Plataformas de Programação Paralela

Programação Paralela

Aula 3
Alessandro L. Koerich

Pontifícia Universidade Católica do Paraná (PUCPR)
Ciência da Computação – 6º Período

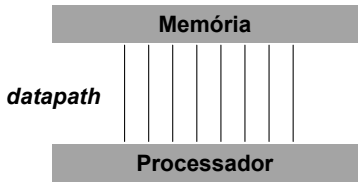
alekoe@ppgia.pucpr.br

Plano de Aula

- Introdução
- Paralelismo Implícito
- *Pipelining*
- Execução Superescalar
- Limitações da Performance da Memória
- Exemplos
- Resumo

Introdução

- Visão tradicional de um computador seqüencial



- Todos os três componentes apresentam gargalos para o desempenho global de um sistema de computação
- Solução: Inovações na arquitetura ⇨ Multiplicidade

Introdução

- Apresentar algumas tendências das arquiteturas de computadores com o objetivo de:
  - Entender suas limitações
  - Entender como elas influenciam o desenvolvimento de algoritmos e códigos

Paralelismo Implícito

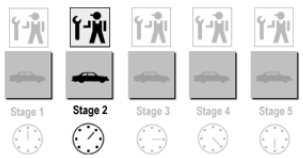
- Avanços significantes na velocidade do *clock* expuseram uma variedade de outras limitações
- Tal avanço é severamente diluído pelas limitações da tecnologia de memórias.
- Rotas alternativas para ganhos de performance com uma boa relação custo–eficiência
- Altos níveis de integração ⇨ de que maneira utilizá–los melhor ⇨ múltiplas instruções em um único ciclo de *clock*

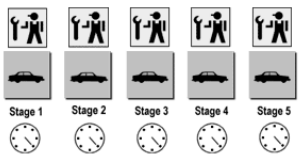
Paralelismo Implícito

- Técnicas que permitem a execução de múltiplas instruções em um único ciclo de *clock*.
  ⇨ *Pipelining*

Pipelining

- Analogia ⇨ Linha de produção onde cada parte de um automóvel é montada por uma equipe





- Fig 1. Somente um automóvel é montado
- Fig 2. Cinco automóveis são montados

Pipelining

De volta ao mundo dos computadores....

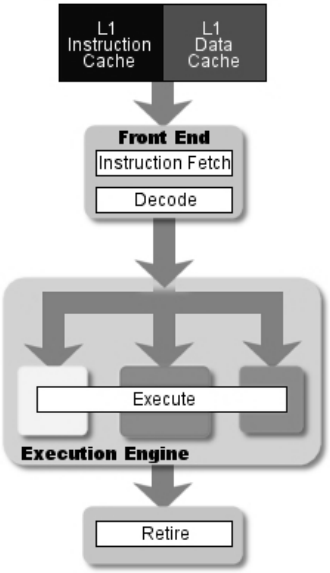
- Um computador basicamente repete somente 4 passos básicos para executar um programa:

1. Pega a instrução do endereço armazenado no contador.
2. Armazena esta instrução no registrador, decodifica e incrementa o endereço no contador.
3. Executa a instrução no registrador.
4. Escreve os resultados da instrução do ALU de volta no registrador de destino

Pipelining

- Retornando a analogia com a linha de produção, uma vez a execução *pipeline* introduzida, os 4 estágios agem como 4 estágios em uma linha de produção de automóveis regulares.
- Quando o *pipeline* está em plena capacidade, cada estágio está ocupado trabalhando sobre uma instrução e o *pipeline* é capaz de expelir uma instrução logo após a outra.
- Se cada estágio necessita de 10 ms para encerrar, então o *pipeline* completo pode processar uma instrução a cada 10ms.

Pipelining



Pipelining

- Exemplo: Pentium 4 @ 2.0 GHz tem um *pipeline* de 20 estágios
- Uma outra maneira de melhorar a taxa de execução de instruções?
- ... utilizar múltiplos *pipelines*

Pipelining

Features and Benefits	AMD Athlon™ XP	Pentium® 4
QuantiSpeed™ Architecture • Optimum balance between work per clock cycle and operating frequency	Yes	No
Operations per clock cycle • More operations per clock cycle enables faster software execution	9	6
Integer Pipelines	3	4
Floating Point Pipelines	3	2
Full x86 Decoders	3	1
L1 Cache Size	128KB	12k µop (Trace Cache) + 8KB (Data Cache)
L2 Cache Size	256KB, 512KB (on-chip)	512KB (on-chip)
Total on-chip full-speed cache	384KB, 640KB	520KB + 12K µop (on-chip)

