1. [30]Considere o seguinte código para execução num processador com uma arquitetura básica elementar.

```
void combine4(float *data, float *dest, int length)
{
  float t = 0;
  for (int i = 0; i < length; i++) {
    t = t + data[i];
  }
  *dest = t;
}
```

a) [10]Apresente um grafo com as dependências de dados para 5 iterações do ciclo `for` (considere apenas as operações aritméticas básicas e os acessos à memória).

b) [10]Calcule quantos ciclos de relógio serão necessários para executar essas 5 iterações do ciclo `for`, numa arquitetura ideal (recursos ilimitados), onde que cada operação demora um ciclo de relógio. **Explique** sucintamente que otimizações permitiriam melhorar o desempenho da execução do código.

c) [10]Os compiladores atuais são bastante eficientes para conseguirem analisar ciclos `for` e vetorizar o código que lá se encontra. Se este código for compilado com indicação para vetorizar, **indique**, justificadamente, se o compilador iria conseguir fazê-lo.

1.a) A dependência de dados encontra-se na variável t. Na iteração i+1 do ciclo, lê-se o valor escrito na iteração i (portanto, uma dependência RAW). O grafo de dependências será portanto uma série de setinhas no sentido crescente dos ciclos de relógio, em que cada add está dependente do add anterior.

b) Com recursos ilimitados (o que evita stalls de memória), o único fator limitante é a RAW dependency na variável t. Nesta arquitetura ideal, teríamos 1 ciclo por add, ou seja um total de 5 ciclos de relógio nestas 5 iterações. Uma otimização possível seria fazer uma soma em árvore: dividir, por exemplo, as somas em índices pares e ímpares e depois fazer uma redução em árvore (isto pode ser posteriormente generalizado para mais do que duas variáveis auxiliares, melhorando ainda mais o tempo execução) -> um parallel unroll. Outra otimização possível seria, se possível, fazer um prefetching dos dados de data para a cache, o que tornaria os acessos mais rápidos.

c) Como já foi mencionado anteriormente, o ciclo detém uma dependência do tipo RAW. O compilador não vai vetorizar este ciclo precicsamente à custa desta dependência: a forma mais eficiente de o fazer seria por redução em árvore, o que significa que não o fará a menos que lhe seja especificamente indicado (com recurso a flags, por exemplo).

2. [20]**Caracterize** e compare o modelo de memória distribuída com os que usam GPUs, na perspetiva **(i)** da arquitetura do sistema de computação e **(ii)** do respetivo modelo de programação.

2. (i) Memória distribuída: Baseia-se na noção de processos. Cada processo é um nodo de computação isolado, com recursos próprios, i.e. registos, memória, unidades funcionais, que comunicam uns com os outros através do envio de mensagens (por ligações Ethernet, entre outras). GPU: As GPU (Graphical Processing Units) são constituídas por um conjunto de SMs, que oor sua vez contém vários CUDA cores, com milhares de registos, e uma memória global. Cada SM tem uma memória que lhe é local (cache / shared memory) e um warp scheduler, responsável por distribuir as threads pelos vários CUDA cores.

(ii)Memória distribuída: Modelo de programação SPMD (Same Program Multiple Data). Cada processo é atribuído com uma parte da computação (paralelização por dados ou funcional). O sincronismo entre diferentes processos é feito pelo envio/recibo de mensagens, que pode ser feito peer to peer ou entre todos os processos da rede de comunicação. Tem uma boa escalabilidade com o número de processos atribuídos. GPU: Modelo de programação SIMD (Same Instruction Multiple Data), otimizado para máximo throughput. As GPU estão logicamente organizadas (em termos abstratos) numa grelha de blocos. Cada bloco é atribuído um conjunto de threads, que são executados nos SM. Sincronismo só existe dentro do bloco, não entre blocos.

3. [10]Os servidores homogéneos *dual-socket* com *chips manycore* da Intel podem tirar partido da memória que é fisicamente partilhada por todos os *cores* do mesmo servidor. **Explique**, justificando, a diferença entre usar um modelo de programação baseado em muitas *threads* num mesmo processo, ou em muitos processos com poucas *threads* cada.

3. O uso dos modelos de programação depende das tarefas a executar e da arquitetura disponível. Usar muitas threads e poucos processos pode ser benéfico quando é necessário haver constante partilha de informação, uma vez que partilham o mesmo espaço de recursos. No entanto, pode dar origem a zonas de contenção no programa devido a locks para prevenir race conditions. Usar muitos processos e poucas threads reduz a contenção nestes casos quando é possível segmentar as tarefas em várias regiões independentes, com partilhas não tão regulares de informações entre processos, pois requer comunicação explícita, recorrendo a APIs como por exemplo MPI. Este último, e no caso de sistemas NUMA (Non Uniform Memory Access), pode ser mais vantajoso, por ser possível mapear processos a cores que estão mais próximos das zonas de memória que irão aceder, tirando melhor proveito da localidade de memória.

