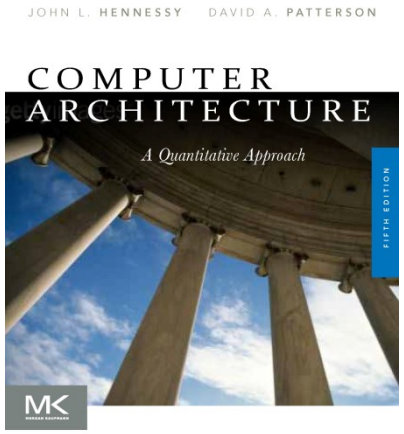


Paralelismo em nível de dados (DLP) em arquiteturas vetoriais, SIMD e GPU

Adaptado por
Marcos E. Barreto
DCC / UFBA

Introdução (1)

- Quanto paralelismo de dados está presente num dado conjunto de aplicações?
  - Arquiteturas SIMD podem explorar uma grande quantidade de DLP em aplicações científicas com dados matriciais e em aplicações multimídia (som e imagem)
- Arquiteturas SIMD são energeticamente mais eficientes do que arquiteturas MIMD
  - Busca uma instrução para operar sobre múltiplos dados
- Arquiteturas SIMD permitem que o programador continue pensando sequencialmente

Introdução (2)

- Classes de arquiteturas SIMD:

- Arquiteturas vetoriais
  - Execução pipeline de múltiplas operações sobre dados.
  - Muito caras (transistores e largura de banda de memória) para a época (década de 1980).

- SIMD (*Single Instruction, Multiple Data*)
  - Operações simultâneas sobre dados.
  - Suporte para multimídia (MMX, SSE e AVX) e operações de ponto flutuante.

- GPUs
  - Semelhante às arquiteturas vetoriais, mas como um ecossistema que provê maior desempenho se comparado aos processadores *multicore*.

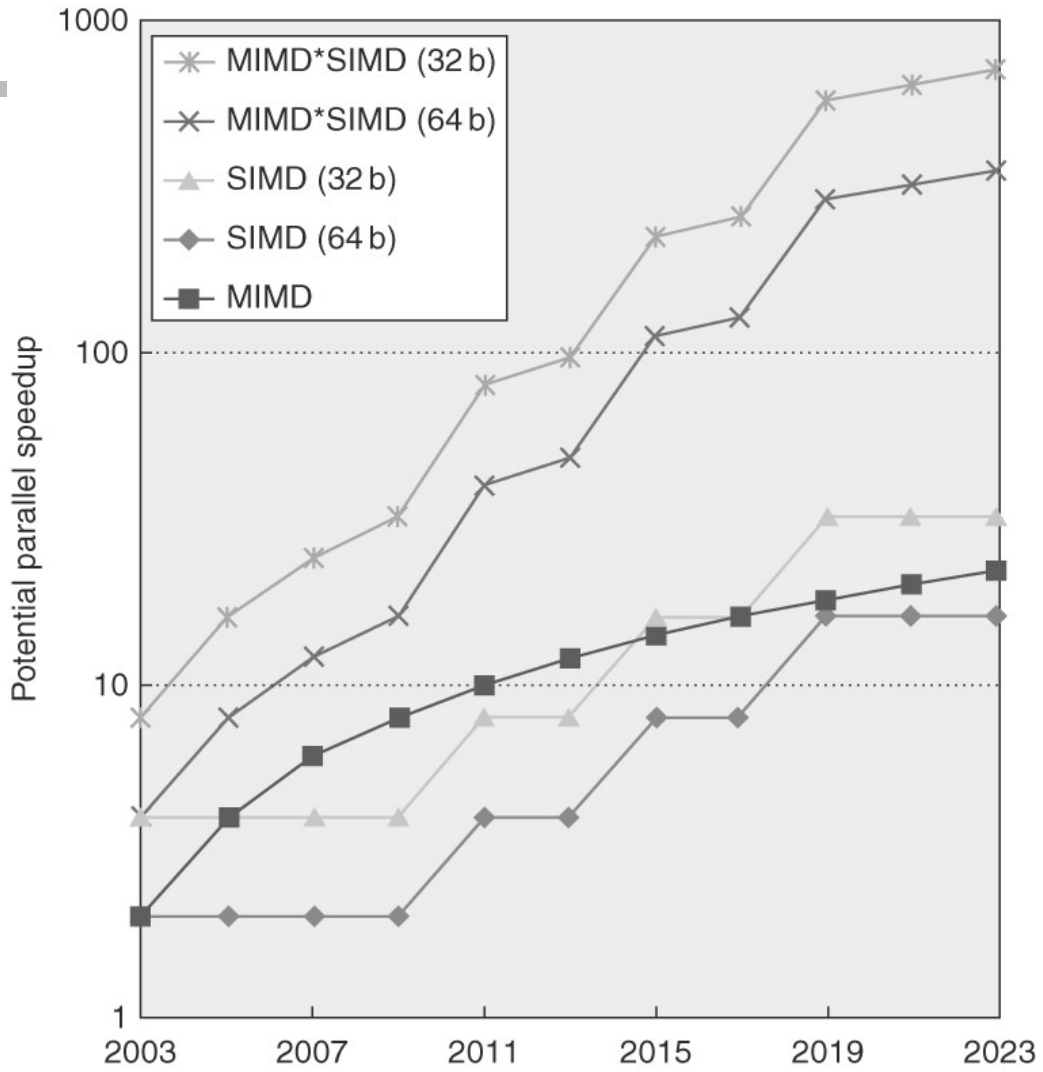
Introdução (3)

Para processadores x86:

- 2 cores por chip a cada 2 anos.

- SIMD vai dobrar sua capacidade a cada 4 anos.

- Próxima década deve observar arquiteturas SIMD com o dobro do *speedup* apresentado pelas arquiteturas MIMD.