Execução Superescalar

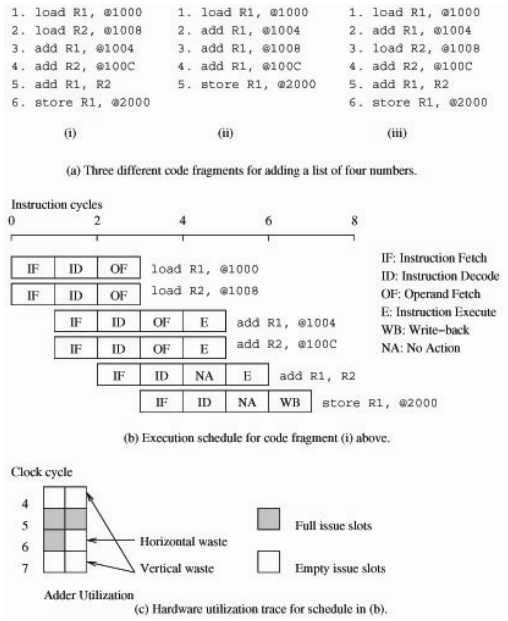
- Os projetistas de processadores tem cada vez mais transistores para trabalhar no projeto de um novo chip.
- O que fazer com tantos transistores?
- Uma das primeiras idéias era de que eles podiam colocar mais de uma ALU em um chip e ter ambas trabalhando em paralelo para processar as instruções mais rapidamente.

Execução Superescalar

- Mais de uma operação escalar de um só vez
  ⇨ Computadores superescalares
- O IBM RS6000 (1990) foi a primeira CPU superescalar.
- Em 1993, a Intel lançou o Pentium com 2 ALUs e trouxe o mundo x86 para a era superescalar.

Execução Superescalar

- Exemplo: Considere um processador com 2 *pipelines* e capacidade de executar 2 instruções



Execução Superescalar

- Parece simples, porém alguns problemas devem ser resolvidos . . .

Execução Superescalar

- **Dependência de Dados**
  - Os resultados de uma instrução podem ser necessários para as instruções subseqüentes
  - Dependências destes tipos devem ser resolvidas antes de enviar a instrução
    - A resolução é feita no *runtime* ⇨ hardware deve suportar
    - A quantidade de paralelismo a nível de instruções em um programa é normalmente limitado
    - Exemplo 2.1 i, ii e iii

Execução Superescalar

- **Dependência de Recursos**
  - Duas instruções competem por um único recurso do processador
  - Resulta dos recursos finitos compartilhados por vários pipelines
  - Ex. Co–agendamento de 2 operações de ponto flutuante em uma máquina com um único FPU.

Execução Superescalar

- **Dependência de Seção ou Dependência Procedural:**
  - Considere a execução de uma seção com instrução condicional. O destino da seção somente é conhecido no ponto de execução
  - O agendamento a priori através de seções condicionais pode levar a erros.
- A habilidade do processador em detectar e agendar instruções concorrentes é critico para performance superescalar.
- Analisar o exemplo da figura anterior

Execução Superescalar

- Freqüentemente, devido ao paralelismo limitado, dependências de recursos ou inabilidade do processador em extrair paralelismo

  ⇨ os recursos dos processadores superescalares são subutilizados
- Microprocessadores atuais tipicamente suportam até 4 execuções superescalares

Limitações da Performance da Memória

- A performance efetiva de um programa em um computador não depende somente da velocidade do processador
- Depende também da habilidade do sistema de memória em alimentar o processador com dados

Limitações da Performance da Memória

Em nível lógico:

- O sistema de memória recebe um pedido por uma palavra da memória
- Retorna um bloco de dados de tamanho *b* contendo a palavra após *l* nanosegundos
- *l* : latência da memória
- *Bandwidth*: taxa na qual os dados podem ser enviados da memória para o processador

Limitações da Performance da Memória

- Latência e *bandwidth* são aspectos críticos para determinar a performance da memória
- Analisaremos estes dois aspectos através de alguns exemplos
- Assumir: Bloco de memória ⇨ uma palavra

Limitações da Performance da Memória

Example 2.2 Effect of memory latency on performance

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The peak processor rating is therefore 4 GFLOPS. Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data. Consider the problem of computing the dot-product of two vectors on such a platform. A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch. It is easy to see that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating. This example highlights the need for effective memory system performance in achieving high computation rates. ■

Limitações da Performance da Memória

- Para aliviar a disparidade entre as velocidades do processador e *DRAM*
  ⇨ inovações arquitetônicas
- Colocar uma memória pequena e rápida entre processador e DRAM
  ⇨ Cache
- Cache ⇨ baixa latência e alta *bandwidth*
- *Hit ratio*: fração de dados servida pelo cache

Limitações da Performance da Memória

Example 2.3 Impact of caches on memory system performance

As in the previous example, consider a 1 GHz processor with a 100 ns latency DRAM. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle (typically on the processor itself). We use this setup to multiply two matrices A and B of dimensions 32×32. We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C. Once again, we assume an ideal cache placement strategy in which none of the data items are overwritten by others. Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 μs. We know from elementary algorithmics that multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 μs) at four instructions per cycle. The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., $200 + 16$ μs. This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS. Note that this is a thirty-fold improvement over the previous example, although it is still less than 10% of the peak processor performance. We see in this example that by placing a small cache memory, we are able to improve processor utilization considerably. ■

Limitações da Performance da Memória

- Largura de banda da memória: taxa pela qual dados podem ser movidos entre memória e processador.
- É determinado pelo bus de memória
- Técnica: aumento no tamanho dos blocos de memória
- Computadores comuns fornecem 2 a 8 palavras juntas no cache

Limitações da Performance da Memória

Example 2.4 Effect of block size: dot-product of two vectors

Consider again a memory system with a single cycle cache and 100 cycle latency DRAM with the processor operating at 1 GHz. If the block size is one word, the processor takes 100 cycles to fetch each word. For each pair of words, the dot-product performs one multiply-add, i.e., two FLOPs. Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS as illustrated in Example 2.2.

Now let us consider what happens if the block size is increased to four words, i.e., the processor can fetch a four-word cache line every 100 cycles. Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles. This is because a single memory access fetches four consecutive words in the vector. Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS. Note that increasing the block size from one to four words did not change the latency of the memory system. However, it increased the bandwidth four-fold. In this case, the increased bandwidth of the memory system enabled us to accelerate the dot-product algorithm which has no data reuse at all.

Another way of quickly estimating performance bounds is to estimate the cache hit ratio, using it to compute mean access time per word, and relating this to the FLOP rate via the underlying algorithm. For example, in this example, there are two DRAM accesses (cache misses) for every eight data accesses required by the algorithm. This corresponds to a cache hit ratio of 75%. Assuming that the dominant overhead is posed by the cache misses, the average memory access time contributed by the misses is 25% at 100 ns (or 25 ns/word). Since the dot-product has one operation/word, this corresponds to a computation rate of 40 MFLOPS as before. A more accurate estimate of this rate would compute the average memory access time as $0.75 \times 1 + 0.25 \times 100$ or 25.75 ns/word. The corresponding computation rate is 38.8 MFLOPS. ■

Limitações da Performance da Memória

- Fisicamente, o exemplo anterior corresponde a um bus de dados largo (4 palavras ou 128bits) conectado à múltiplos bancos de memória ⇨ construção muito cara
- Na prática, palavras consecutivas são enviadas ao *bus* de memória em ciclos de bus subseqüentes,após a primeira palavra ser recuperada.
- Assume que palavras consecutivas na memória são usadas por instruções sucessivas.

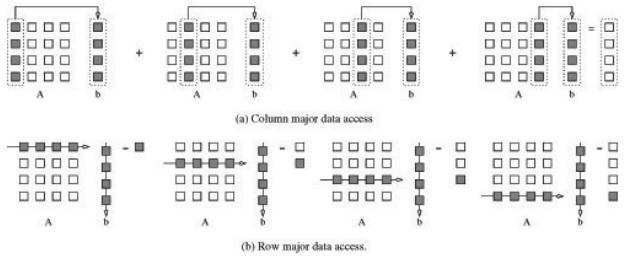
Limitações da Performance da Memória

Example 2.5 Impact of strided access

Consider the following code fragment:

```
1  for (i = 0; i < 1000; i++)
2          column_sum[i] = 0.0;
3          for (j = 0; j < 1000; j++)
4                  column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix `b` into a vector `column_sum`. There are two observations that can be made: (i) the vector `column_sum` is small and easily fits into the cache; and (ii) the matrix `b` is accessed in a column order as illustrated in Figure 2.2(a). For a matrix of size 1000 × 1000, stored in a row-major order, this corresponds to accessing every 1000^{th} entry. Therefore, it is likely that only one word in each cache line fetched from memory will be used. Consequently, the code fragment as written above is likely to yield poor performance. ■



Limitações da Performance da Memória

- A falta de localidade espacial na computação, provoca uma performance do sistema de memória muito pobre.

Limitações da Performance da Memória

Example 2.6 Eliminating strided access

Consider the following restructuring of the column-sum fragment:

```
1  for (i = 0; i < 1000; i++)
2          column_sum[i] = 0.0;
3  for (j = 0; j < 1000; j++)
4          for (i = 0; i < 1000; i++)
5                  column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order as illustrated in Figure 2.2(b). However, the reader will note that this code fragment relies on the fact that the vector `column_sum` can be retained in the cache through the loops. Indeed, for this particular example, our assumption is reasonable. If the vector is larger, we would have to break the iteration space into blocks and compute the product one block at a time. This concept is also called ***tiling*** an iteration space. The improved performance of this loop is left as an exercise for the reader. ■

Limitações da Performance da Memória

- Enquanto as taxas de pico dos processadores cresceram significativamente, a latência e largura de banda de memória não acompanhou a mesma taxa de crescimento.
- Conseqüentemente, para computadores típicos, a razão entre pico FLOPS e pico largura de banda de memória é algo entre 1MFLOPS/MBs to 100MFLOPS/ MBs (FLOPS per MB/second of bandwidth)

Limitações da Performance da Memória

Example 2.7 Threaded execution of matrix multiplication

Consider the following code segment for multiplying an $n \times n$ matrix a by a vector b to get vector c.

```
1   for (i = 0; i < n; i++)
2       c[i] = dot_product(get_row(a, i), b);
```

This code computes each element of c as the dot product of the corresponding row of a with the vector b. Notice that each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
1   for (i = 0; i < n; i++)
2       c[i] = create_thread(dot_product, get_row(a, i), b);
```

The only difference between the two code segments is that we have explicitly specified each instance of the dot-product computation as being a thread. (As we shall learn in Chapter 7, there are a number of APIs for specifying threads. We have simply chosen an intuitive name for a function to create threads.) Now, consider the execution of each instance of the function dot_product. The first instance of this function accesses a pair of vector elements and waits for them. In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on. After l units of time, where l is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation. In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation. ■

Limitações da Performance da Memória

Example 2.6 Eliminating strided access

Consider the following restructuring of the column-sum fragment:

```
1   for (i = 0; i < 1000; i++)
2           column_sum[i] = 0.0;
3   for (j = 0; j < 1000; j++)
4           for (i = 0; i < 1000; i++)
5                   column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order as illustrated in Figure 2.2(b). However, the reader will note that this code fragment relies on the fact that the vector column_sum can be retained in the cache through the loops. Indeed, for this particular example, our assumption is reasonable. If the vector is larger, we would have to break the iteration space into blocks and compute the product one block at a time. This concept is also called ***tiling*** an iteration space. The improved performance of this loop is left as an exercise for the reader. ■

Limitações da Performance da Memória

Example 2.9 Impact of bandwidth on multithreaded programs

Consider a computation running on a machine with a 1 GHz clock, 4-word cache line, single cycle access to the cache, and 100 ns latency to DRAM. The computation has a cache hit ratio at 1 KB of 25% and at 32 KB of 90%. Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB. If the computation makes one data request in every cycle of 1 ns, in the first case the bandwidth requirement to DRAM is one word every 10 ns since the other words come from the cache (90% cache hit ratio). This corresponds to a bandwidth of 400 MB/s. In the second case, the bandwidth requirement to DRAM increases to three words every four cycles of each thread (25% cache hit ratio). Assuming that all threads exhibit similar cache behavior, this corresponds to 0.75 words/ns, or 3 GB/s. ■

Resumo

- Apresentamos vários fatores que influenciam a performance de programas seriais ou programas paralelos implícitos.
- Motivações para o paralelismo
  - Crescente distância entre performance de pico e sustentável dos processadores atuais
  - O impacto da performance dos sistemas de memória
  - A natureza distribuída de muitos problemas

Próxima Aula

- Dicotomia de Plataformas de Computação Paralela
  - Estrutura de Controle
  - Modelos de Comunicação
- Organização Física de Plataformas Paralelas