4. [10]Um processador x86-64 para servidores, desenvolvido pela AMD, o EPYC baseado na microarquitetura Zen 3 (também conhecido por "Milan), pode trabalhar com uma frequência de relógio de 3 GHz e ter até 64 *cores*, e cada *chip* tem 8 canais para aceder à memória DDR4-3200 (cada canal tem a capacidade de transferir 25 600 MB/sec).

Se todos os *cores* precisarem de aceder no mesmo ciclo de *relógio* a valores *double-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*.

4. Cada core precisa de 1 double, i.e. 8 bytes / core
Total de 8 * 64 = 512 bytes por clock cycle
Multiplicando pela frequência de clock, temos
512 bytes / cc * 3 cc / GHz = 1536 GB/s
No mínimo, para que não haja contenção e todos os cores recebam doubles com a mesma latência, teremos de ter uma bandwith >= 1536 GB/s
Temos 8 canais DDR4-3200 com 25.6 GB/s, um total de 8 * 25.6 = 204.8 GB/s
Não temos bandwidth suficiente para satisfazer a solicitação simultêanea de todos os cores no mesmo cc.

5. [10]Duas vezes por ano é divulgada a lista TOP500 com a distribuição ordenada dos 500 sistemas de computação mais potentes a nível mundial. **Mostre** como é construída essa lista.

5. A lista do top 500 é construída com base na performance do LINPACK: um conjunto de sistemas de equações lineares que são executadas no supercomputador a ser avaliado. Os resultados são construídos com base na melhor performance em termos do Rmax (PFlop/s) -> performance máxima alcançada no LINPACK. Também são listados o Rpeak (PFlop/s) -> o peak teórico da performance alcançável, o número de cores e a potência consumida. Resumidamente, determina o máximo de operações FP por segundo com uma benchmark e compara a performance entre supercomputadores.

1. [30]Pretende-se analisar e otimizar a versão sequencial deste programa.

a) [10]**Indique** a complexidade de cada uma das fases de execução do algoritmo e quais a(s) fases que apresentam maior benefício em ser otimizada(s).

b) [10]**Calcule** o número de *cache misses* no nível 1 originado pela execução da função `computeHistogram` (utilize os parâmetros do processador Xeon Ivy Bridge: 32 KiB de L1 e linhas com 64 bytes).

c) [10]**Apresente** duas otimizações possíveis e **discuta** o seu impacto no tempo de execução com base na equação de desempenho do processador.

Considere o programa seguinte, com várias funções, que efetua uma transformação de uma imagem, representada por uma matriz de valores inteiros compreendidos entre 0 e 255, com a dimensão indicada no código e que será executada num servidor com um processador Xeon Ivy Bridge (igual ao utilizado durante as aulas práticas, Intel® Xeon® E5-2695 v2).

```
#define N 10000

int imgIn[N][N];  // 256-gray input image
int imgOut[N][N]; // 256-gray output image

void init(int *h) {
    for(int i=0; i<256; i++)
        h[i]=0;
}

void computeHistogram(int imgIn[N][N], int *h) {
    int j, i;
    for(j=0; j<N; j++)
        for(i=0; i<N; i++)
            h[ imgIn[i][j] ]++;
}

void acumulateHistogram(int *h) {
    int i;
    for(i=1; i<256; i++)
        h[i] += h[i-1];
}

void transformImage(int imgIn[N][N], int *h, int imgOut[N][N]) {
    int j, i;
    for(j=0; j<N; j++)
        for(i=0; i<N; i++)
            imgOut[i][j] = h[ imgIn[i][j] ];
}

int main() {
    int hist[256];

    // ... /* read imgIn ... */

    init(hist);

    computeHistogram(imgIn,hist);

    acumulateHistogram(hist);

    transformImage(imgIn,hist,imgOut);

    // ... /* save imgOut ... */
}
```

1.a) As funções init e acumulateHistogram são lineares no tamanho do histograma (256), são na globalidade do programa irrelevantes para o tempo de execução. As funções transformImage e computeHistogram, por outro lado, são de complexidade O(N^2) com N = 10^4, o que significa que serão os hotspots do programa, e com certeza apresentarão maior benefício em ser otimizadas.

b) Por cada linha de imgIn, teremos N = 10^4 * 4 bytes (inteiros) = 40kB. Como o tamanho da cache L1 é 32KiB, nenhuma linha caberia inteira na cache. Como o padrão de acesso dos loops é feito por ordem ji (portanto, por linhas), cada iteração do ciclo leva a um cache miss no acesso a imgIn. O histograma h pode ser trazido para cache, pois 256 * 4 = 1024 = 1KiB, portanto todos os misses virão de acessos a imgIn, ou seja, N^2 = 10^8 misses por chamada à função.

c) Uma otimização mais óbvia seria a troca da ordem dos ciclos, uma vez que o resultado é independente disso, por forma a melhor aproveitar a localidade da cache, uma vez que cada fetch de dados traz 64 bytes / 4 bytes = 16 inteiros por cada miss. Isto reduz o número de misses por um fator de 16, baixando de 100M para 6.25M misses. De acordo com a equação de performance, T_exec = #I * CPI * T_cc esta melhoria diminui o número de clocks por cada iteração do ciclo, ou seja, reduz o CPI, e portanto levaria a uma melhoria do tempo de execução. Outra otimização possível seria fazer um unroll do ciclo com um prefetching do próximo bloco de 16 inteiros para cache. Isto levaria a uma redução do CPI e do número de

instruções, o que por sua vez levaria a um melhor T_exec.