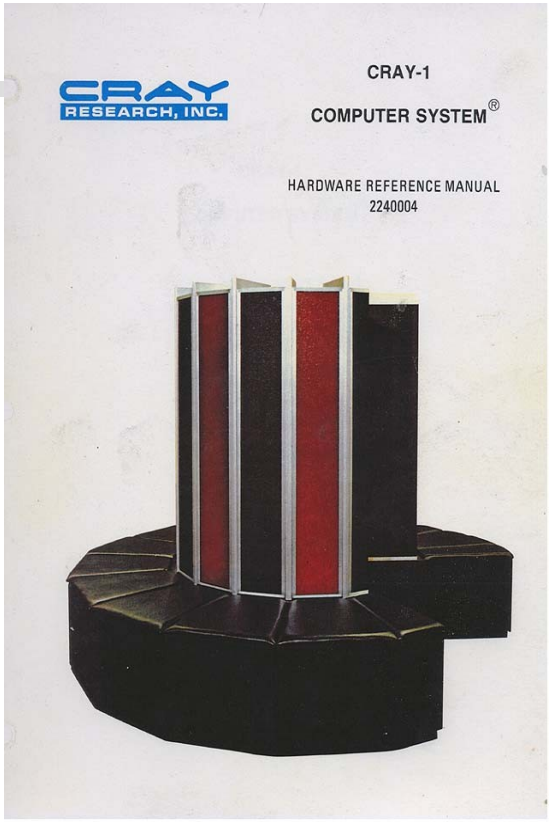


Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

Arquiteturas vetoriais

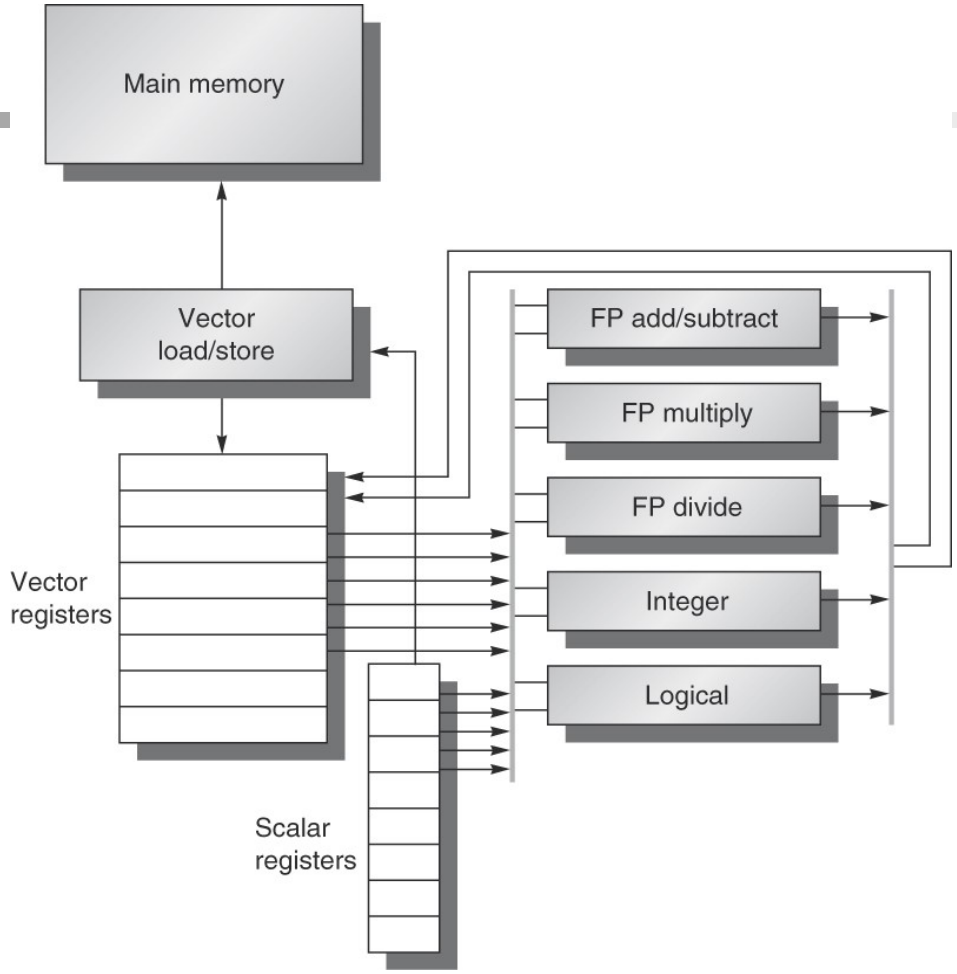
- Ideia básica:
  - Lê um conjunto de dados dispersos na memória para dentro de “registradores vetoriais”
  - Opera nos dados contidos nestes registradores
  - Escreve os resultados de volta na memória
  - Uma instrução opera sobre um vetor de dados, resultando em diversas operações registrador-registrador.
- Registradores são controlados pelo compilador
  - Escondem a latência de memória
  - Aumentam a largura de banda da memória

Exemplo: VMIPS



- Levemente baseado no Cray-1
- Registradores vetoriais
  - 8 registradores vetoriais, cada um com 64 elementos de 64 bits cada.
  - 16 portas de leitura e 8 portas de escrita
  - Unidades funcionais vetoriais
  - Pipelining total (nova instrução a cada ciclo)
  - Detecção de penalidades estruturais e de dados
  - Unidade de load/store vetorial
  - Pipelining total
  - Uma palavra de memória por ciclo após latência inicial
  - Registradores escalares
  - 32 de propósito geral e 32 de ponto flutuante

VMIPS



- **The basic structure of a vector architecture, VMIPS.** This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,R1	Move contents of R1 to vector-length register VL.
MFC1	R1,VLR	Move the contents of vector-length register VL to R1.
MVTM	VM,F0	Move contents of F0 to vector-mask register VM.
MVFM	F0,VM	Move contents of vector-mask register VM to F0.

