMM 8x, ZMM 16x, ...
  - CPUs *multithrething* potencializam os ganhos de desempenho
    - 16 threads permitem, em tese, ganho de 16x
    - Se cada *thread* usar registradores vetoriais ZMM de 512 bits, cada uma terá um ganho de 16x
    - O ganho seria de 256x em relação à versão escalar com uma *thread*

- Entender como o modelo SIMD funciona é imperativo para extrair este potencial ganho de desempenho das arquiteturas com extensões SIMD

Extensões SIMD/Vetoriais nas CPUs

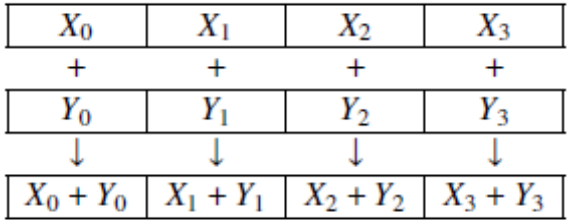
- Instruções SIMD no hardware operam sobre operandos com 8, 16, 32 e 64 bits
  - Em C isso corresponde a char, short, int, long

- Operações aritméticas básicas
  - add, sub, mult, and, or, xor, bit-shift, not, ...

- Conjunto exato, com suas limitações e disponibilidades depende da arquitetura

- Operações sobre registradores podem ser **verticais** ou **horizontais**

  - **Operações verticais** atuam em registradores diferentes



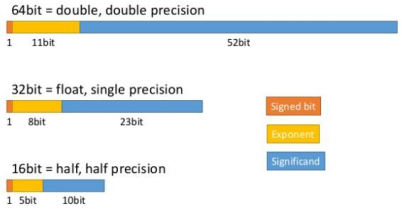
Joseph et al. (2017)

Figure 2. Vertical SIMD addition between four packed elements

  - **Operações horizontais** atuam em um mesmo registrador
    - Como em operações de redução

Extensões SIMD/Vetoriais nas CPUs

- Instruções SIMD de ponto flutuante operam com 32 e 64 bits:
  - *float* e *double* do C
  - São mais complexas que operações com inteiros (3 a 4x mais lentas)
    - Há instruções para sqrt, div e FMA (*Fused Multiply Add*) (X*Y+Z)
  - Têm o mesmo comportamento (vertical/horizontal)
    - funcionalidades variam com a arquitetura



- Instruções para **movimento de dados**:
  - Determinam se o uso de instruções SIMD melhoram o desempenho ou não
  - Dados podem vir para um registrador vetorial (de):
    - memória, registrador de propósito geral ou outro registrador vetorial

  - *Shuffle*: vetor como origem e rearranja valores (vlr imediato controla destino)
  - *Blended*: escolhe entre valores de dois registradores
  - *Unpacked*: instruções trabalhando com bits mais significativos ou menos significativos de um registrador vetorial
  - *Gathering (AVX2)*: *load* de posições disjuntas da memória
  - *Scattering (AVX2)*: *store* em posições disjuntas na memória

Extensões SIMD/Vetoriais nas CPUs

- Instruções para **operações condicionais**
  - Desvios precisam ser tratados com atenção em códigos vetorizados
    - Iterações diferentes de um loop podem ter *jumps* distintos
    - Impedem a sincronia requerida de uma instrução com dados diferentes
      - O *if* seta flag da comparação e determina uso do *jump*.
    - Solução: executar incondicionalmente as iterações, com instruções SIMD de comparação
- Instruções SIMD de comparação criam um *bit-mask* no registrador vetorial
  - Compara verticalmente cada elemento no registrador vetorial
    - *True* seta todos os bits para 1 e *False* seta todos para 0
    - Resultado pode ser usado com uma máscara AND e com isso executar todas as instruções do *if...then...else* juntas

```
float X[SIZE], Y[SIZE], Z[SIZE];
for (int i = 0; i < SIZE; i++){
    if (X[i] < Y[i]){
        Z[i] = X[i];
    }
    else {
        Z[i] = 0.0f;
    }
}
```

Joseph et al. (2017)