2. [30]**Desenvolva** uma versão paralela com **OpenMP**.
   a) [10]**Discuta, apresentando** argumentos, se a função `acumulateHistogram` beneficia da execução paralela.
   b) [20]**Desenvolva** a versão paralela das funções `computeHistogram` e `transformImage`, e **comente** o propósito de cada uma das diretivas que usar no contexto destas funções.

2.a) A função acumulateHistogram não beneficia de execução paralela dada a dependência do tipo RAW que existe entre dois ciclos consecutivos (a iteração i+1 lê o valor h[i] que é esecrito durante a iteração i), já para não falar que é linear no tempo do histograma, o que se demonstra praticamente irrelevante quando comparado a funções quadráticas em N, como transformImage e computeHistogram.

b)

```
void computeHistogram(int imgIn[N][N], int *h) {
	int i, j, index;
	/* Explanation of the directives
	#pragma omp parallel creates a parallel fork region
	*/ for is a directive specific to for loops, partitioning the first loop's iteration space and assigning it to the threads in the parallel region. The collapse(2) clause makes it so that the API accounts for the total number of iterations of both loops together. schedule(static) is enabled by default, but here we explicitly cast to ensure the user that loads will be equal throughout all iterations, so there is no need to induce in synchronization overhead. private(i, j,index) ensures that loop indexes are different loop indexes to each thread, as well as
having separate index variables, preventing race conditions. */
	#pragma omp parallel for schedule(static) collapse(2) private(i, j, index)
	for(i=0; i<N; i++)
		for(j=0; j<N; j++){
			index = imgIn[i][j];
			/* We compute the index to know which memory position from h the thread will access, and if two different threads from the same parallel region are to access the same value of h, it makes sure they have a critical section that's shared only between those threads, to prevent unecessary locks with other threads that access different, safe positions */
			#pragma omp critical(index)
			h[index]++;
		}
}

void acumulateHistogram(int *h) {
	int i;
	for(i=1; i<256; i++)
		h[i] += h[i-1];
}

void transformImage(int imgIn[N][N], int *h, int imgOut[N][N]) {
	int i, j;

	/* Explanation of the directives: Same as the first explanation above. Here since no two iterations overlap in accessing the same element of imgOut, there is no race condition to worry about. */
	#pragma omp parallel for schedule(static) collapse(2) private(i, j)
	for(i=0; i<N; i++)
		for(j=0; j<N; j++)
			imgOut[i][j] = h[ imgIn[i][j] ];
```

}

3. [30]Pretende-se desenvolver um *kernel* em CUDA para implementar a função `transformImage`. Considere o *kernel* CUDA apresentado em baixo e ignore a chamada do *kernel*, as alocações e as cópias de memória entre *host* e *device*.

```
__global__
void transformImage (int *imgIn, int *hist, int *imgOut) {
     int thread_id = threadId.x + blockId.x * blockDim.x;

     ... // transformar imgIn
```

a) [20]**Complete** o *kernel* CUDA apresentado. Quantas *threads* deverão ser lançadas para executar este *kernel*? Justifique.

b) [10]**Desenvolva** uma versão **otimizada** do *kernel* CUDA implementado em **a)** tirando partido de *shared memory*. Quantas *threads* deverão ser lançadas por bloco de *threads*? Justifique.

3.a)

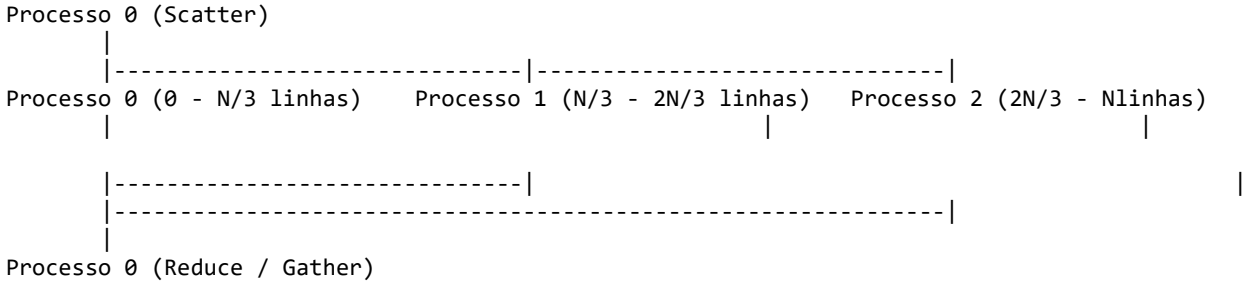
```
__global__
void transformKernel(int *imgIn, int *hist, int *imgOut) {
      int thread_id = blockIdx.x * blockDim.x * x + threadIdx.x;

      if (thread_id < N * N)
            imgOut[thread_id] = hist[imgIn[thread_id]];

      /* Para lançar este kernel, são necessárias, no mínimo, tantas threads
quantos elementos de imgIn e imgOut (N^2 = 10^8). Cada CUDA core recebe uma thread
e executa o store da imgOut correspondente à posição de memória que lhe é
atribuída com base no thread_id. */
}
```