Figure 4.3 The VMIPS vector instructions, showing only the double-precision floating-point operations. In

Exemplo de instruções VMIPS

- Exemplo: DAXPY (Double precision A x X plus Y)

MIPS

```
        L.D     F0,a        ; lê escalar a para F0
        DADDIU  R4,Rx,#512  ; último endereço para leitura
Loop:   L.D     F2,0(Rx)    ; lê X[i]
        MUL.D   F2,F2,F0    ; a * X[i]
        L.D     F4,0(Ry)    ; lê Y[i]
        ADD.D   F4,F4,F2    ; a * X[i] + Y[i]
        S.D     F4,9(Ry)    ; armazena em Y[i]
        DADDIU  Rx,Rx,#8    ; incrementa índice de X
        DADDIU  Ry,Ry,#8    ; incrementa índice de Y
        DSUBU   R20,R4,Rx   ; calcula limite do vetor
        BNEZ    R20,Loop    ; volta no laço caso ainda tenha elementos
```

VMIPS

```
L.D      F0,a       ; lê escalar a para F0
LV       V1,Rx      ; lê vetor X para V1
MULVS.D  V2,V1,F0   ; multiplicação vetor-escalar
LV       V3,Ry      ; lê vetor Y para V3
ADDVV    V4,V2,V3   ; soma vetores X e Y
SV       V4,Ry      ; armazena o resultado do vetor em Y
```

Dependências

```
        L.D     F0,a        ; lê escalar a para F0
        DADDIU  R4,Rx,#512  ; último endereço para leitura
Loop:   L.D     F2,0(Rx)    ; lê X[i]
        MUL.D   F2,F2,F0    ; a * X[i]
        L.D     F4,0(Ry)    ; lê Y[i]
        ADD.D   F4,F4,F2    ; a * X[i] + Y[i]
        S.D     F4,9(Ry)    ; armazena em Y[i]
        DADDIU  Rx,Rx,#8    ; incrementa índice de X
        DADDIU  Ry,Ry,#8    ; incrementa índice de Y
        DSUBU   R20,R4,Rx   ; calcula limite do vetor
        BNEZ    R20,Loop    ; volta no laço caso ainda tenha elementos
```

MIPS

```
L.D      F0,a       ; lê escalar a para F0
LV       V1,Rx      ; lê vetor X para V1
MULVS.D  V2,V1,F0   ; multiplicação vetor-escalar
LV       V3,Ry      ; lê vetor Y para V3
ADDVV    V4,V2,V3   ; soma vetores X e Y
SV       V4,Ry      ; armazena o resultado do vetor em Y
```

VMIPS

Cada instrução vetorial espera apenas pelo primeiro elemento do vetor (único *pipeline stall* para cada instrução vetorial).

Tempo de execução vetorial

- Tempo de execução depende de 3 fatores:
  - Tamanho dos vetores de operandos
  - Dependências estruturais
  - Dependências de dados
- Unidades funcionais do VMIPS consomem um elemento por ciclo de relógio
  - Tempo de execução é ≃ tamanho do vetor
- Convoy (*comboio, caravana*)
  - Conjunto de instruções vetoriais que potencialmente podem executar juntas, as quais não apresentam dependências estruturais.

Chimes

- Sequências com dependências RAW (*read-after-write*) podem estar no mesmo *convoy* via encadeamento (*chaining*).

- Encadeamento (*chaining*)
  - Permite que uma operação vetorial comece tão logo os elementos individuais (operandos vetoriais necessários) estejam disponíveis.

- *Chime*
  - Unidade de tempo para executar um convoy.
  - *m* blocos executam em *m* chimes
  - Para vetor de tamanho *n*, requer *m* x *n* ciclos de clock

Exemplo

Assumindo uma cópia de cada unidade funcional vetorial, estime quantos *chimes* e quantos ciclos por FLOP esta sequência demanda.

```
I1. LV       V1,Rx      ; lê vetor X
I2. MULVS.D  V2,V1,F0   ; multiplicação vetor - escalar
I3. LV       V3,Ry      ; lê vetor Y
I4. ADDVV.D  V4,V2,V3   ; soma os dois vetores
I5. SV       V4, Ry     ; armazena o resultado
```

Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

3 chimes; 2 operações de ponto flutuante (FLOP) por resultado => 1,5 ciclo por FLOP

Para vetores com 64 elementos = 64 x 3 = 192 ciclos de relógio

Múltiplas linhas

- Elemento *n* do registrador vetorial *A* somente pode operar com o elemento *n* do registrador vetorial *B* *em operações aritméticas*
- *Múltiplas linhas permitem reduzir o número de ciclos necessários para executar um chime.*

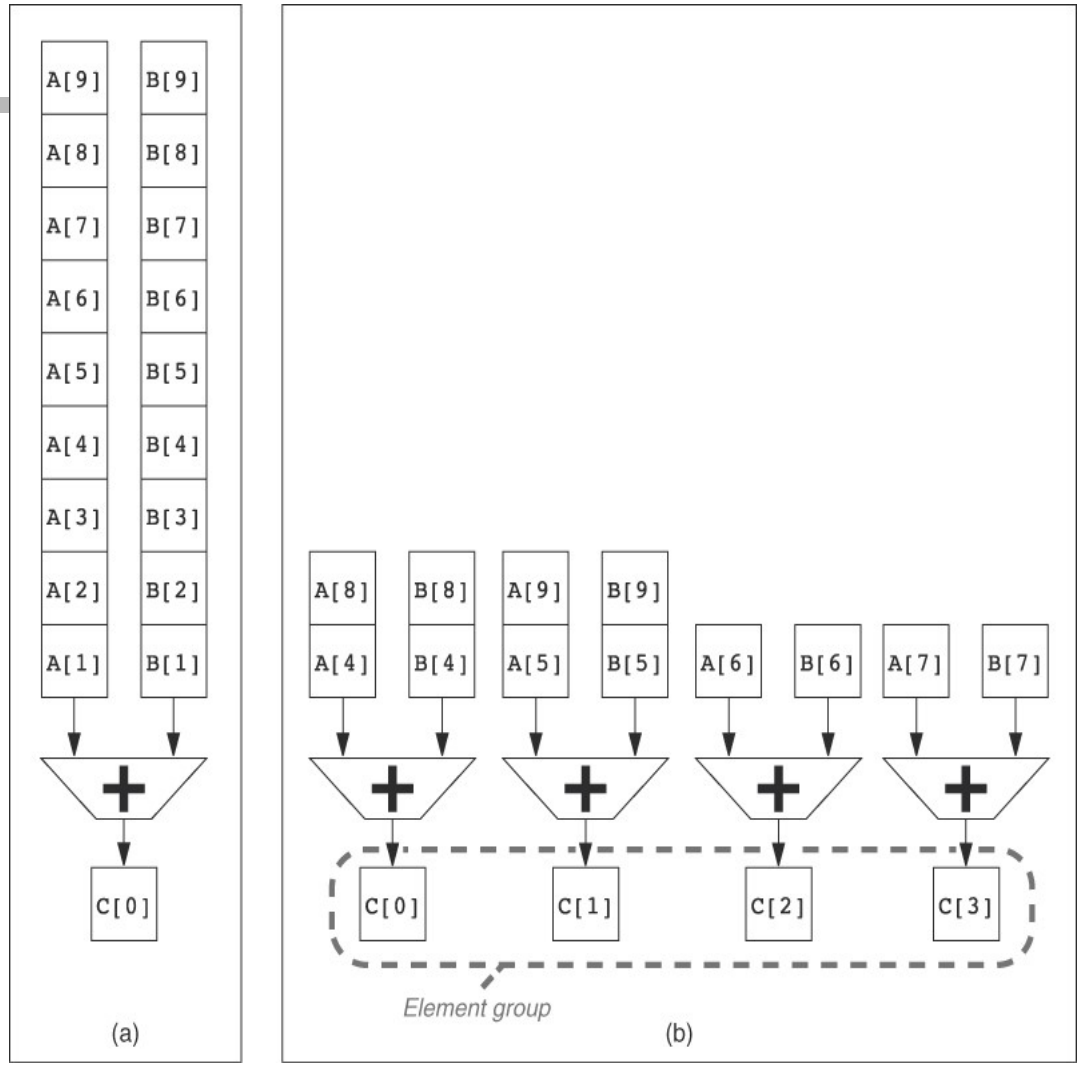
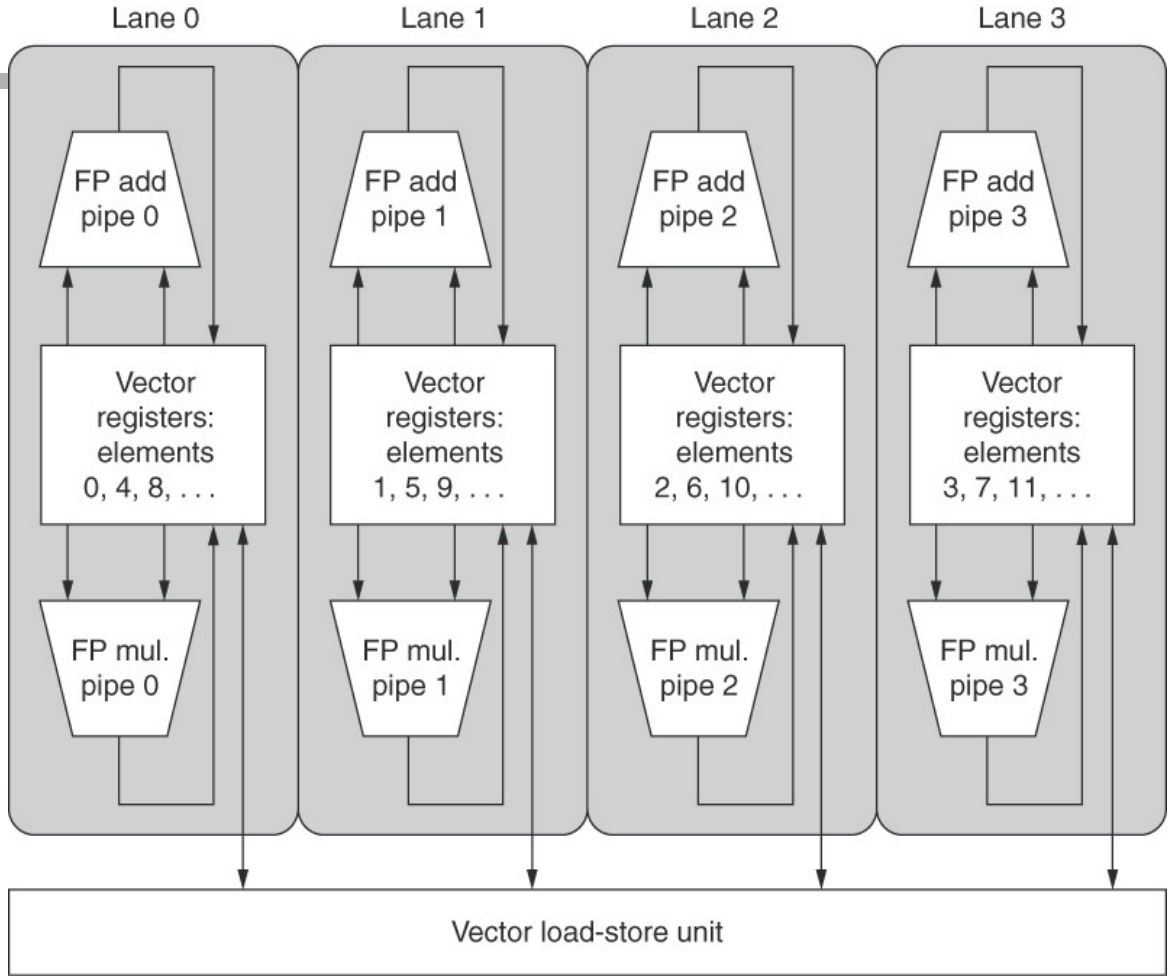


Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*.

Múltiplas linhas



- **Structure of a vector unit containing four lanes**. The vector register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which act in concert to complete a single vector instruction. Note how each section of the vector register file only needs to provide enough ports for pipelines local to its lane. This figure does not show the path to provide the scalar operand for vector-scalar instructions, but the scalar processor (or control processor) broadcasts a scalar value to all lanes.

