GPU : Graphics Processing Units

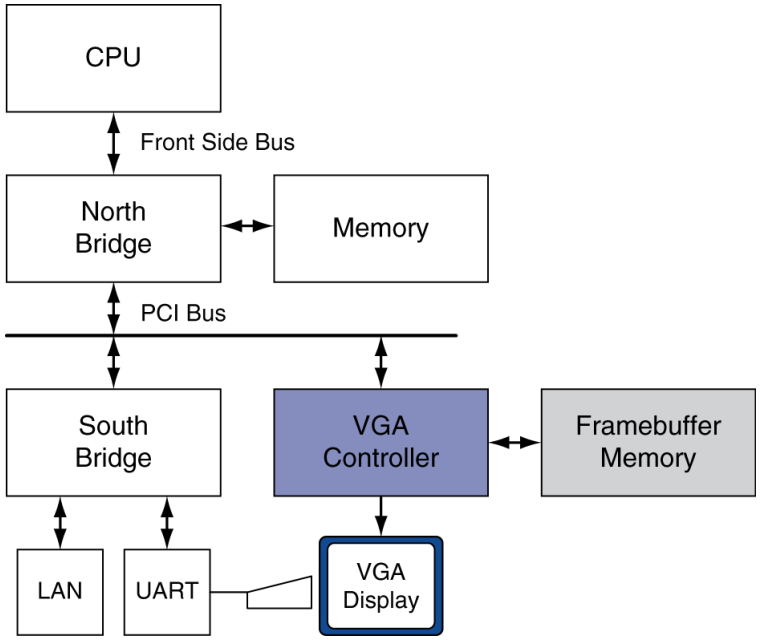
Arquitetura de Computadores

Mestrado Integrado em

Engenharia Informática

Controladores gráficos –80's

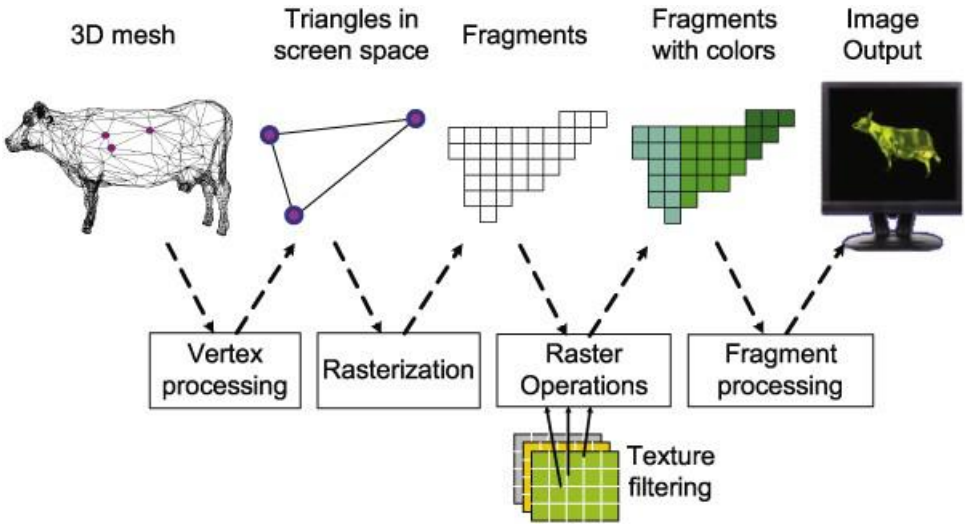
- Nos anos 80 era responsabilidade do CPU gerar a imagem gráfica e copiá-la para um espaço de endereçamento conhecido como *Framebuffer,* de onde era depois convertida em sinais analógicos e enviada para o *display*



Nome	Resolução	Cores
MDA (1981)	Caracteres (80x25)	B/W
CGA (1981)	640x200	16
EGA (1984)	640x350	16 de uma palette de 64
VGA (1987)	640x480	16

GPUs – 90's

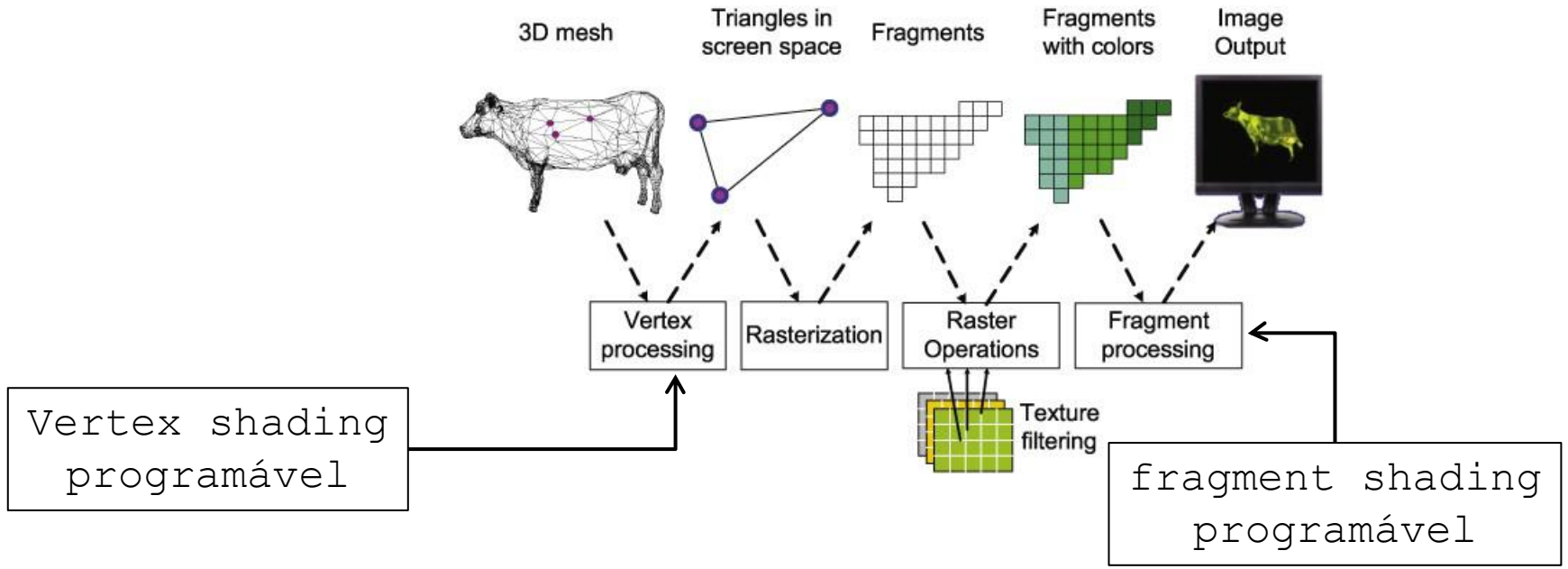
- Na década de 90 aparecem os GPUs, como co-processadores que aliviam o CPU dessa carga



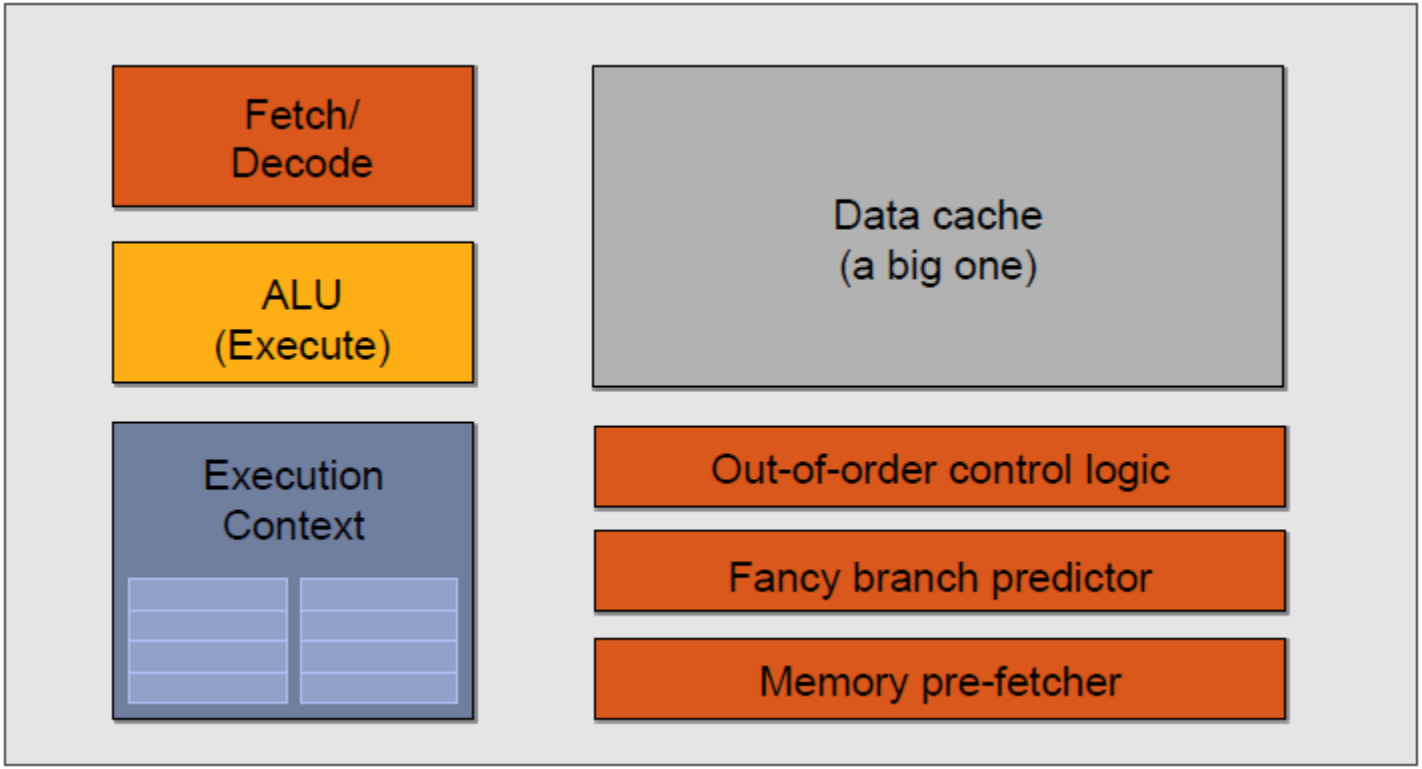
- Extremamente paralelos O(pixels)
- Apenas processavam gráficos
- Toda a lógica era replicada pra providenciar paralelismo, mas estas unidades não eram programáveis