b)

```
__global__
void transformKernel(int *imgIn, int *hist, int *imgOut) {
      int thread_id = blockIdx.x * blockDim.x * x + threadIdx.x;
      int lid = threadIdx.x;

      __shared__ int h[256];
      h[lid] = hist[lid];
      __syncthreads();

      if (thread_id < N * N)
            imgOut[thread_id] = h[imgIn[thread_id]];

      /* Uma otimização do Kernel acima é feita ao trazer o histograma para a
shared memory do SM. Cada posição do histograma pode ser repetidamente acedida o
que causa stalls desnecessários. Assim, chamando 256 threads por bloco
(correspondente ao número de valores do histograma), trazemos cada posição para a
shared memory e sincronizamos as threads antes de efetuar o store no imgOut. */
}
```

4.a) Trata-se de um problema que pode ser subdividido em várias regiões independentes (diferentes linhas, por exemplo, de imgOut), pelo que o padrão que mais se adequa neste caso (para obter melhor desempenho) seria farming: o processo 0 particiona a matriz imgOut por linhas, em 3 partes,
que distribui para os processos 0,1,2.

Diagrama de troca de mensagens (para ambas as funções computeHistogram e transformImage)



b)

```
void computeLocalHistogram(int imgIn[N][N], int *h, int row_start, int row_end) {
      int i, j;
      for(i=row_start; i<row_end; i++)
            for(j=0; j<N; j++)
                  h[ imgIn[i][j] ]++;
}

void transformLocalImage(int imgIn[N][N], int *h, int imgOut[N][N], int row_start,
int row_end) {
      int i, j;
      for(i=row_start; i<row_end; i++)
            for(j=0; j<N; j++)
                  imgOut[i][j] = h[ imgIn[i][j] ];
}

int main(int argc, char* argv[]) {
      int hist[256], rank, size;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      int loc_imgIn[N / size][N], loc_imgOut[N / size][N], loc_hist[256];

      // Process 0 has the original image
      if (rank == 0) {
            /* Read imgIn */
      }

      // Initialize histogram locally for each node
      init(local_hist);

      // Compute histogram (each process treats 1/size of the matrix's rows)
      computeLocalHistogram(imgIn, local_hist, rank * (N / size), (rank + 1) * (N /
size));

      // Reduce results in process 0
      MPI_Reduce(loc_hist, hist, 256, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

      // This is a simple task, so we compute it locally in node 0 and then
broadcast the results to other nodes
      if (rank == 0) {
            acumulateHistogram(hist);
      }
      MPI_Bcast(hist, 256, MPI_INT, 0, MPI_COMM_WORLD);

      // Scatter imgIn to other processes from root 0
      MPI_Scatter(imgIn, (N / size) * N, MPI_INT, loc_imgIn, (N / size) * N,
MPI_INT, 0, MPI_COMM_WORLD);
```

```
	// Transform image locally
	transformImage(loc_imgIn, hist, loc_imgOut, rank * (N / size), (rank + 1) *
(N / size));

	// Gather the results at the root process 0
	MPI_Gather(loc_imgOut, (N / size) * N, MPI_INT, imgOut, (N / size) * N,
MPI_INT, 0, MPI_COMM_WORLD);

	// Process 0 saves the output image
	if (rank == 0) {
		/* Save imgOut */
	}

	MPI_Finalize();
	return 0;
}
```

1. [40]Vários modelos de programação são atualmente usados nos sistemas de computação de médio e grande porte com várias unidades de processamento: (i) em sistemas homogéneos 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 distribuída com os que usam GPUs, na perspetiva **(i)** da arquitetura do sistema de computação e **(ii)** do respetivo modelo de programação.

   (i) Em sistemas de mem distrib: o acesso à mem remota é por msg entre PUs usando uma rede de interconexão de nós de baixa latência e elevada b/w e a execução em nós diferentes é autónoma

   Em sistemas com GPU o host tem acesso à mem do GPU para troca de dados entre as 2 memórias, usando normalmente PCIe (com baixa b/w), e a execução no GPU é controlada pelo host através da execução de kernels (CUDA)

   (ii) Em sistemas de mem distrib: modelo de programação usando comunicação de msg's (MPI) entre processos independentes em diferentes PUs

   Em sistemas com GPU usa-se um modelo de programação baseado em CUDA (ou OpenCL) que é responsável por gerar código executável distinto para o host e para os devices (GPUs) e com comunicação síncrona ou assíncrona entre o host e os devices.

   Comentários:

   Um processador é um *core* ou é um *chip multicore*?

   Ao longo desta UC foi usado sempre o termo PU como sinónimo de processador e correspondente à parte de processamento de um *core* (i.e., um *core* sem a hierarquia de memória) e nunca foi considerado um *chip multicore*, como alguns usaram na resposta a esta questão.

   E poucos ainda entenderam o significado de "*arquitetura do sistema de computação*"...