Vector-length registers

- O tamanho dos vetores é naturalmente determinado pela quantidade de elementos no registrador vetorial
  - pode não corresponder à uma situação real

```
for (i=0; i<n; i++)
   Y[i] = a * X[i] + Y[i];
```

- Uso de vector-length register (VLR)
  - Controla o tamanho de cada operação com vetor
  - VLR <= MVL (maximum vector length)

Strip mining

- E se o tamanho do vetor (**n**) for maior que MVL?
- Strip mining
  - Geração de código tal que qualquer operação vetorial é feita com um tamanho menor ou igual a MVL.

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL);  j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL);  i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ; /*main operation*/
     low = low + VL; /*start of next vector*/
     VL = MVL; /*reset the length to maximum vector length*/
}
```

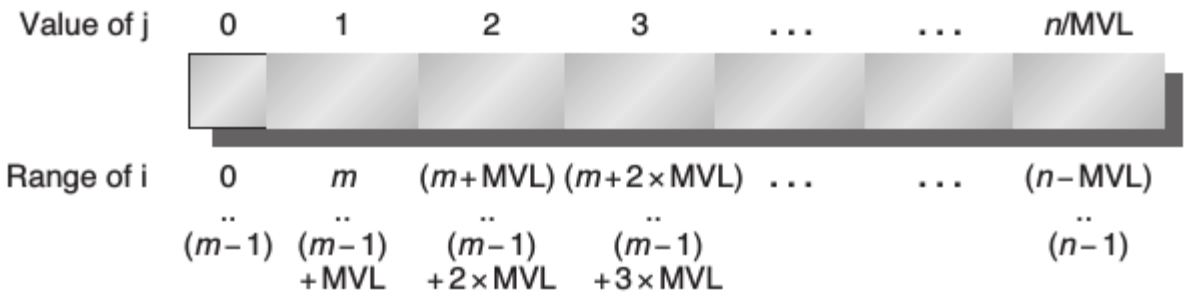


Figure 4.6 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, we use the variable *m* for the expression (n % MVL). (The C operator % is modulo.)

Programação de arq. vetoriais

- Relação entre compiladores e programadores.

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Extensões SIMD para multimídia

- Aplicações multimídia operam em dados menores do que o tamanho da palavra (geralmente, 32 bits) para o qual os processadores são otimizados.
  - Ex.: 8 bits para cada cor primária, 8 btis para transparência, 8 e 16 bits para representar sequência de áudio.

Limitações, comparando com instruções vetoriais:
- Número de operandos definidos no opcode
- Nenhum modo de endereçamento sofisticado (tais como *strided* e *scatter-gather*)
- Não emprega mascaramento de registradores

Implementações SIMD

- Intel MMX (1996)
- Registradores de ponto flutuante de 64 bits
- 8 operações de inteiros de 8 bits ou 4 operações de inteiros de 16 bits

- Streaming SIMD Extensions (SSE) (1999)
- 8 operações de inteiros de 16 bits
- 4 operações de inteiros ou ponto flutuante de 32 bits ou 2 operações de 64 bits

- Advanced Vector Extensions (AVX) (2010)
- 4 operações de inteiros ou ponto flutuante de 64 bits
- Inclui suporte para extensões de 512 e 1024 bits

- Operandos devem ser consecutivos e alinhados em memória

Exemplo de código SIMD

- Exemplo: DAXPY (Double precision A x X plus Y)