GPUs – 2000 .. 2004

- O aumento do número de transístores (Lei de Moore) permitiu flexibilizar os GPUs
- Alguns componentes do *pipeline* passam a ser programáveis, dando origem ao que se chamou de *shader programming*

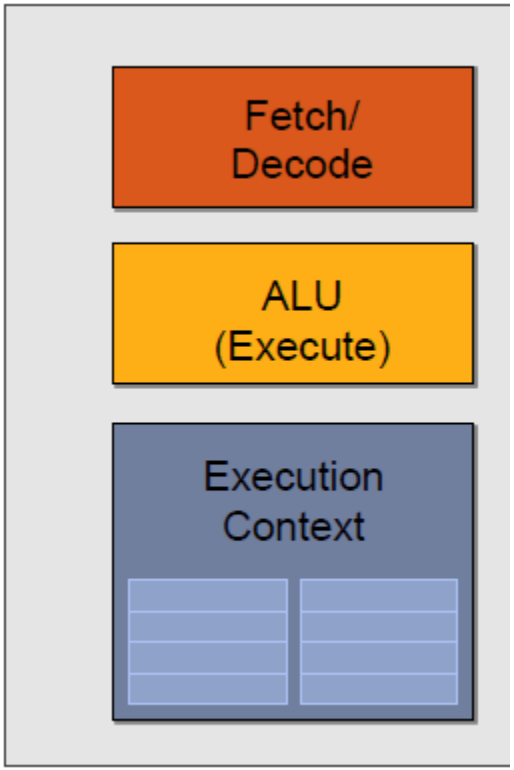


CPU-*style core*

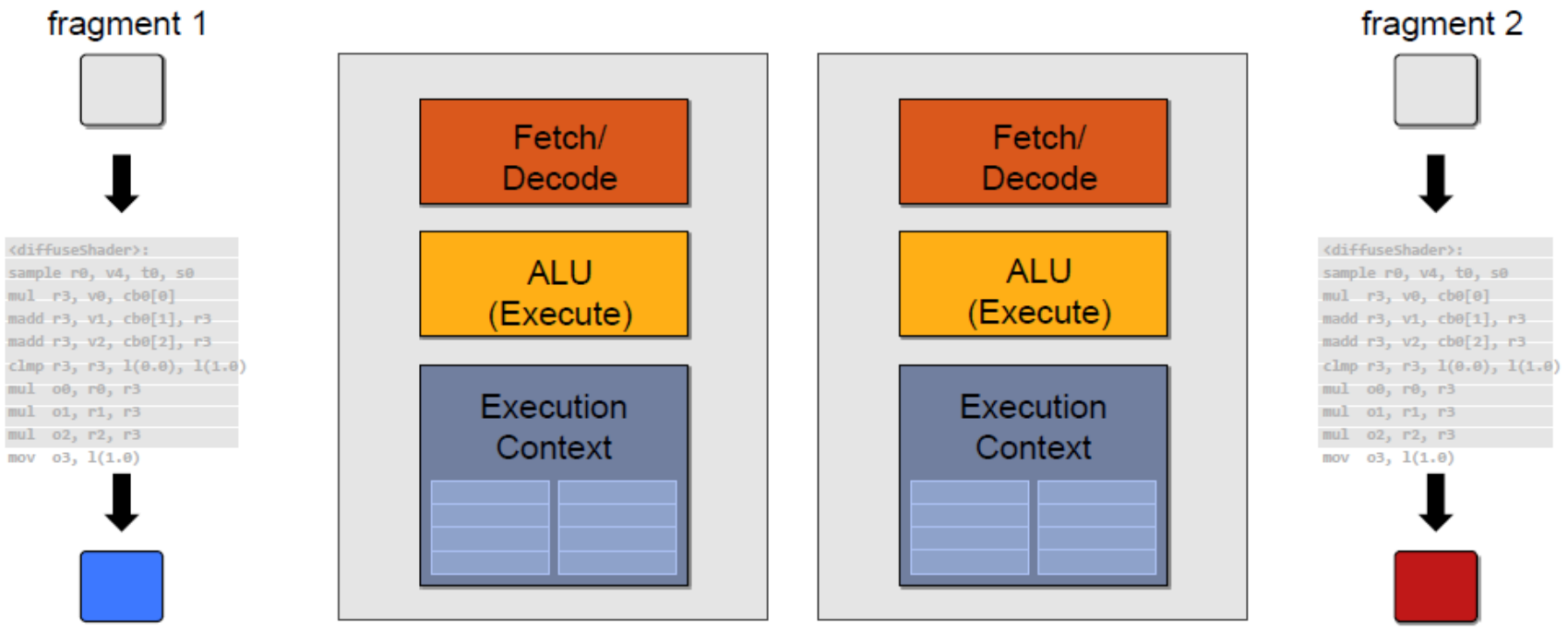


Core elementar

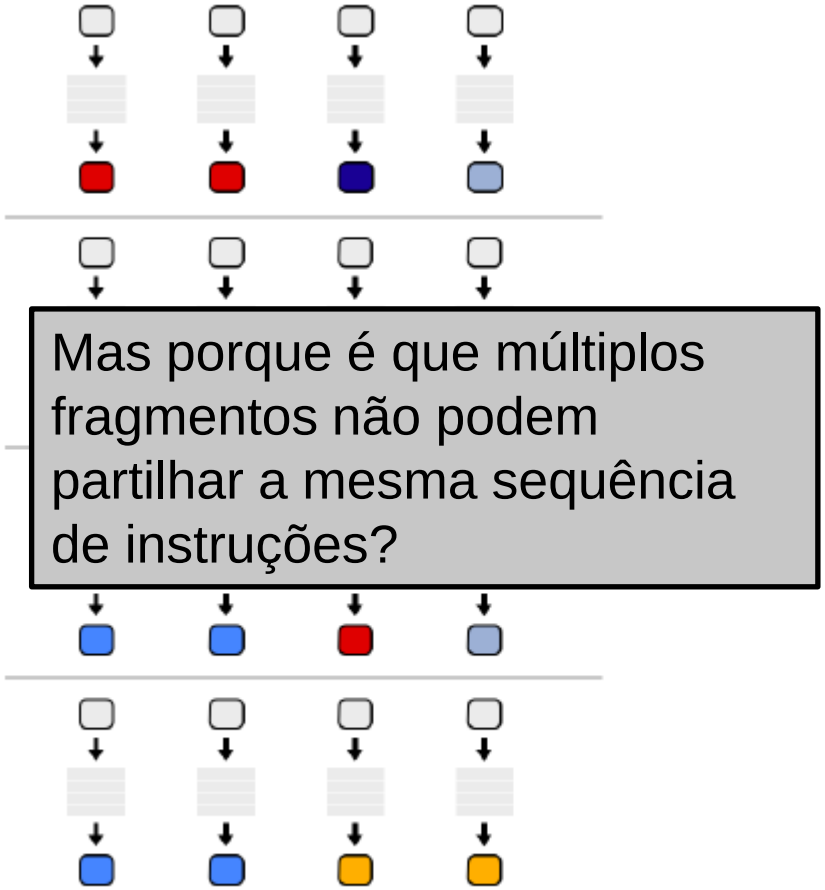
- Remover toda a complexidade de controlo que explora ILP