   Para esta questão esperava-se que dissessem que um sistema de computação de memória distribuída é aquele em que o(s) *chip(s) multicore* não partilham fisicamente do acesso à memória RAM ligada a outro(s) *chip(s) multicore*; já no modelo de programação é possível implementar este modelo num sistema com memória fisicamente partilhada, embora não exista de facto qualquer partilha de memória entre os *cores* que estejam a correr processos distintos.

   E quanto ao GPU era importante referir a diferença entre, por ex., um sistema *dual-socket* com 2 *chips* idênticos, dum sistema em que o 2º *chip* era um acelerador GPU (isto raramente acontece, apenas vimos isto na ligação do POWER9 da IBM ao GPU NVIDIA Volta, normalmente a ligação ao GPU é através de I/O por PCIe); mais concretamente, referir que, para além de a maioria das implementações físicas serem de memória distribuída, estes aceleradores não podem correr código sozinho, precisam de um hospedeiro para arrancar e correr parte do código, pelo menos a parte de código sequencial; e que os GPUs não correm C standard, correm uma adaptação do C a estes componentes, código este gerado por CUDA ou OpenCL, que cria a ilusão ao programador que está apenas a usar código C.

   b) [20]Os servidores homogéneos *dual-socket* com *chips manycore* da Intel podem tirar partido da memória que é fisicamente partilhada por todos os *cores* do mesmo servidor. **Explique**, justificando, se (i) nestes servidores o acesso à memória é UMA ou NUMA e (ii) a diferença entre usar-se o modelo de programação, em cada um destes servidores, baseado em muitas *threads* num mesmo processo, ou em muitos processos com poucas *threads* cada.

   (i) Cada chip tem o seu próprio controlador de memória e as RAMs são ligadas a cada chip, logo o acesso é NUMA uma vez que o acesso a dados que estão na RAM ligada ao outro chip tem de

passar pelo canal de comunicação entre chips e competir com acessos à memória a partir de cores do outro chip

(ii) um único processo com threads nos cores dos 2 chips permite partilha de dados nas caches L3 nos 2 chips mas pode ser penalizado por comunicações entre chips e por remoção excessiva de linhas de cache; quando há vários processos independentes que não partilham dados o espaço da hierarquia de memória é dividida pelos vários processos e não há contenção em acessos à memória

2. [20]Um dos mais recente processadores 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).

Se todos os *cores* precisarem de aceder no mesmo ciclo de *clock* a valores *double-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/w requerida para ir buscar o 1º double: 64 cores * 64 Bytes (1 linha de cache) * 3GHz ≈ $4*10^3 * 3*10^9$ o que dá aproximadamente 12000 GB/sec

Para os seguintes 7 doubles: esses já estão em cache

B/w máx disponível: 8 channels * 25.6 GB/s ≈ 205 GB/sec

Logo, a b/w é insuficiente para ir buscar o 1º double mas se for preciso buscar os 7 seguintes que já estarão na cache a b/w já chega

Comentários:

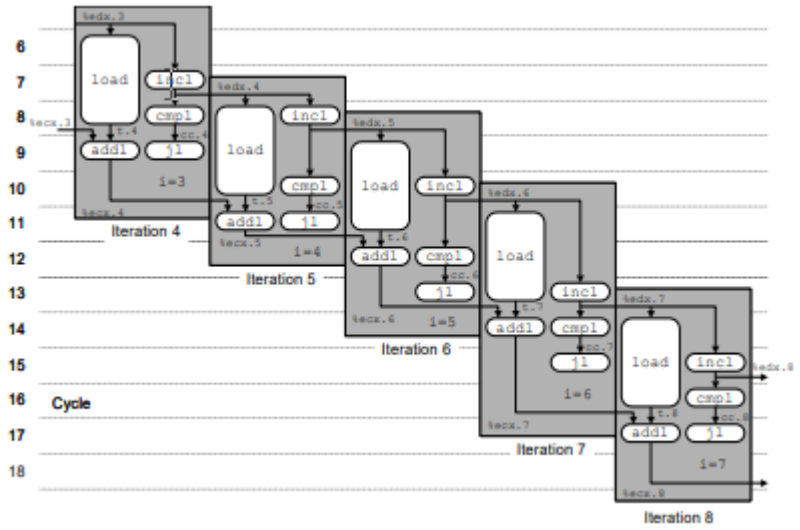
Uma memória DDR4-3200 tem capacidade de 3200 MegaTransferências/seg com um *clock* de 1600 MHz, e como cada transferência de/para a memória é de 64 bits (8 bytes), então consegue transferir 3200x8 B/s, que é o valor que foi indicado no enunciado, i.e., 25600 MB/s ou 25.6 GB/S. E isto é a capacidade de transferir dados, por cada canal de acesso à memória que o *chip multicore* tiver...