```
        L.D     F0,a        ; lê escalar a para F0
        MOV     F1,F0       ; copia a para F1 para SIMD MUL
        MOV     F2,F0       ; copia a para F2 para SIMD MUL
        MOV     F3,F0       ; copia a para F3 para SIMD MUL
        DADDIU  R4,Rx,#512  ; último endereço para leitura
Loop:   L.4D    F4,0(Rx)    ; lê X[i],X[i+1],X[i+2],X[i+3]
        MUL.4D  F4,F4,F0    ; a * X[i],a * X[i+1],a * X[i+2],a * X[i+3]
        L.4D    F8,0(Ry)    ; lê Y[i],Y[i+1],Y[i+2],Y[i+3]
        ADD.4D  F8,F8,F4    ; a * X[i] + Y[i], … … , a * X[i+3] + Y[i+3]
        S.4D    F8,0(Ry)    ; armazena em Y[i],Y[i+1],Y[i+2],Y[i+3]
        DADDIU  Rx,Rx,#32   ; incrementa índice de X
        DADDIU  Ry,Ry,#32   ; incrementa índice de Y
        DSUBU   R20,R4,Rx   ; calcula limite do vetor
        BNEZ    R20,Loop    ; volta no laço caso ainda tenha elementos
```

Opcodes com “4D” operam em 4 operandos de dupla precisão ao mesmo tempo.

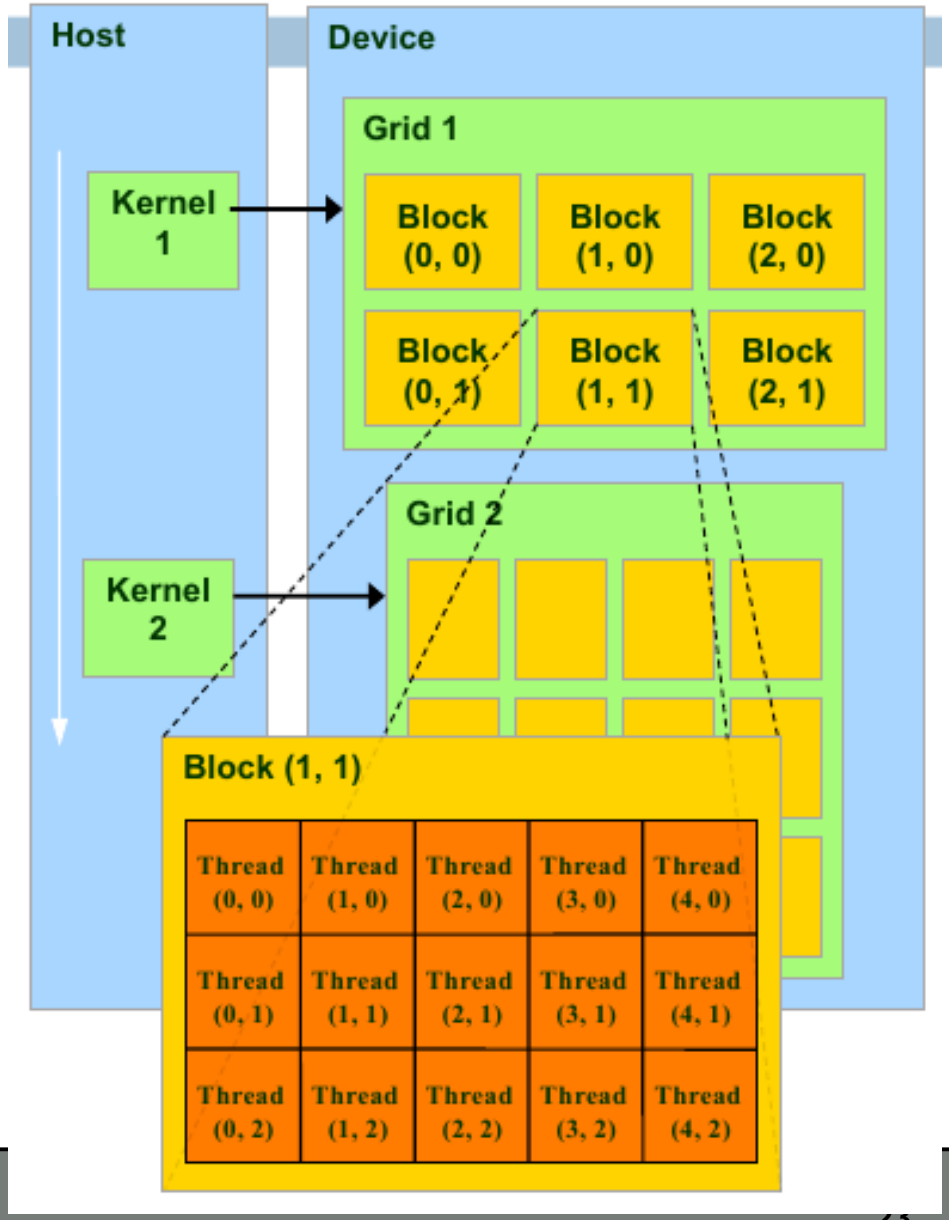
Graphical Processing Units

- Investimento no hardware para processamento gráfico eficiente e rápido.
- Emprego desse hardware para programação de propósito geral (GPGPU), através de linguagens e bibliotecas de programação

- Ideia básica:
  - Modelo de execução heterogêneo
  - CPU é o *host*, GPU é o *device*
  - Emprego de uma linguagem baseada em C
  - Modelo de programação é “Single Instruction Multiple Thread”

Threads e Blocos

- Um **thread** é associado a cada elemento de dado
- Threads são organizados em **blocos**
- Blocos são organizados num **grid**



Exemplo

- Exemplo: DAXPY (Double precision A x X plus Y)

```
// Invocação da função daxpy
daxpy(n, 2.0, x, y);
// Função daxpy em C
void daxpy(int n, double a, double *x, double *y)
{
  for(int=0; i<n; ++i)
    y[i] = a * x[i] + y[i];
}
```

C

```
// invoca a função daxpy com 256 threads por thread block
__host__ int nblocks = (n+255) / 256;
daxpy<<<nblocks,256>>>(n,2.0, x,y);