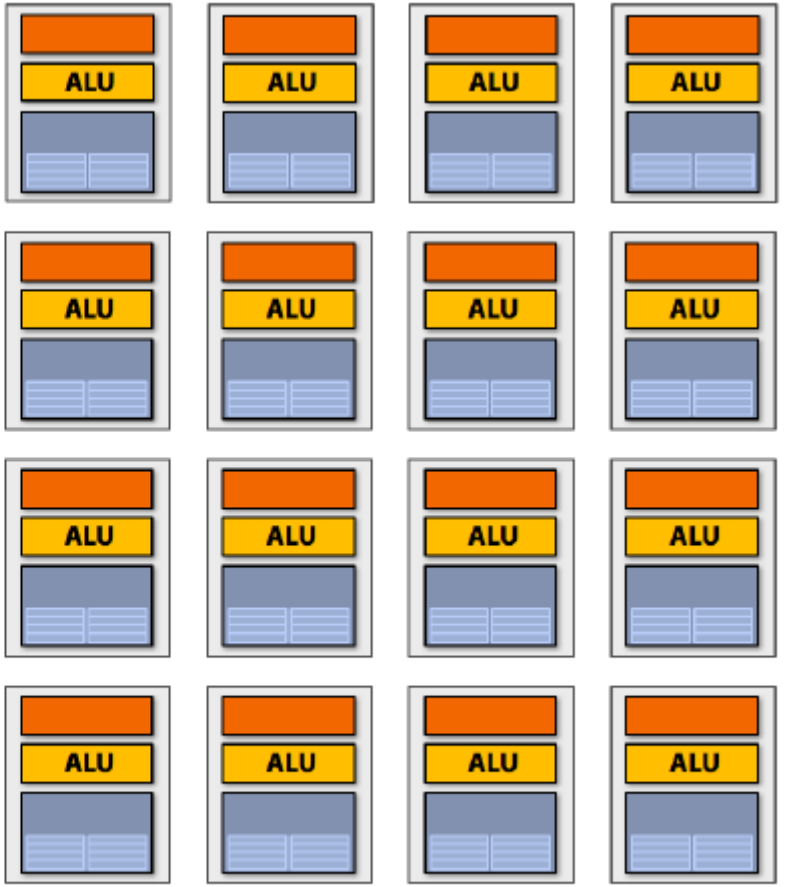
2 *cores* elementares



16 *cores* elementares

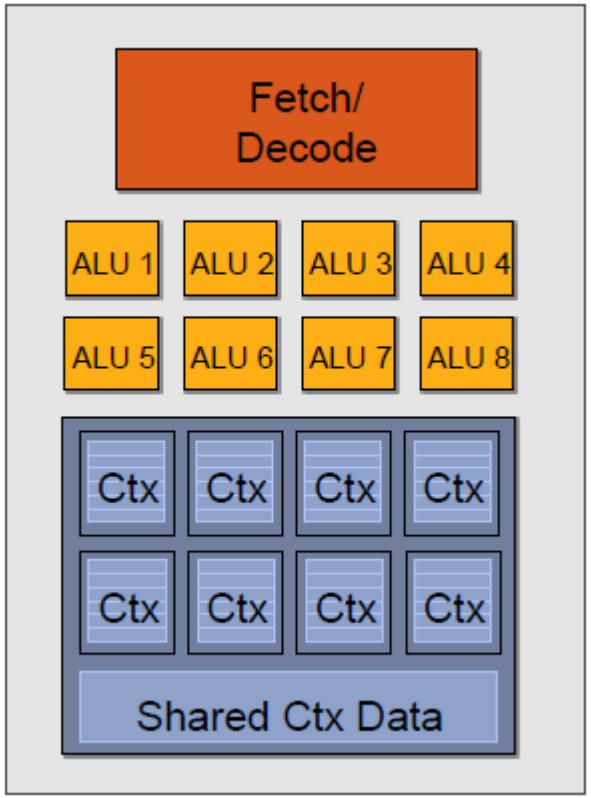


Mas porque é que múltiplos fragmentos não podem partilhar a mesma sequência de instruções?



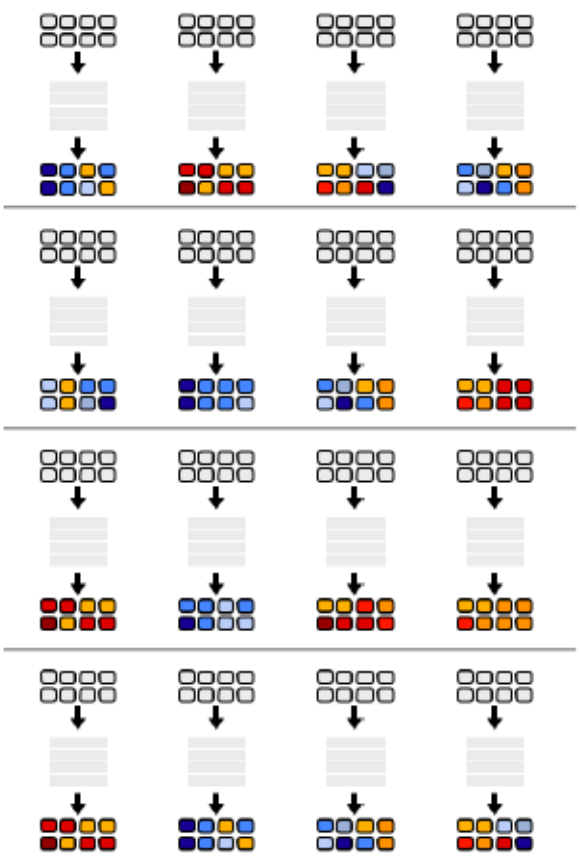
16 cores = 16 simultaneous instruction streams

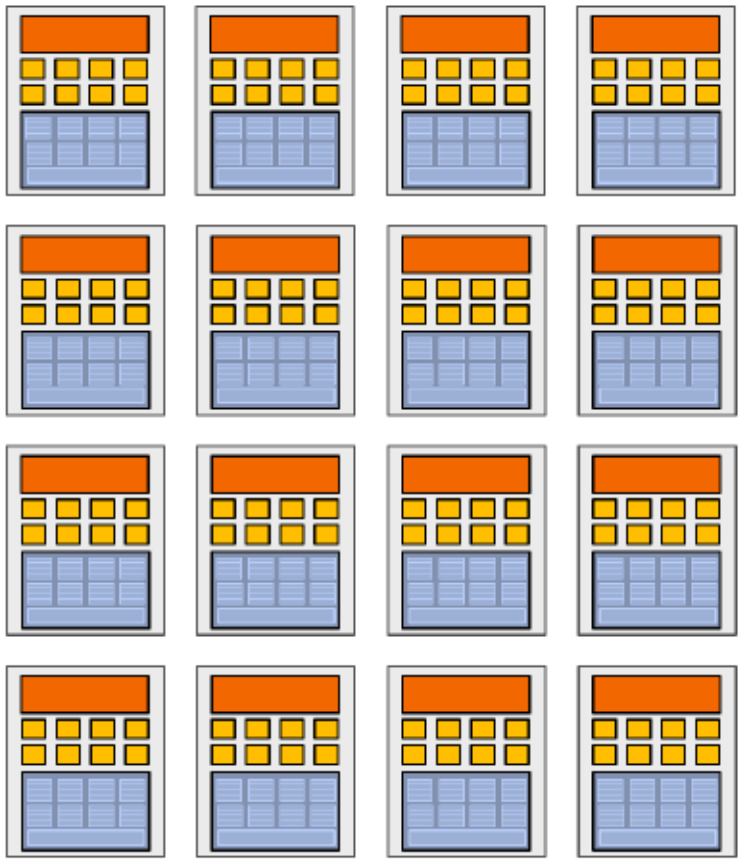
Streaming Multi Processor (SM)



- Amortizar a complexidade usando a mesma *stream* de instruções para as várias ALUs (unidades funcionais ou *stream processor* (SP))
- Nota: um único *Instruction Pointer* por SM
- SIMT:
  Single Instruction Multiple Threads

Múltiplos SM por GPU





- Neste exemplo, 16 SMs com 8 SPs = 128 *threads* a executar simultaneamente (*threads* do mesmo SM executam a MESMA instrução)