3. [30]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: (1) buscar uma instrução à memória (leva 3 ciclos de *clock*), (2) descodificá-la (1 ciclo de *clock*), (3) colocar o IP a apontar para a próxima instrução (1 ciclo de *clock*), (4) buscar os operandos à memória se necessário (3 ciclos de *clock*), (5) executar a operação descodificada (1 ou 4 ciclos de *clock*, consoante seja uma adição/salto ou multiplicação) e (6) guardar o resultado da operação, em registo (incluído no tempo da operação) ou na memória (3 ciclos de *clock*).

```
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;
}
```

A execução de cada iteração do ciclo `for` neste código (com 5 instruções máquina, conforme mostra a figura em baixo) e nesta arquitetura básica elementar, teria a duração estimada de 5 ciclos de *clock* (para os primeiros 3 passos em cada instrução, logo 25 ciclos em cada iteração), 3 ciclos de *clock* para ir buscar o elemento à memória e colocá-lo em registo e 1 ou 4 ciclos de *clock* para executar uma adição/salto ou multiplicação, dando um total de 32 ciclos de *clock* por iteração.

Contudo, numa dada arquitetura x-86, verifica-se o seguinte comportamento ao longo de 5 iterações:



a) [20]Para que tal comportamento seja possível, várias alterações tiveram que ser feitas nessa arquitetura básica elementar para melhorar consideravelmente o desempenho deste código em vetores grandes, nomeadamente, permitir que a execução das 5 instruções em cada iteração do ciclo `for`, cada com uma latência de 32 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.

1) Pipeline da unidade de acessos à mem (p/ buscar e descodificar instruções em avanço) suficientemente rápido para não atrasar a sua execução **e** da unidade de execução de instruções (cuja execução está visível na figura)
2) Hierarquia de memória para permitir que os dados necessários se encontrem na cache L1 para que LOAD possa ser executado em apenas 3 ciclos de clock
3) Superescalaridade para suportar o arranque de pelo menos 3 instruções em linguagem máquina em cada ciclo de clock
4) LOAD em pipeline com a possibilidade de arrancar um novo LOAD em ciclos consecutivos
5) Suporte à execução de instruções fora de ordem
6) Execução especulativa, permite o controlo da execução de saltos condicionais

Comentários:

A arquitetura básica elementar dum processador tem a ver apenas com o processador, tal como referido em **1. a)**, portanto sem qq referência à hierarquia de memória.

Se alguém tivesse dúvida sobre se esta arquitetura elementar tinha ou não *cache*, bastaria ler com cuidado o resto do enunciado, onde se dizia que a arquitetura elementar precisava de 3 ciclos de *clock* para ir buscar a instrução e outros 3 para is buscar dados, o que coincide com o tempo que necessita a arquitetura x-86 na figura (que precisava de 3 ciclos de *clock* para aceder a dados em memória com um LOAD), a qual já tinha os melhoramentos pedidos para executar uma instrução em média de 2 ciclos/elemento

b) [10]Os compiladores atuais são bastante eficientes para conseguirem analisar ciclos `for` e vetorizar o código que lá se encontra. Se este código for compilado com indicação para vetorizar, **indique**, justificadamente, se o compilador iria conseguir fazê-lo.

Para se conseguir vetorizar o ciclo for era preciso que as iterações nesse ciclo não dependessem umas das outras, o que não acontece neste caso: o cálculo da variável t em cada iteração está dependente do valor de t calculado na iteração anterior

4. a) [10]Duas vezes por ano é divulgada a lista TOP500 com a distribuição ordenada dos 500 sistemas de computação mais potentes a nível mundial. **Mostre** como é construída essa lista.

Esta lista é construída com base na potência máx estimada obtida em medições feitas em cada sistema com parte do sistema em funcionamento, em GFLOP/s, usando LINPACK Benchmarks, os quais resolvem um sistema denso de equações lineares com aritmética FP de precisão dupla

Comentários:

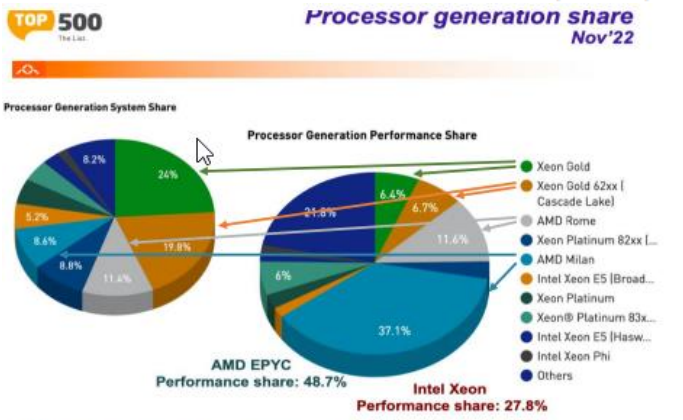
No *website* do TOP500 há várias listas ordenadas de sistemas de computação: a lista propriamente dita de TOP500, a lista Green500, a lista HPCG, a lista HPL-AI, ...

A questão colocada era apenas a lista TOP500 (LINPACK) (tal como estava no índice dos *slides*) e assim o que se pretendia era saber se tinham a noção de como era medida a *performance* dos sistemas (usando o Benchmark LINPACK e uma breve descrição do que era esta aplicação); e pretendia-se a *performance* dos sistemas e não apenas a dos processadores, como alguns disseram.