// Função daxpy em CUDA
__device__ void daxpy(int n, double a, double *x, double *y)
{
  int i = blockIdx.x * blockDim.x + threadId.x;
  if (i < n) y[i] = a * x[i] + y[i];
}
```

CUDA

Arquitetura GPU NVIDIA

- Semelhanças com arquiteturas vetoriais:
  - Apropriada para paralelismo de dados
  - Transferências scatter-gather
  - Regitradores de mascaramento
  - Grande quantia de registradores

- Differenças:
  - Ausência de processador escalar
  - Uso de multithreading para esconder latência de memória
  - Muitas unidades funcionais, em oposição ao pipeline de poucos estágios das arquiteturas vetoriais

Terminologia

- *Threads de instruções SIMD*
  - Cada thread tem seu contador de programa (PC)
  - Escalonador de threads baseado em *scoreboard*
  - Sem dependência de dados entre threads!
  - Escalonador garante até 48 threads
  - Esconde latência de memória
- Escalonador de blocos de thread
- Dentro de cada processador SIMD:
  - 32 threads

Terminologia

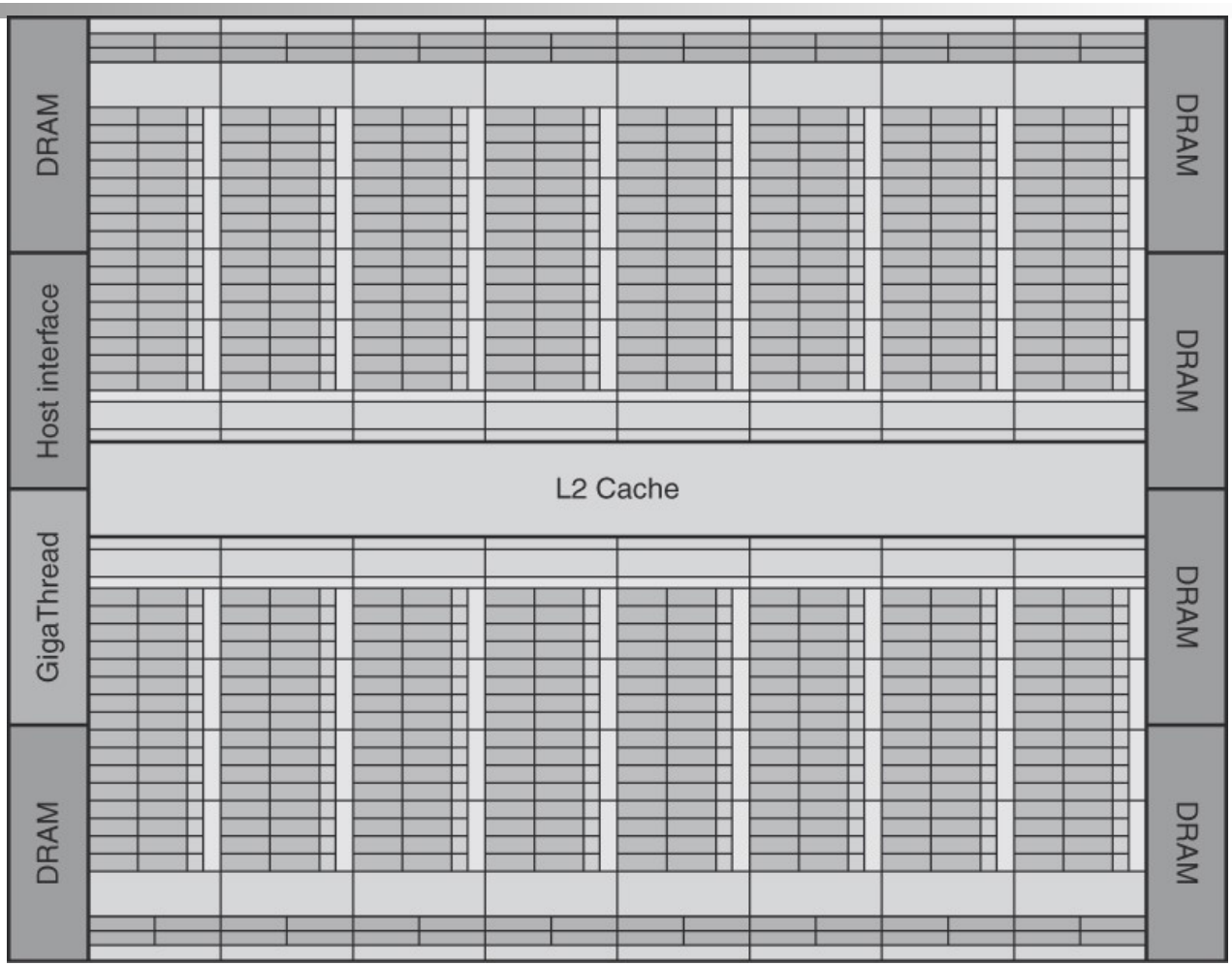


Figure 4.15 Floor plan of the Fermi GTX 480 GPU. This diagram shows 16 multithreaded SIMD Processors. The Thread Block Scheduler is highlighted on the left. The GTX 480 has 6 GDDR5 ports, each 64 bits wide, supporting up to 6 GB of capacity. The Host Interface is PCI Express 2.0 x 16. Giga Thread is the name of the scheduler that distributes thread blocks to Multiprocessors, each of which has its own SIMD Thread Scheduler.

Exemplo

- NVIDIA GPU tem 32,768 registradores
  - Divididos entre as linhas de processadores
  - Cada thread SIMD é limitada a 64 registradores
  - Cada thread SIMD tem até:
  - 64 registradores vetoriais de 32 elementos de 32 bits
  - 32 registradores vetoriais de 32 elementos de 64 bits
  - Fermi tem 16 linhas físicas SIMD, cada uma com 2048 registradores

Terminologia

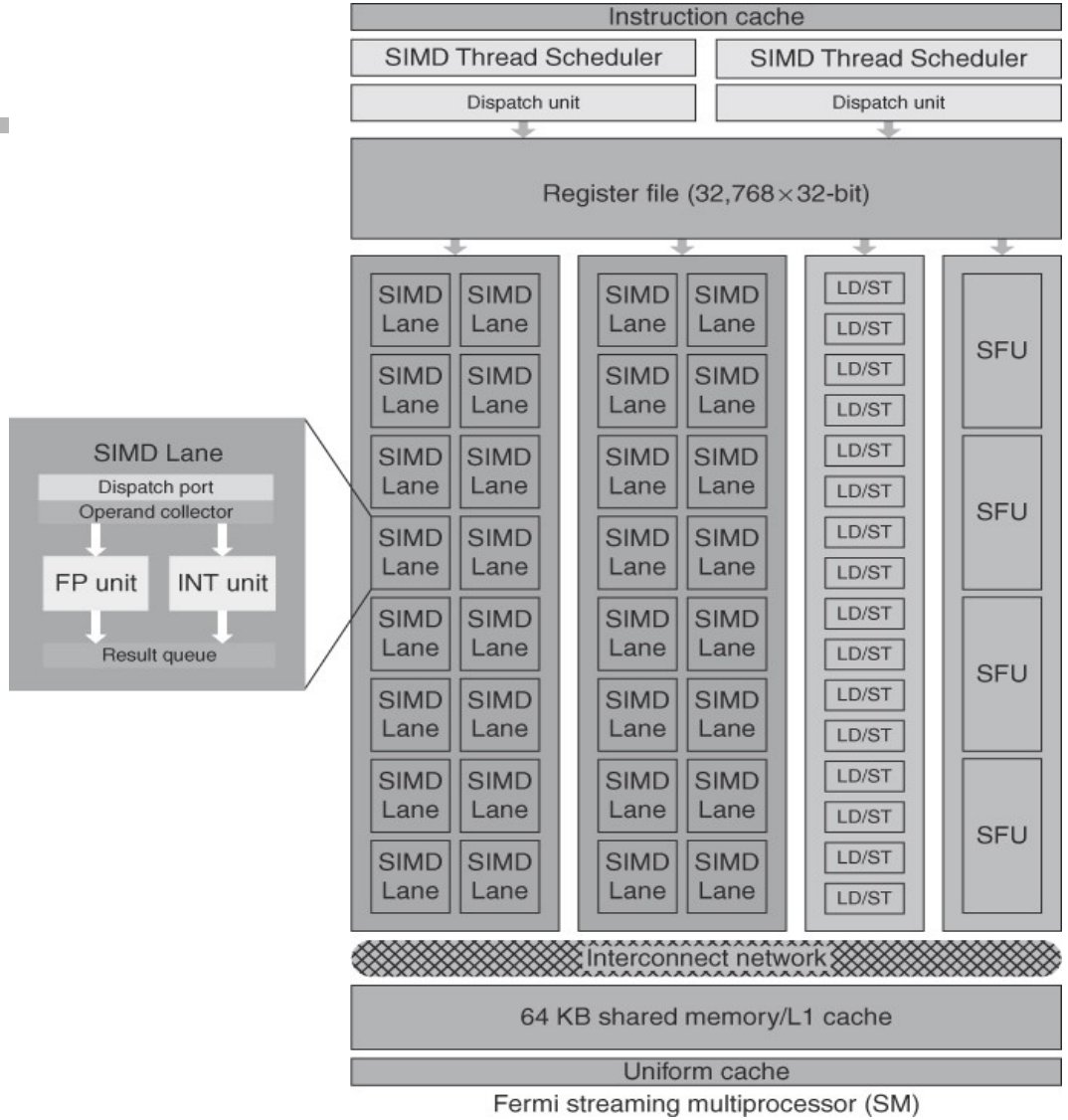


Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Fermi GPU. Each SIMD Lane has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The four Special Function units (SFUs) calculate functions such as square roots, reciprocals, sines, and cosines.

NVIDIA Instruction Set Arch.

- ISA é uma abstração do conjunto de instruções do hardware
  - “Parallel Thread Execution (PTX)”
  - Formato: `opcode.type d, a, b, c`
    - d = operando destino
    - a, b, c = operandos-fonte
    - type pode ser untyped bits, unsigned integer, signed integer ou floating point

NVIDIA Instruction Set Arch.

- Exemplo: DAXPY (Double precision A x X plus Y)