NVIDIA Kepler GK110

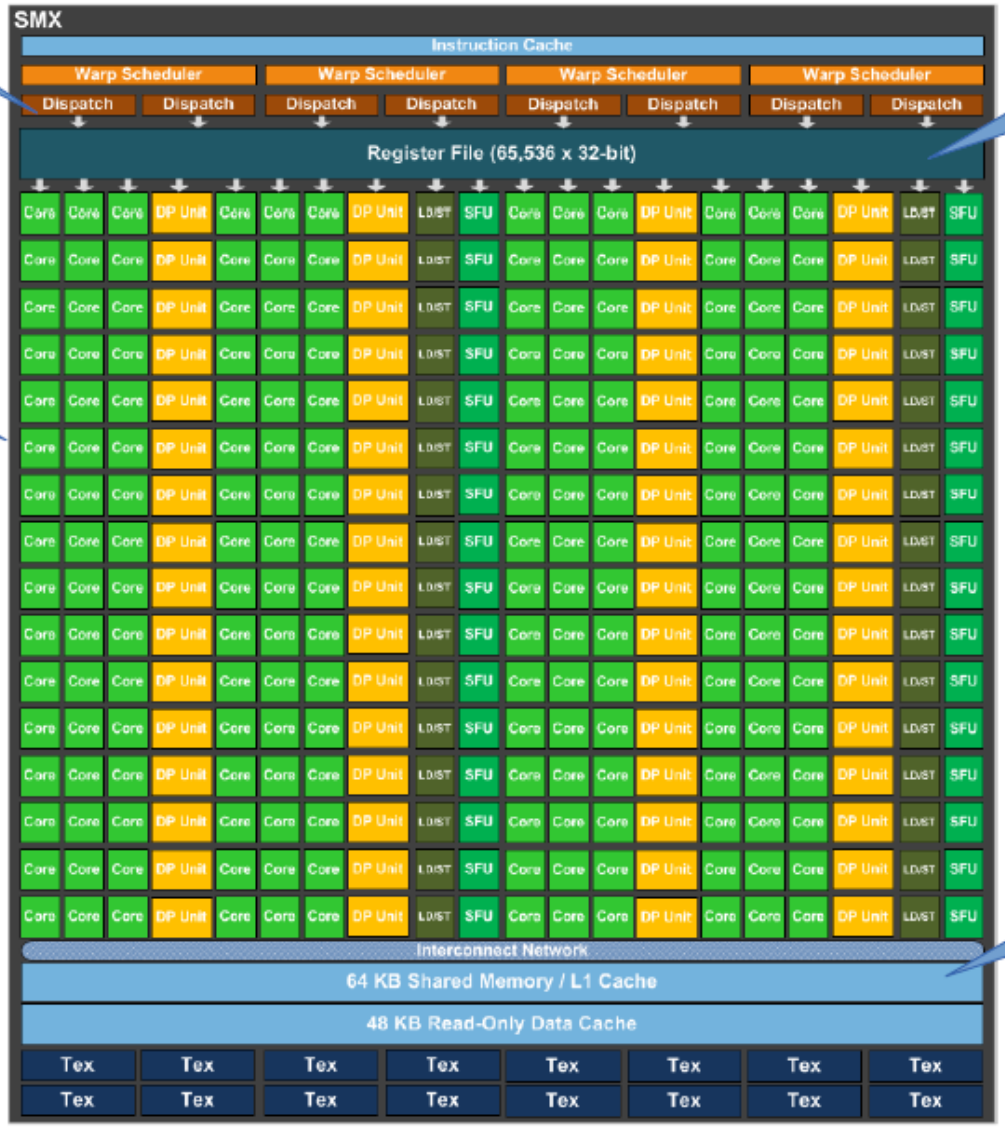
NVIDIA Kepler SMX

2-way
In-order

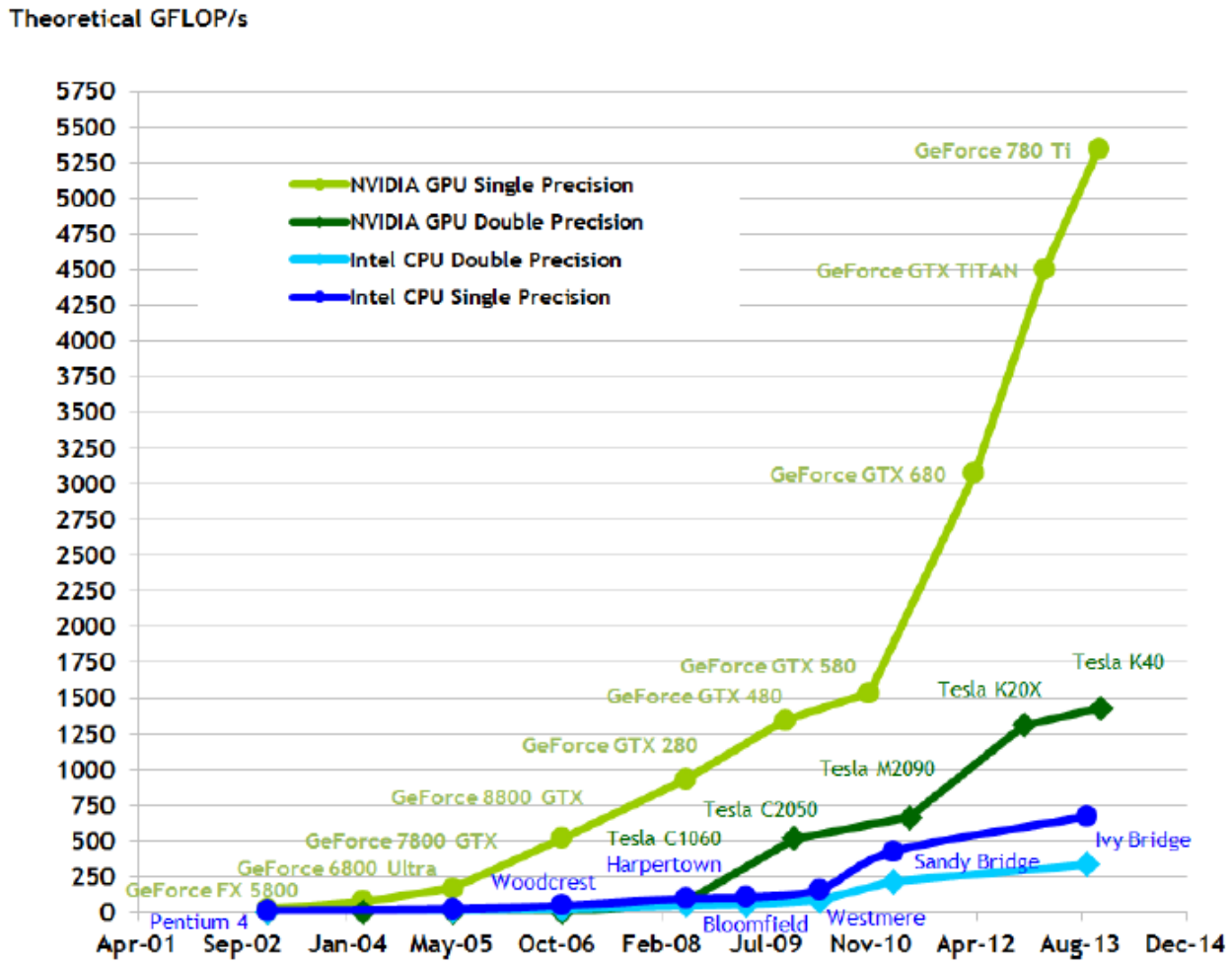
192 FP/Int
64 DP
32 LD/ST

256KB!

Partitioned
(user-defined)



Desempenho GPUs actuais

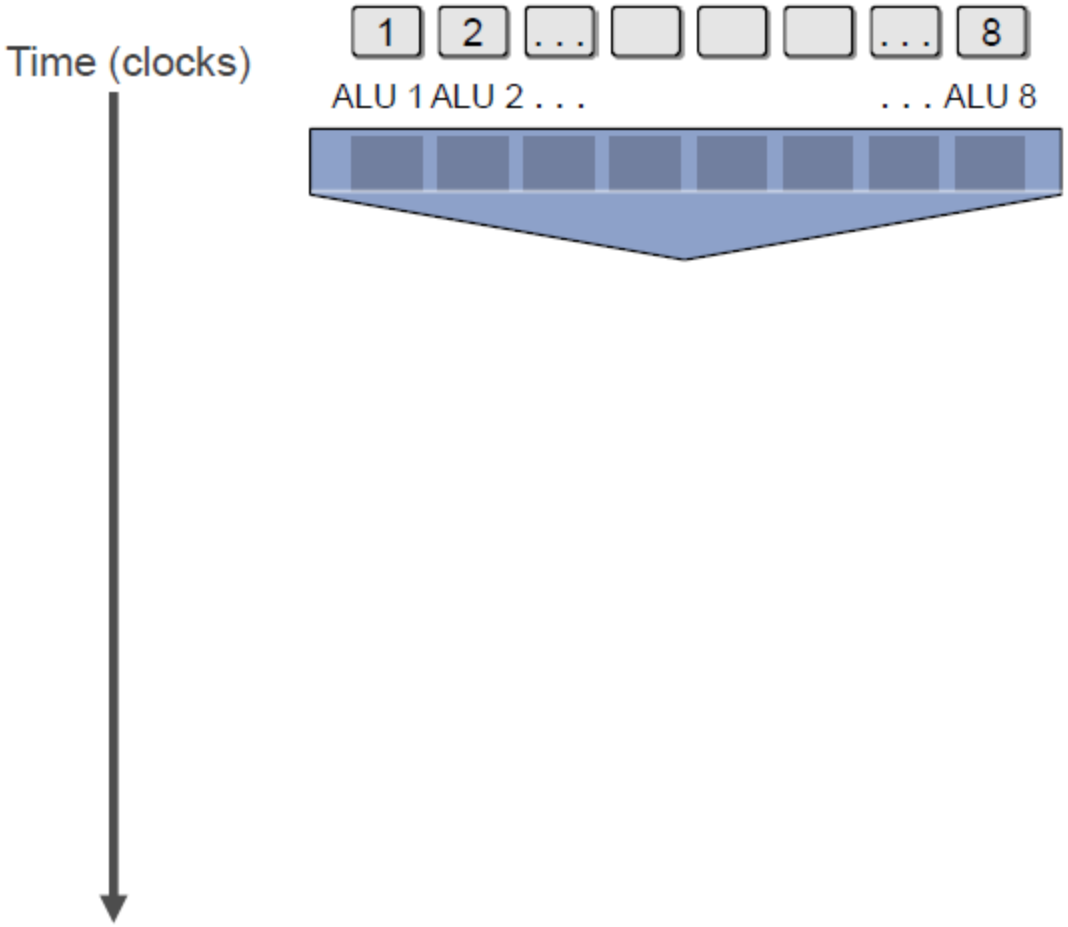


Estruturas Condicionais

- Todos os SPs de um mesmo SM executam a mesma instrução.
- Como lidar então com estruturas condicionais?

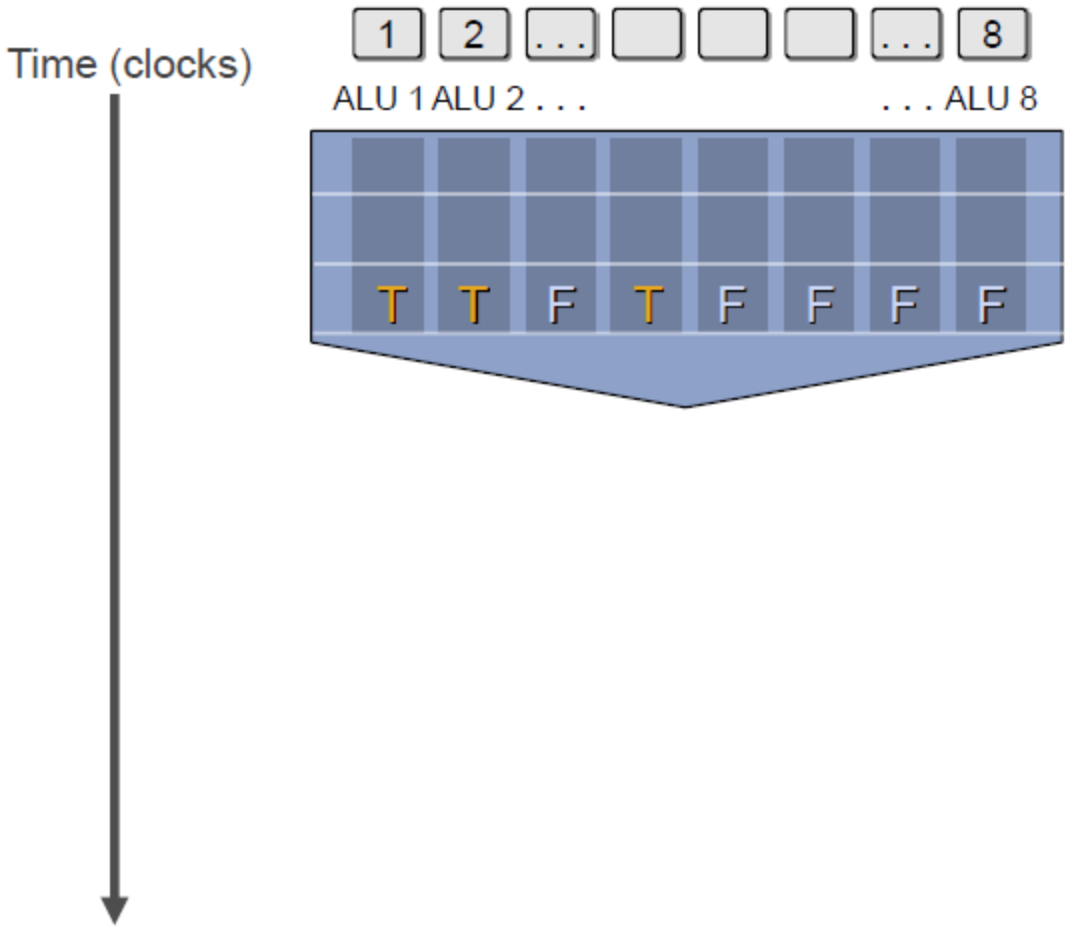
- Divergência do código:
  - Dividir as *threads*
  - Executar um dos caminhos da condição
  - Executar o outro caminho da condição
  - Combinar no fim
- Penalização do desempenho