Vários alunos colocaram uma resposta igual, que não sei de onde foi retirada, mas que dei pouca cotação por não terem explicitado como era feita a medição da *performance*:

"Esta lista é constituída com base na execução de um determinado pacote de instruções segundo umas regras bem definidas e voluntariamente submetidas pelas entidades detentoras desses equipamentos."

b) [20]Considere a seguinte figura que mostra a distribuição das gerações de processadores, baseado em dados do TOP500 de nov-22, 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.



Estes 2 gráficos mostram essencialmente várias coisas:

- que a grande maioria dos 500 sistemas é baseado em arquiteturas x86-64, Intel e AMD (91.8%)
- que embora a maioria dos sistemas sejam baseados em processadores Xeon da Intel (71.8%) vs. apenas 20% com AMD EPYC, o somatório dos sistemas com AMD EPYC têm uma performance muito superior ao somatório dos sistemas com Xeon (48.7% vs. 27.8%), com destaque relevante para o Milan da AMD, em que 8.6% dos sistemas têm 37.1% da performance total
- que o conjunto dos restantes processadores (IBM, Sunway e Fujitsu/ARM) estão presentes em poucos sistemas (8.2%) mas que ainda representam algum peso na performance global (21.6%)
- que a performance da maioria destes sistemas tb foi conseguida à custa de aceleradores de computação, em especial os GPUs da NVidia e da AMD (exceções: Sunway e Fujitsu/ARM)

Comentários:

Os gráficos apenas mostravam explicitamente o papel dos processadores Intel Xeon e AMD EPYC, quer em termos da percentagem dos 500 sistemas que os contenham, quer a ainda a percentagem da *performance* global desses sistemas. Eu esperava comentários sobre isto.

E penalizei quem chamou de "*aceleradores*" aos processadores XEON e EPYC!!!

Adicionalmente:

- na lista de "*outros*" seria de referir o processador FUJITSU/ARM que esteve em 1º lugar até Jun'22, os POWER da IBM que ainda há vários que já estiveram no topo da lista, e o Sunway (o tal com 256+4 *cores*) que esteve recentemente em 1º lugar, os quais, embora representem poucos sistemas, tinham uma boa *performance*;

- na avaliação da percentagem da *performance* global, esta referia-se aos sistemas como um todo, quer tivessem aceleradores (normalmente GPUs), quer não, até porque normalmente a parte responsável pelo aumento da *performance* eram precisamente os GPUs...

FICHA 1

Exercise 1 - The basic processor performance equation

a) Performance estimation: What is the complexity of the MM (i.e., in big O notation, where N is the problem size and NxN is the matrix size)? What increase in execution time is expected when the N doubles? Which component of the performance equation is affected (e.g., #I, CPI or Tcc)?

b) Instructions estimation: Look at the MM assembly code that is generated (e. g., `gcc -O2 -S ...`) with and without optimisation (-O2 versus -O0). Can you estimate the number of instructions executed for a NxN matrix, on each case? What is the expected gain from the compiler optimised version?

c) Measure: The number of instructions executed (#I) and clock cycles (#CC) can be directly measured with `perf stat -e cycles,instructions ./a.out` (note: this takes overall metrics, not metrics by function as `perf record`, but it is enough for this case. Why?). Fill the following table for matrix sizes of 128x128, 256x256 and 512x512 using the -O2 optimisation level. Measure the time without optimisation just for a 512x512 matrix size. Also, fill the table by calculating the average CPI column for the 512x512 matrix size (see lecture notes for CPI definition).

	size	Texe	#CC	#I	#I Estimated	Average CPI (calculated)
-O2	128					
	256					
	512					
-O0	512					

d) Accuracy of the #I estimation: Compare the measured #I against the estimative from 1b). Comment the difference. Does it increases at the expected rate when the N value doubles?

e) #I vs CPI tradeoff: What is the gain obtained with the O2 optimisation level on a 512x512 matrix? What component of the performance equation is responsible for this gain? Why it is lower than expected (estimated in 1b)?

a)

There are three nested loops, each iterating N times. Therefore, the total number of iterations is $N \times N \times N = N^3$. The operation inside the innermost loop, `C[i][j] += A[i][k] * B[k][j]`, is constant time $O(1)$ (assuming basic operations). Thus, the overall time complexity of matrix multiplication is **O(N^3)**. When N doubles, the execution time increases by a factor of 8. The components affected are the instruction count (#I) and potentially Tcc (due to larger memory access), and CPI may also be affected depending on the system's architecture and memory access patterns.

b) **For 02:** Estimate of instructions in hotspot = 8 per cycle if n=512, then: 8 x 512 x 512 x 512 = **1073741824 intructions**. We should only look at the hotspot because the other instructions are executed at least 512(n) times less.
For 00: Estimate of instructions in hotspot = 45 per cycle if n=512, then: 45 x 512 x 512 x 512 = **6039797760 intructions**.
Gain: 45/8 = 5,625, looking only to the instructions.

c) O CPI é calculado ao multiplicar #CC x #I x Texe

d) The measured #I is significantly higher than the estimated #I, indicating the presence of additional instructions beyond the basic matrix multiplication operations. This could be due to optimizations, memory access overhead, and other factors. The increase in #I when N doubles does not follow the expected cubic rate and is influenced by compiler settings and possibly other system-specific factors.

e) The time gain was 1,126789 - 0.494284 = **0,632505s** (valores que estariam na tabela que foram calculados num pc)
Gain in percentage: (0,632505 / 1,126789) x 100 = **56,13%**
CPI is the one responsible because the O2 optimization can also reduce the number of cycles required per instruction.
The lower-than-expected gain can be attributed to diminishing returns from optimization as the matrix size increases, memory access bottlenecks, and the complexity of optimizing large matrix operations on a given CPU architecture.