```
shl.s32 R8,blockIdx,9          ; ThreadBlockID * Block size
                                 (512 or 29)
add.s32 R8, R8, threadIdx      ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]      ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]      ; RD2 = Y[i]
mul.f64       R0D, RD0, RD4    ; RD0 = RD0 * RD4 (escalar a)
add.f64       R0D, RD0, RD2    ; RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0      ; Y[i] = sum (X[i]*a + Y[i])
```

Estruturas de memória

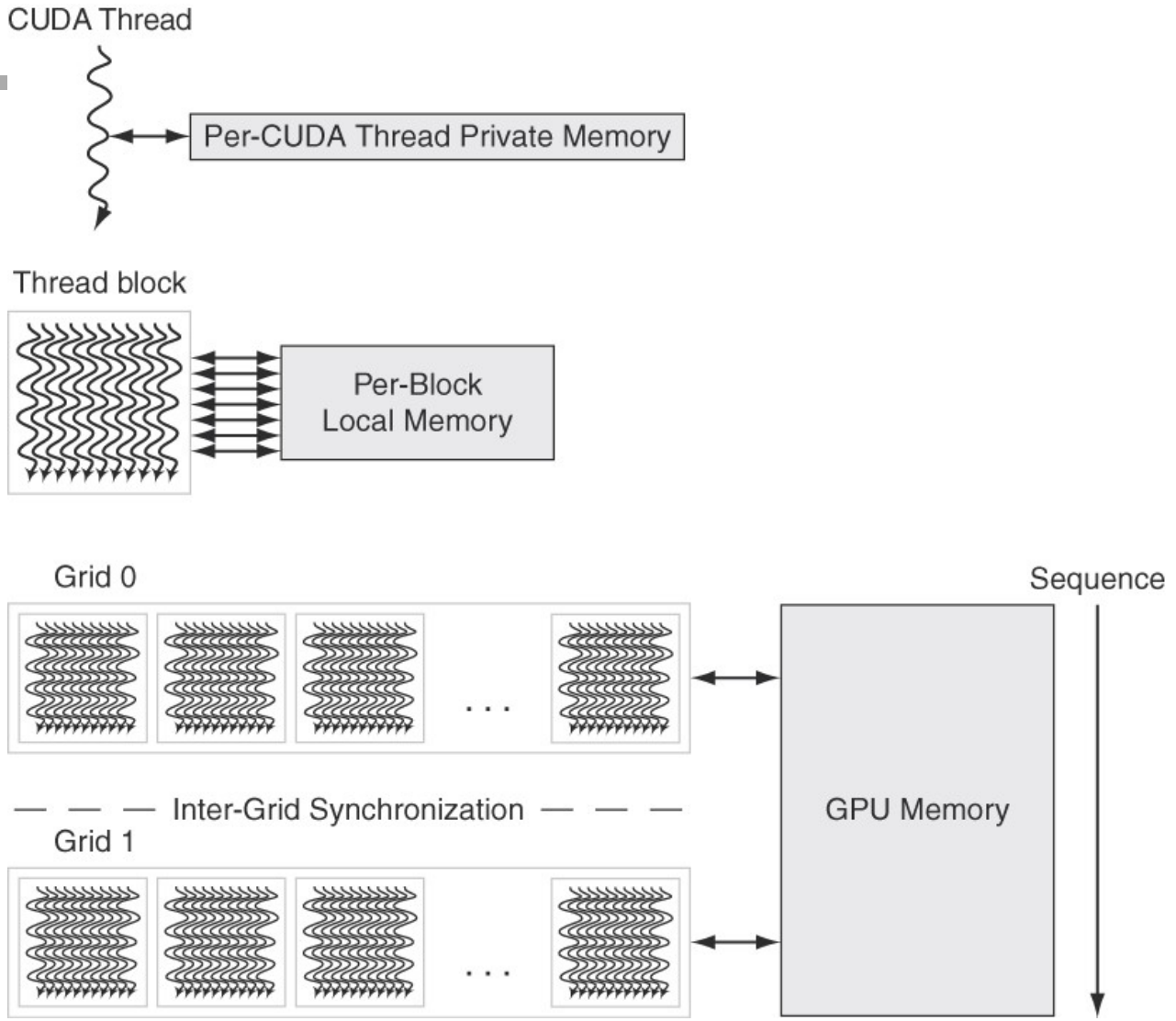


Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.