Divergência



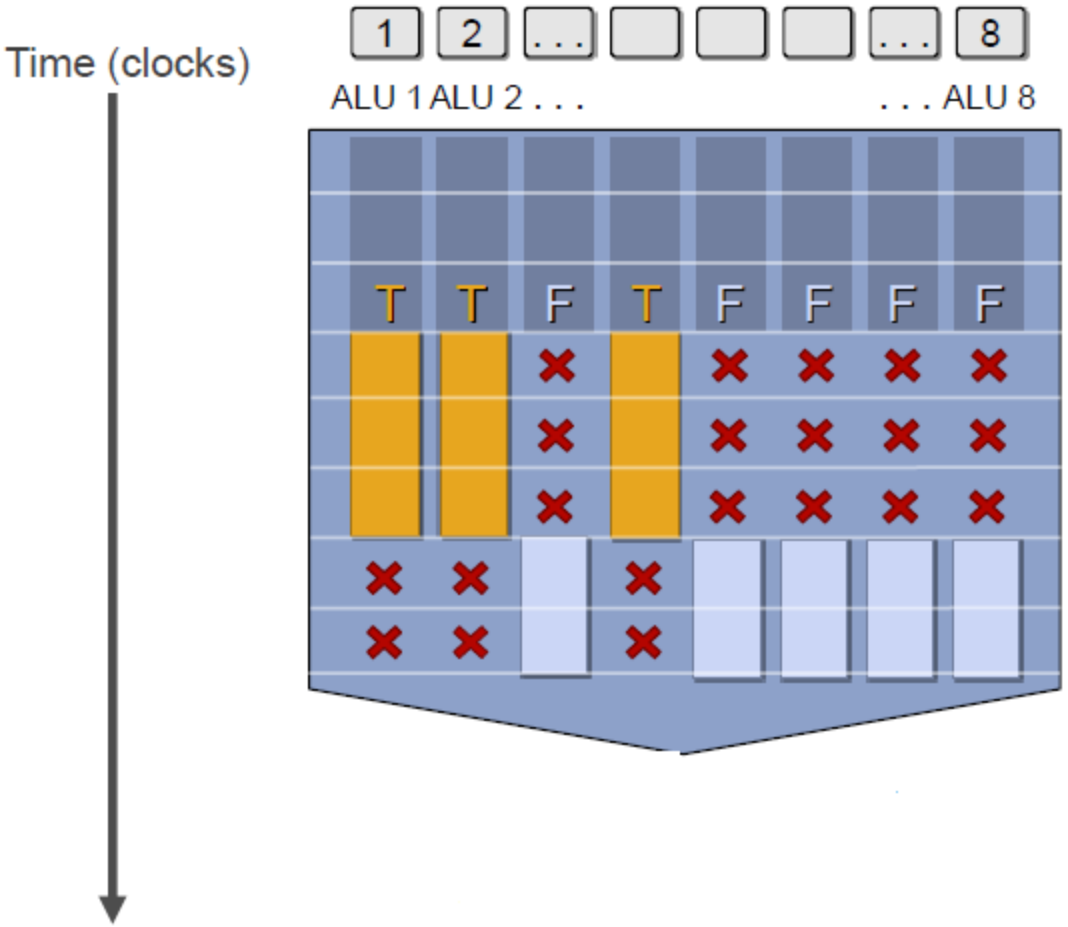

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Divergência



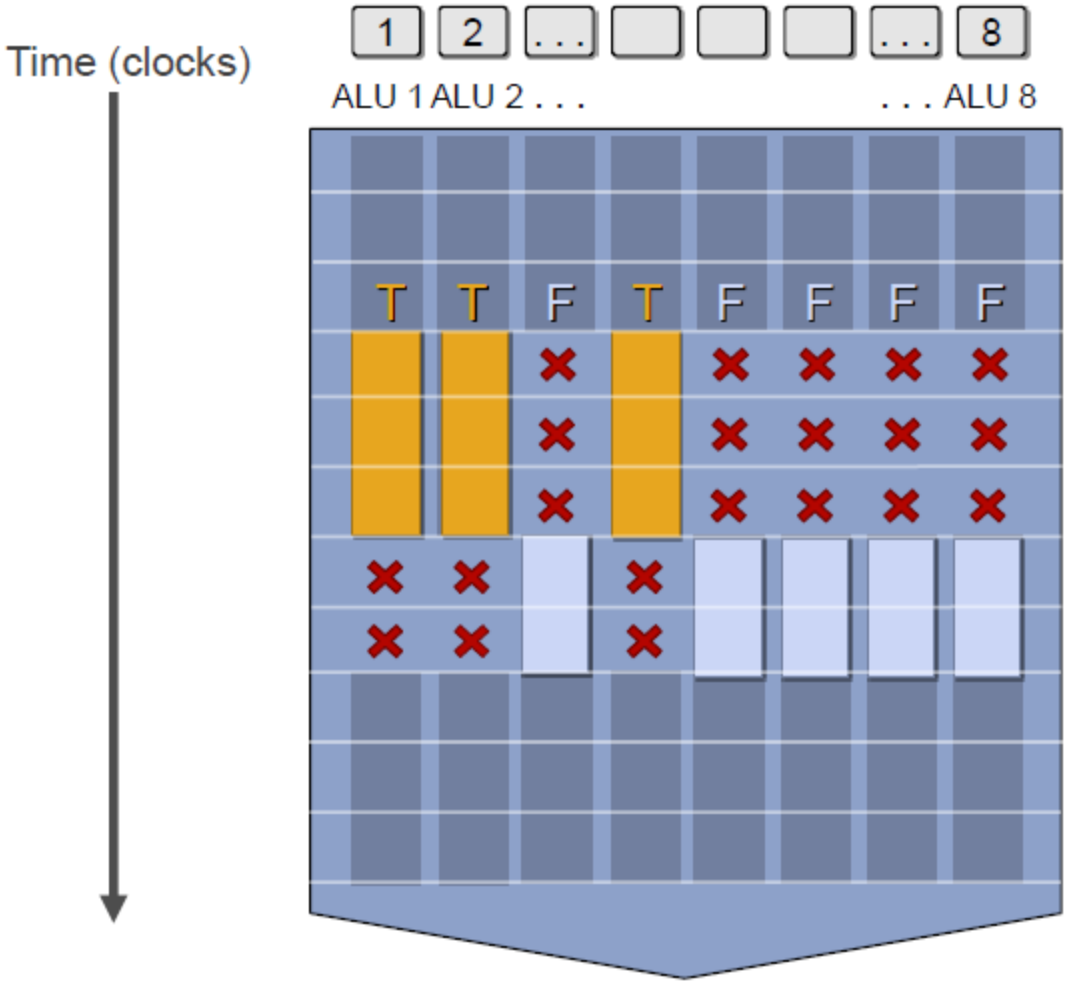

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Divergência




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Divergência



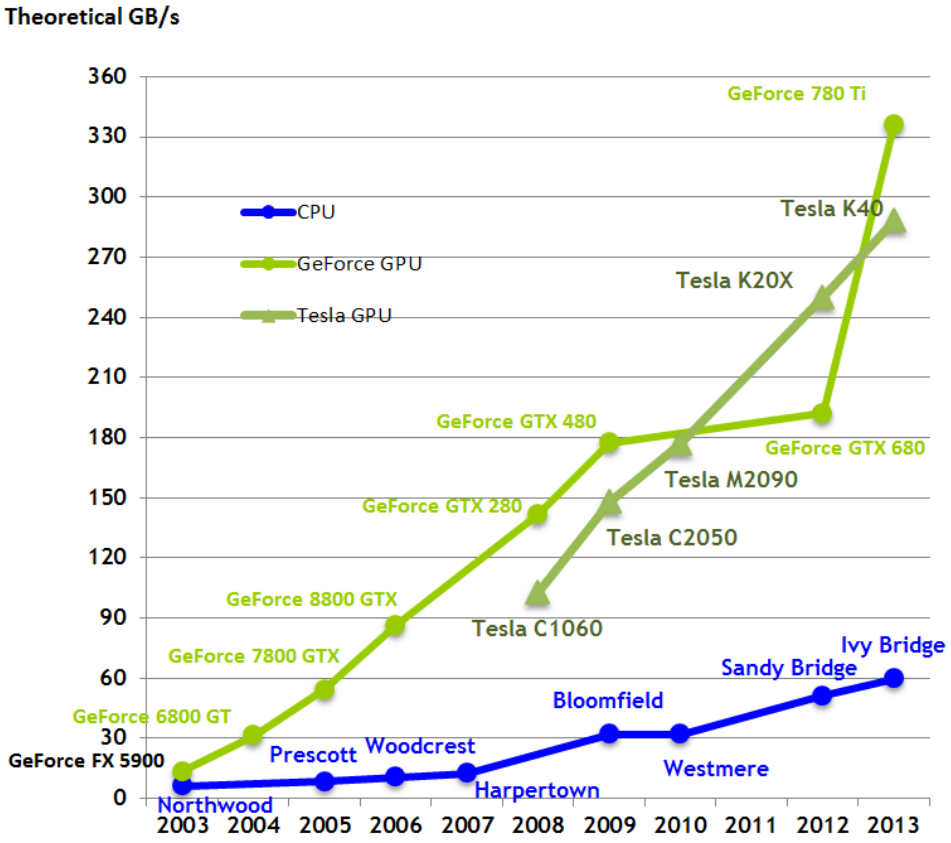

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Memória: latência e largura de banda