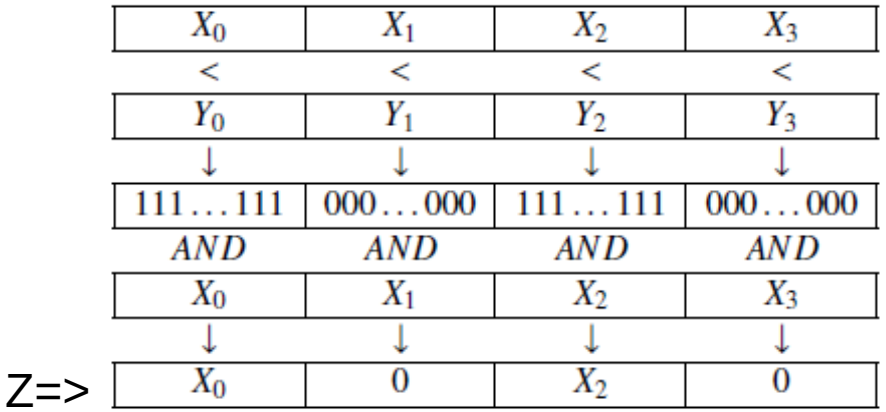


Figure 8. A conditional statement that returns X if true and 0 if false

Extensões SIMD/Vetoriais nas CPUs

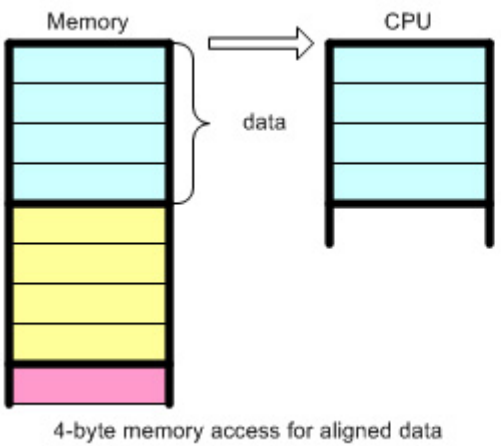
- Instruções para carga de **Constantes** em Registradores Vetoriais
  - Muitas arquiteturas têm restrições à carga de constantes
  - Usualmente registradores vetoriais são carregados com valores de/da
    - Memória ou
    - Registradores de propósito geral
  - As constantes conhecidas em tempo de compilação
    - são copiadas previamente em
      - Memória ou
      - Registradores de propósito geral

  - Após carga prévia em registrador ou memória, o valor da constante deve ser copiado para todas as posições válidas do tipo de dado no registrador vetorial

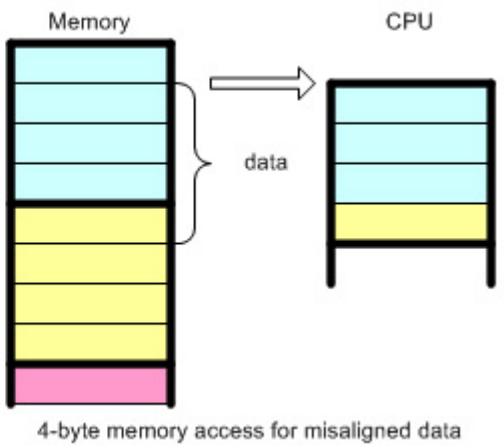
Extensões SIMD/Vetoriais nas CPUs

- **Alinhamento de Memória**
  - Divisibilidade de uma posição de memória em relação ao valor armazenado na memória
    - ***Int***: alinhamento de 32 bits, ***Double***: de 64 bits, ...
- Acessos a dados desalinhados são mais lentos
  - Prejudica uso da cache: leitura de blocos/linhas de dados
  - Também podem resultar em *seg-fault* a depender da arquitetura

- Em arquiteturas SIMD o tamanho dos registradores vetoriais é um ponto crítico
  - Usar este espaço com eficiência afeta *speedup*
  - Combinação do uso eficiente de cache com os registradores é fundamental



4-byte memory access for aligned data



4-byte memory access for misaligned data

Data Type	Alignment
char	1
short	2
int	4
long	8
float	4
double	8
long double	16
XMM register	16
YMM register	32
ZMM register	64

Table 3. memory alignment for each data type

Joseph et al. (2017)

Referências

Stallings, W.; Computer Organization and Architecture: Designing for Performance. Ninth Edition. Pearson. 2013.

Tanenbaum, A. S.; Austin, T.; Structured Computer Organization. Sixth Edition. Pearson. 2013.

Patterson, D. A.; Hennessy, J. L.; Computer Organization and Design: the hardware / software interface. Fith Edition. Elsevier, 2014.

Rauber, T.; Rünger, G.; Prallel Programming for Multicore and Cluster Systems. Second Edition. Springer. 2013.

Huber Joseph; Hernandez, Oscar, Lopez, Graham; Effective Vectorization with OpenMP 4.5. Report, Oak Ridge National Laboratory, 2017. ORNL/TM-2016/391

Arquiteturas Paralelas: máquinas SIMD

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente