Curso: LEF/MEI/MIEF/MIEI
U.C.: Computação/Programação Paralela

Teste 19-jan-2022
Duração: 2h30m

Nota: Este teste tem 3 partes que devem respondidas em 3 cadernos distintos para permitir que cada um dos 3 docentes das UCs possa corrigir em paralelo cada uma das 3 partes até ao fim de semana.

Parte I (60%)

1. [45]Dois modelos de programação são atualmente usados nos sistemas de computação homogéneos de médio e grande porte com várias unidades de processamento: (i) em sistemas com memória partilhada, (ii) em sistemas com memória distribuída e (iii) em sistemas heterogéneos com GPUs (CUDA ou OpenCL)..

   a) [20]**Caracterize** e compare os modelos de memória partilhada e distribuída, na perspetiva (i) da arquitetura do sistema de computação e (ii) do respetivo modelo de programação.

   b) [10]Uma tentação que os *chips manycore* podem trazer é tirar partido da memória que é fisicamente partilhada por todos os *cores*, com uma solução de programação baseada em muitas *threads* num mesmo processo. Contudo, há vários fatores que podem limitar seriamente a escalabilidade dessas aplicações num paradigma de memória partilhada.
   Identifique algumas dessas limitações.

   c) [15]Considere uma aplicação científica para execução num *dual-socket server* (com *8-core x86-64 devices*) equipado com Nvidia Tesla Volta GPUs, cujo código é integralmente executado no acelerador Tesla.
   Fazendo o *profile* dessa aplicação constatou-se que cerca de 20% de atividades são para execução em modo sequencial ou com paralelismo limitado a operações de teste e de *sorting* com inteiros, e os restantes 80% da aplicação contém pesadas operações algébricas com valores FP.
   Comente a estratégia usada no desenvolvimento deste código e **apresente sugestões** (fundamentadas) para melhorar o desempenho da sua execução neste servidor.

2. [25]O mais recente processador x86-64 para servidores, desenvolvido pela AMD, o EPYC baseado na microarquitetura Zen 3 (também conhecido por "Milan"), é uma das famílias de processadores que vai equipar o Decalion no MACC.
   O Milan pode trabalhar com uma frequência de relógio de 3 GHz e ter até 64 *cores*, e cada *chip* tem 8 *memory channels* para aceder a memória DDR4-3200 (com capacidade de transferir 25 600 MB/sec).

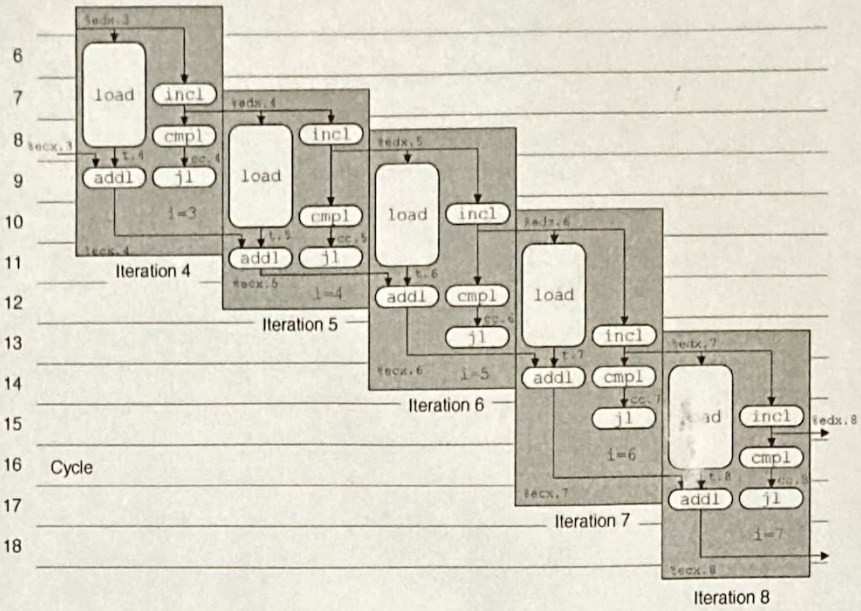
   a) [15]Se todos os *cores* precisarem de aceder no mesmo ciclo de *clock* a valores *dual-precision FP* que ainda estão em memória (um valor por core, e assumindo que as *caches* com dados ainda estão vazias), **mostre**, apresentando os cálculos que efetuar, se a largura de banda agregada deste *chip* é ou não suficiente para satisfazer, com a mesma latência, esta solicitação de todos os *cores*.

   b) [10]**Repita a resolução**, mas considerando agora que o valor a ir buscar à memória é um *array* de 8 valores *dual-precision FP*.

3. [15]Considere o seguinte código para execução num processador com uma arquitetura básica elementar, i.e., em que o ciclo repetitivo de execução de instruções segue uma sequência simples de passos: buscar uma instrução à memória, descodificá-la, colocar o IP a apontar para a próxima instrução, buscar os operandos à memória se necessário, executar a operação descodificada e guardar o resultado da operação, em registo ou na memória.

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP data[i];
  *dest = t;
}
```

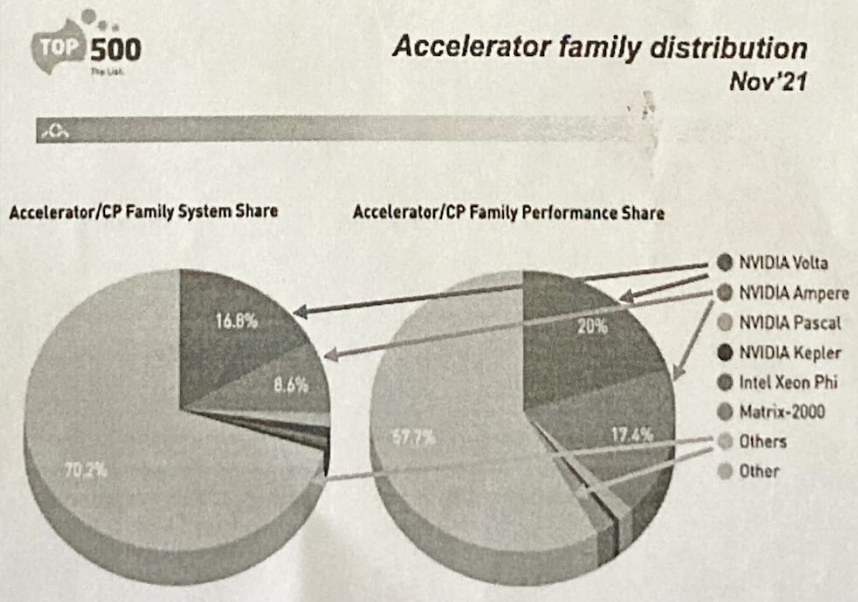
O ciclo `for` neste código tem, numa dada arquitetura x-86, o seguinte comportamento ao longo de 5 iterações.



Para que tal comportamento seja possível, várias alterações tiveram que ser feitas nessa arquitetura para melhorar consideravelmente o desempenho deste código, nomeadamente permitir que a execução das 5 instruções em cada iteração do ciclo `for`, cada com uma latência de alguns ciclos de *clock*, pudesse ser realizada em apenas 2 ciclos de *clock*.
Identifique essas melhorias e **explique** sucintamente como permitiram melhorar o desempenho da execução do código.

4. [20]Com o crescente aumento da diferença entre a frequência do *clock* na execução de instruções num processador e o tempo que os circuitos de memória levam a responder a pedidos do acesso a dados feitos pelo processador, os arquitetos de processadores tiveram de arranjar forma de esconder essa longa latência no acesso a dados em memória.
Identifique e **caracterize** sumariamente a estratégia seguida por estes engenheiros **e** a que seguiram os arquitetos dos GPUs da Nvidia para atacarem o mesmo problema

5. [15]Considere o seguinte gráfico do TOP500 com a distribuição das famílias de aceleradores, baseado em dados do TOP500 de nov-21, em termos do nº de sistemas HPC e da *performance* global. A ordem dos aceleradores na *pie chart* é a mesma que a ordem da legenda.
Teça comentários adequados e pertinentes relativamente ao que observa.



Parte Prática (40%)

Considere a função `calculo_matriz` que efetua várias operações sobre uma matriz e que será executada num servidor com um processador Xeon Ivy Bridge (igual ao utilizado durante as aulas práticas, Intel® Xeon® E5-2695 v2).

```
void mult_matriz  (float *a, float *b, float *c, int n) {
    int i, j, k;
    float cij;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            cij = c[i*n+j];
            for(k = 0; k < n; k++ ) {
                cij += a[i*n+k] * b[k*n+j];
            }
            c[i*n+j]=cij;
        }
    }
}
float soma_matriz  (float *m, int n) {
    int i, j;
    float soma=0;
    for (j = 0; j < n; ++j) {
        for (i = 0; i < n; ++i) {
            soma += m[i*n+j];
        }
    }
}

void escala_matriz  (float *m, int n, float escala) {
    int i, j;
    float soma;
    for (j = 0; j < n; ++j) {
        for (i = 0; i < n; ++i) {
            m[i*n+j]+=escala;
        }
    }
}

void calculo_matriz  (float *a, float *b, float *c, int n) {
    int escalar;
    //Fase 1
    mult_matriz (a, b, c, n);
    //Fase 2
    escalar = soma_matriz (c, n)
    //Fase 3
    escala_matriz (c,n, escala)
}
```

Parte II

1. [20]Pretende-se analisar e otimizar a versão sequencial deste algoritmo.

   a) [6]**Indique** a complexidade de cada uma das fases de execução do algoritmo (considere `n` o tamanho da linha da matriz).

   b) [7]**Indique**, justificando, a fase de execução com maior benefício em ser otimizada.

   c) [7]**Apresente** duas otimizações possíveis e **discuta** o seu impacto, nomeadamente nos acessos à memória e no número de instruções executadas.

2. [30]Pretende-se desenvolver uma versão paralela da função `escala_matriz` para memória distribuída com **passagem de mensagens (i.e., MPI)** e em que o processo *master* contém a matriz original e que no final da execução deverá conter a matriz processada.

   a) [5]**Identifique**, justificando, qual o padrão que se adequa melhor à resolução do problema com MPI, a partir dos padrões paralelos *pipeline* e *farm*.

   b) [10]**Identifique** as trocas de mensagens necessárias entre os vários processos, para o padrão escolhido anteriormente (**ilustre** através de um diagrama de troca de mensagens com 3 processos).

   c) [15]**Implemente** a versão proposta anteriormente.

Parte III

1. [15]**Desenvolva** uma versão paralela das funções `mult_matriz` e `soma_matriz` em **OpenMP**, e **comente** o propósito de cada uma das diretivas que usar no contexto destas funções.

2. [15]**Complete** o *kernel* CUDA apresentado em baixo, responsável por escalar um vetor unidimensional passado como argumento (semelhante à função `escala_matriz`).
   Ignore a chamada do *kernel*, as alocações e as cópias de memória entre *host* e *device*.

```
__global__
void escala_vec_kernel (float *vec, int n, int escala) {
    int thread_id = threadId.x + blockId.x * blockDim.x;

    // escalar o vetor vec
```