- Os GPUs modernos são massivamente paralelos, processando centenas ou milhares de elementos de dados em cada ciclo do relógio
- As ligações entre o GPU e a memória são muito largas, no sentido em que um elevado número de bits é transferido em cada ciclo
  Consequência: elevada largura de banda
- Uma largura de banda elevada resulta numa **latência elevada**

É fundamental esconder esta latência, mantendo as unidades de processamento ocupadas

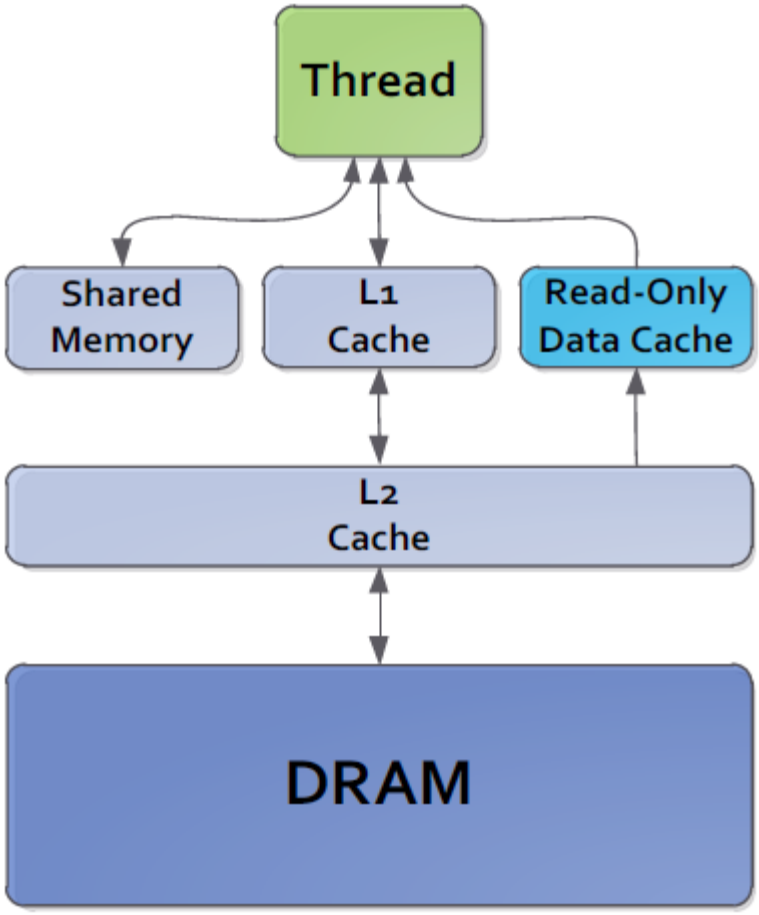
Memória: largura de banda



Hierarquia de memória e *working set*

- Os GPUs processam grandes quantidades de dados (na ordem dos MibiBytes) para gerar uma única imagem. Embora possam exibir boa localidade espacial, a localidade temporal é normalmente baixa. **Consequência**: baixa *hit-rate* (~90%) *versus* CPU (~99.9%)

- É necessário um mecanismo complementar para esconder a latência



Configurable Shared Memory and L1 Cache

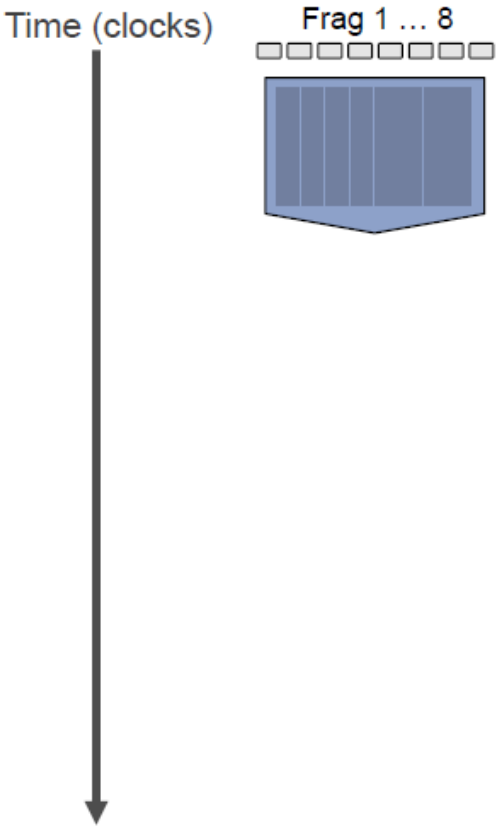
Interleaving threads

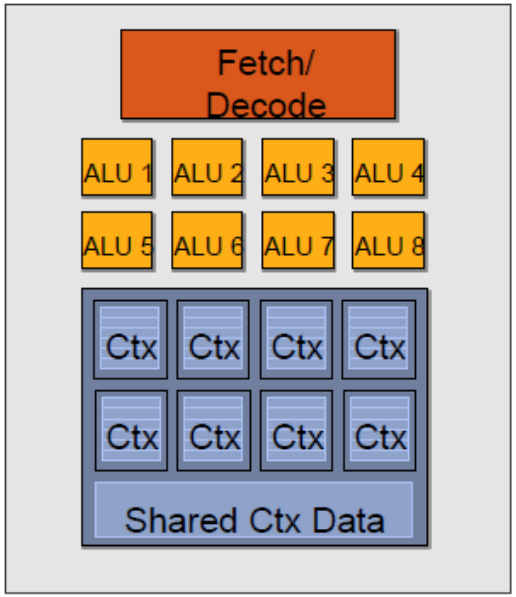
- Sempre que um conjunto de *threads* em execução num SM bloqueia à espera de resposta da memória, o SM muda de contexto e executa outro conjunto de *threads* cujos dados estejam disponíveis -> *thread interleaving*

- Este mecanismo requer que:
  - A mudança de contexto seja extremamente rápida
  - Existam muitas mais *threads* do que SPs (cujo númeor já é da ordem dos milhares

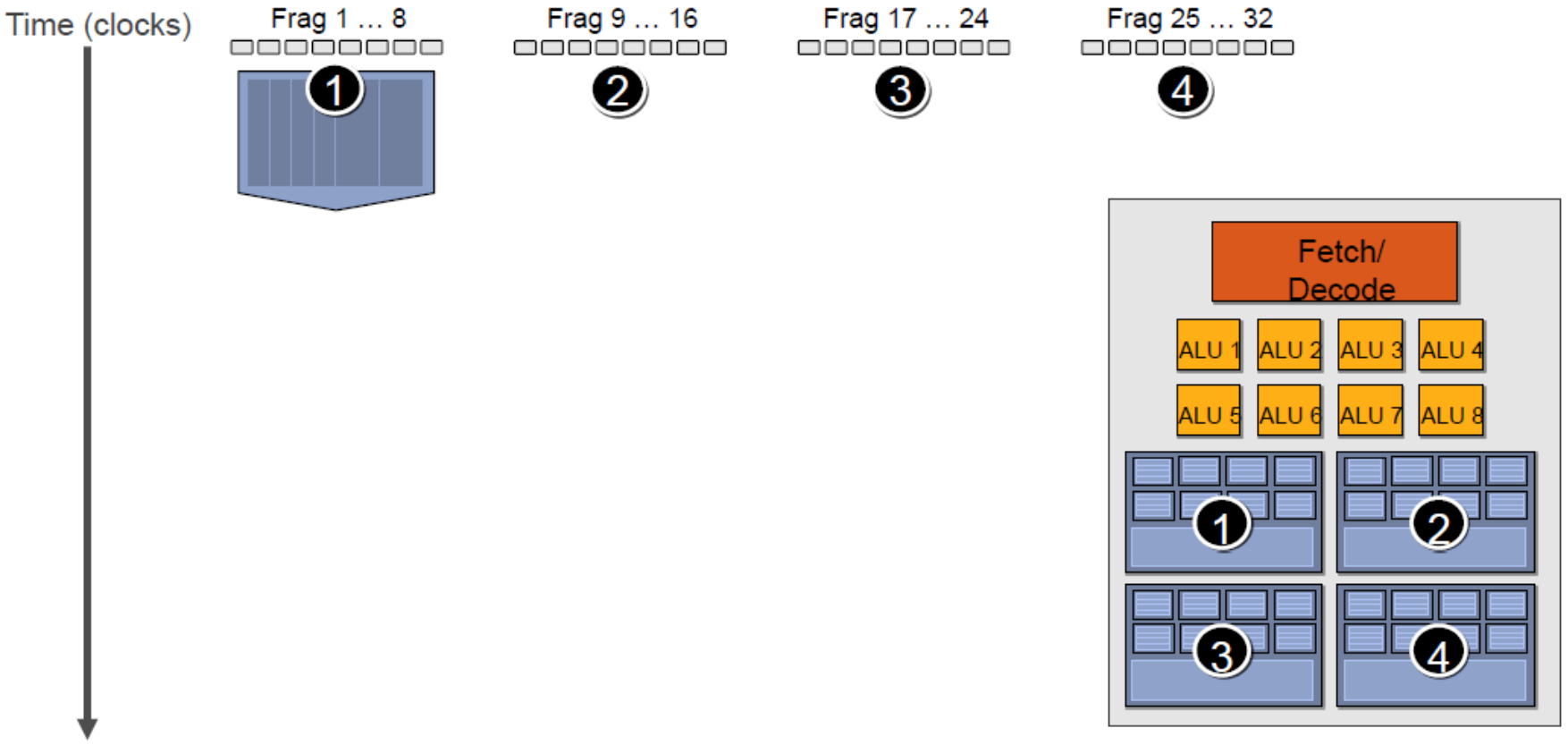
  Consequência: requer programas massivamente paralelos cujas tarefas sejam de grão muito fino!

Interleaving threads

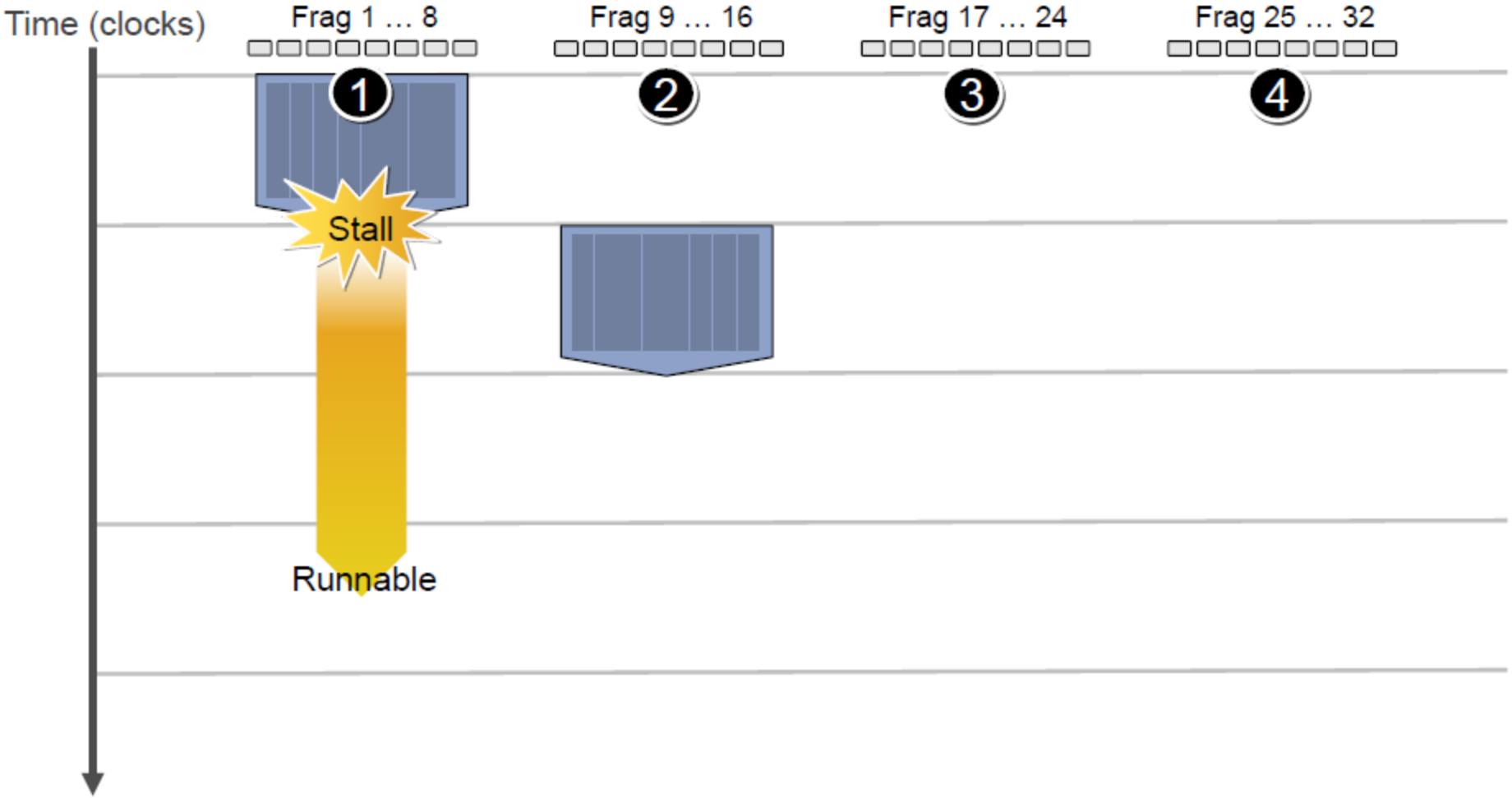




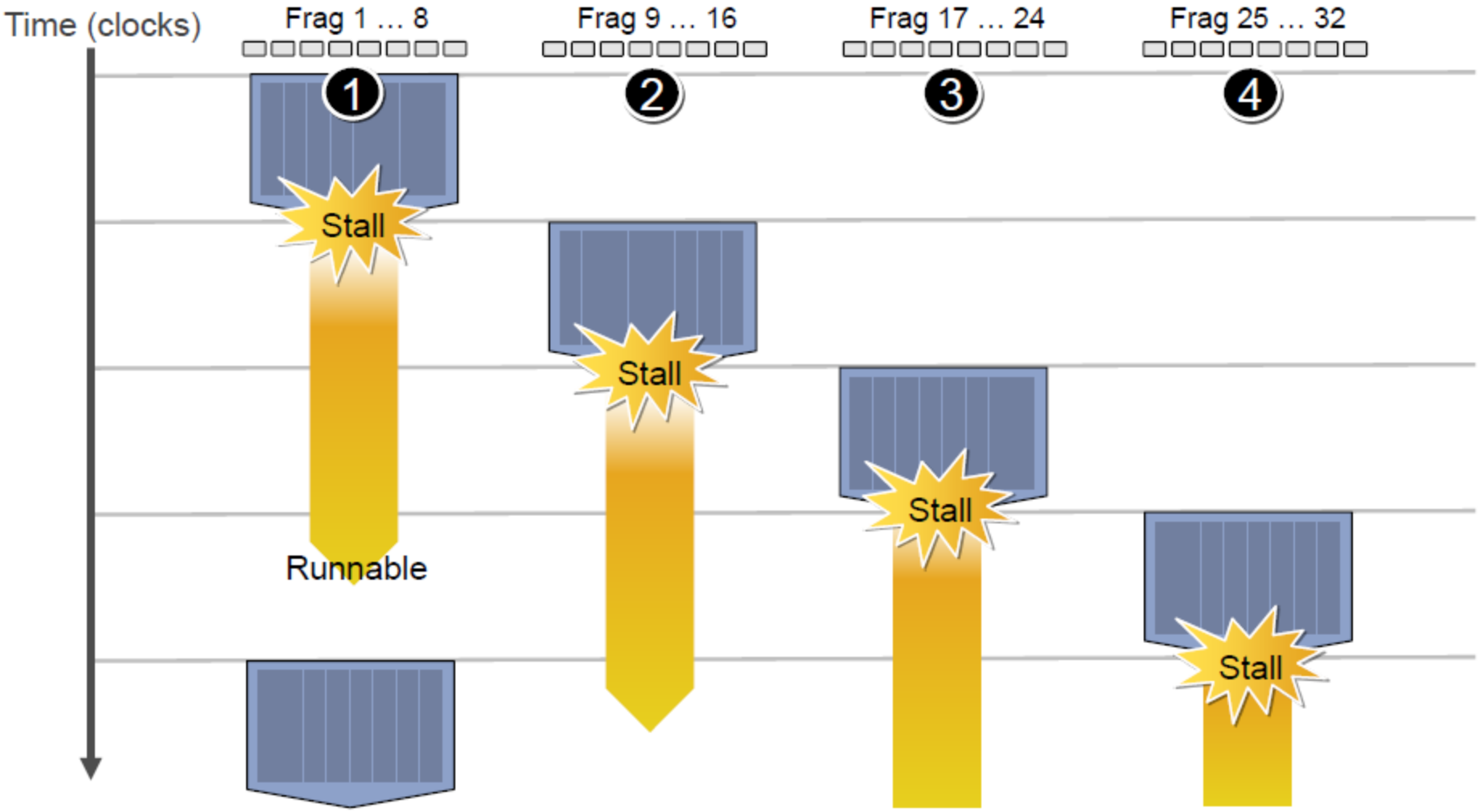
Interleaving threads



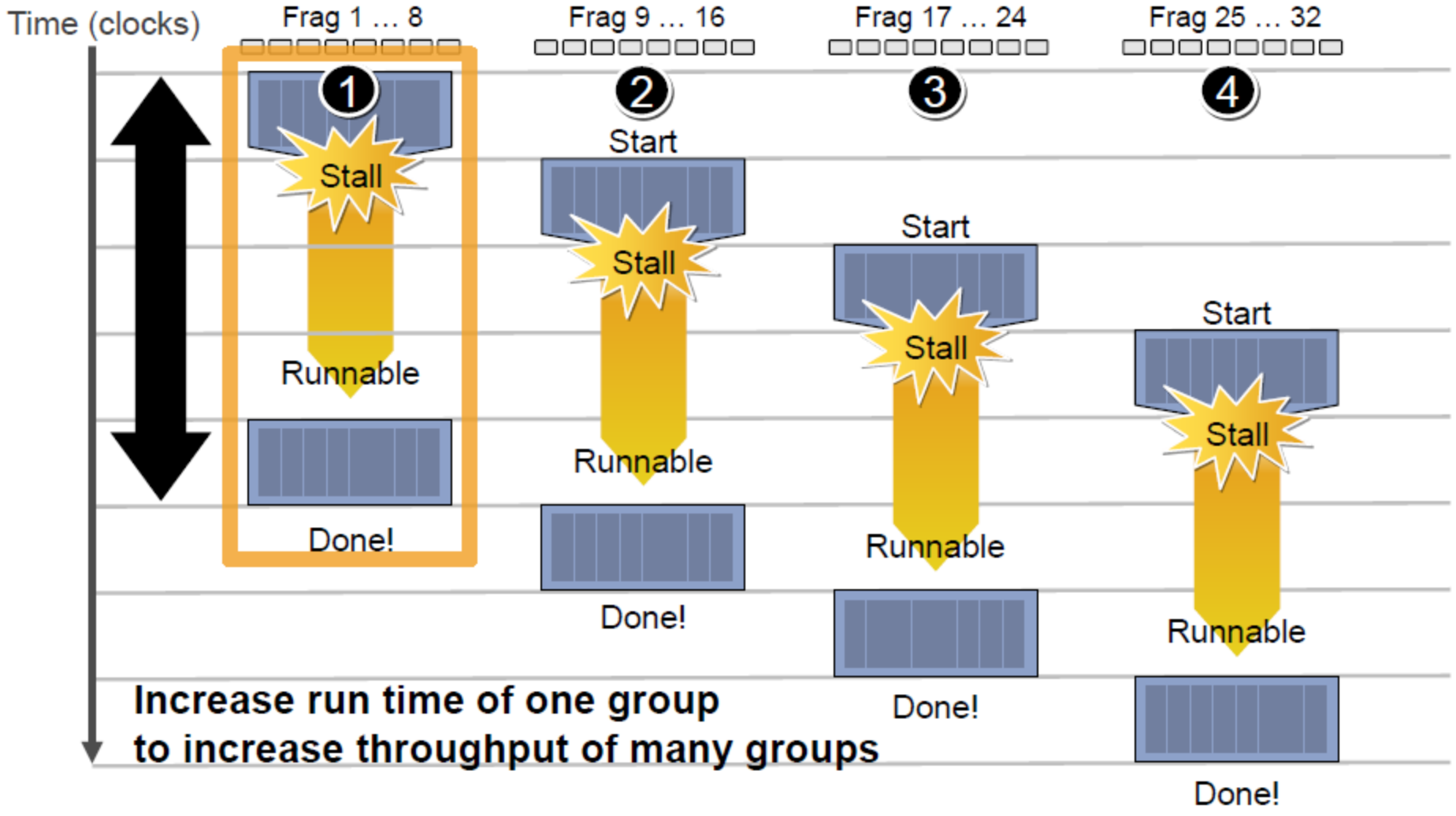
Interleaving threads



Interleaving threads



Interleaving threads

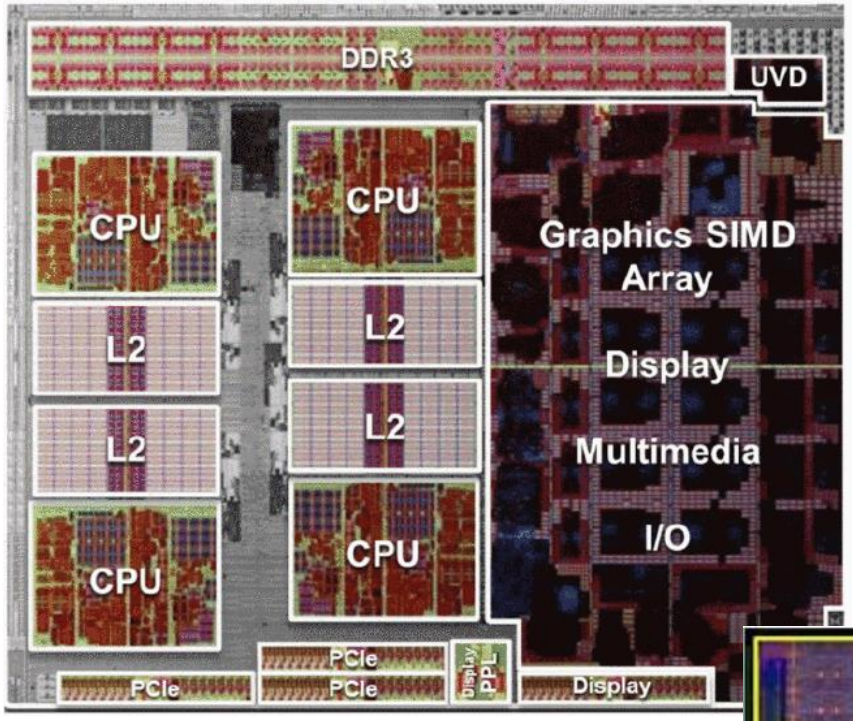


Comunicação CPU – GPU

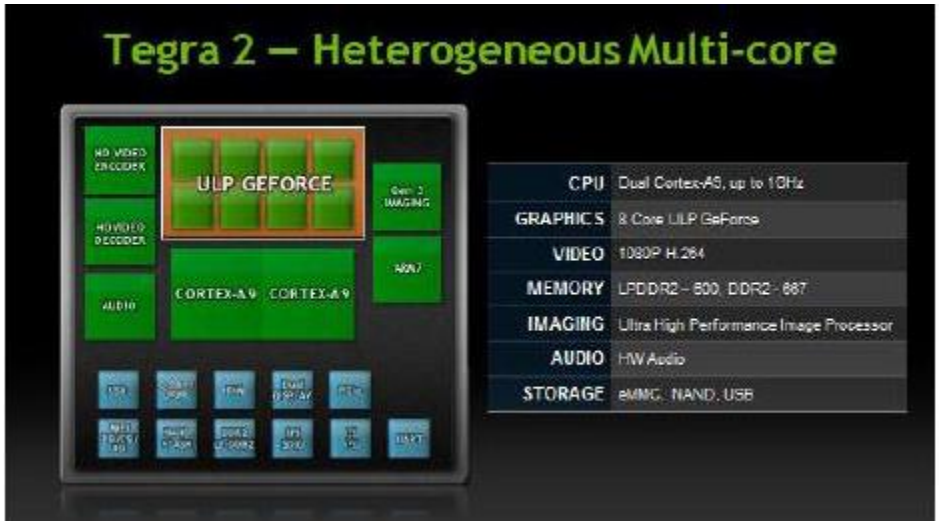
GPUs em placas gráficas

- **Bottleneck:** Os dados têm que ser transferidos da memória do sistema para a memória da placa (e os resultados eventualmente transferidos no outro sentido) através do barramento PCI
- **Tendência:** Integração do CPU com o GPU, reduzindo o custo de comunicação entre ambos
  - AMD A-Series
  - Intel Sandy Bridge
  - NVIDIA Tegra2

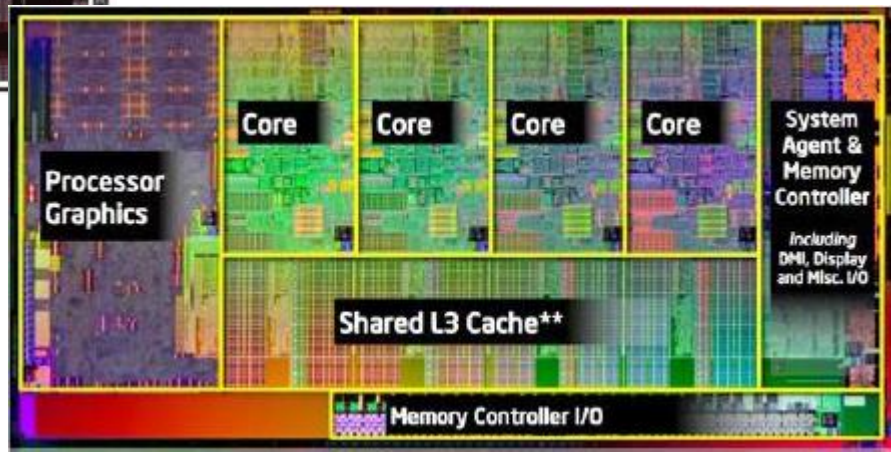
Processadores Heterogéneos



AMD APU



NVIDIA Tegra 2



Intel SandyBridge

Modelos de Programação

Model	GPU	CPU Equivalent
Vectorizing Compiler	PGI CUDA Fortran	gcc, icc, etc.
“Drop-in” Libraries	cuBLAS	ATLAS
Directive-driven	OpenACC, OpenMP-to-CUDA	OpenMP
High-level languages	pyCUDA, OpenCL, CUDA	python
Mid-level languages		pthreads + C/C++
Low-level languages	-	PTX, Shader
Bare-metal	Assembly/Machine code	SASS

GEMM “à la CUDA” – CPU code

```
#define N 512
#define Ne (N*N)
float A[Ne], B[Ne], C[Ne];
main (){
  float *devA, *devB, *devC;

  ini_matrix (Ne, A, B);
  GPU_Alloc (Ne, devA, devB, devC);
  MemCpy (CPU2GPU, Ne, A, devA, B, devB);

  GPU_GEMM <<< Ne threads >>> (N, devA, devB, devC);

  MemCpy (GPU2CPU, Ne, devC, C);
  GPU_Free (devA, devB, devC);

  printf (“That’s all, folks!\n”);
}
```

GEMM “à la CUDA” – GPU code

```
/*
 * Note: there are as many threads as elements in
 * each matrix
 */

GPU_GEMM (int N, float *A, float *B, float *C){
  const int tid = get_my_tid();

  // each GPU thread will process an element of C
  const int row = tid /N, col = tid % N;
  float lc=0.0;

  for (int k= 0 ; k<N ; k++)
    lc += A[row*N+k] * B[k*N+col];

  C[tid] = lc;
}
```