Exercise 2 - Instruction level parallelism and its limits

a) Look again at the previous table. Why is the CPI less that one in some cases? What is the ideal CPI on this machine?

b) Explain why the CPI value is lower (i.e., better!) with the -O0 optimisation level (there are two important factors; one of them is in the answer to the next question).

c) Look at the assembly of the inner loop of the code generated with -O2 level. In this code identify the Read After Write (RAW) data dependencies. (note: your generated assembly code should be similar to the following assembly code for double data type: SD is single double):

```
.L12:
     movsd (%rdx), %xmm0  ;move SD from memory to register %xmm0
     mulsd (%rax), %xmm0  ;move SD from memory and multiply with %xmm0
     addq  $4096, %rax    ; add 4096 to %rax
     addq  $8, %rdx
     addsd %xmm0, %xmm1
     movsd %xmm1, (%rcx)
     cmpq  %rax, %rsi
     jne   .L12
```

d) Draw an instruction dependency graph (i.e., task dependency graph, each instruction is a task) where each box is an assembly instruction (e.g., primitive operation) and each arrow represents a data dependency (one of these instructions should be represented by two boxes, why?).

e) Look at the dependency graph, without further improvements, and perform a static instruction schedule on a processing unit with infinite resources and 1 cycle operations: memory load, add, multiplication, ..., What is the effective CPI of this code on this ideal machine (suggestion: focus on the longest dependency chain: the critical path).

f) (*) Suggest code improvements to increase the ILP available on this code and compute the potential gain of that code on an ideal machine by drawing a new dependency graph.

g) (*) Implement the optimised code and test it on small matrices (e.g., 128x128). Compare the performance with and without this optimisation.

a) The CPI is below 1 because the CPU can take advantage of the native parallelism. The ideal CPI is 0,25.

b) The lower CPI at the -O0 optimization level can be attributed to two main factors:

- The lack of complex optimizations (e.g., loop unrolling, inlining) at -O0, which can result in fewer instructions and a better alignment with the CPU's instruction pipeline.
- Simpler memory access patterns that align better with the CPU's cache, reducing the memory access time and improving the overall execution efficiency.

c) The second instruction (mulsd) reads %xmm0, which was written by the first instruction (movsd), creating a RAW dependency. The fifth instruction (movsd) depends on the value in %xmm1, which is written by the fourth instruction (addsd).

d)

```
[movsd (1)] ---->
                 \
[movsd (2)] ---->  [mulsd (2)]----> [addsd (5)] ----> [movsd (6)]

[addq (3)] ----> [cmpq (7)] ----> [jne (8)]

[addq (4)]
```

e) Longest path = 4 instructions => in this machine => 4 cycles. Total Instructions = 8. Effective CPI = 4 / 8 = 0,5

f) By reordering independent instructions, you can increase the ILP and potentially double the performance on an ideal machine. However, real-world performance gains will depend on various factors like CPU architecture, memory hierarchy, and whether SIMD instructions are used. The theoretical gain in an ideal scenario is a 2x speedup.

g) ->

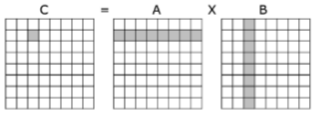
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 128

void matrix_multiply_opt(double A[N][N], double B[N][N], double C[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            // Reorder address updates before the main computation
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    double A[N][N], B[N][N], C[N][N];

    // Initialize matrices A and B
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }

    // Measure time for matrix multiplication with optimization
    clock_t start = clock();
    matrix_multiply_opt(A, B, C);
    clock_t end = clock();
    printf("Time with optimization: %lf seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    return 0;
}
```

FICHA 2:

Exercise 1 - Spatial Locality

```
foreach line of A
    foreach column of B
        Cline,column = DOTline of A,column of B
```

$$C_{ij} = DOT_{linha\_A_i, coluna\_B_j} = \sum_{k=0}^{n-1}(A_{ik} * B_{kj})$$



a) Data locality analysis: Look at the traditional matrix multiplication version (the DOT version, in the algorithm/figure above). For each memory access (C_{ij}, A_{ik} and B_{kj}) identify those that exhibit spatial locality (suggestion: start by looking at the DOT definition, eventually also look at the DOT assembly code in lab guide 1, exercise 2c).

b) Estimation of data locality impact: Estimate the number of level 1 cache misses for a NxN matrix, assuming that matrix size is greater than the L1 data cache size (suggestion: focus on the matrix accesses without spatial locality). Fill the table with the estimated values for this base version. How many L1 cache misses are expected per instruction executed (with `-O2` optimisation)?

c) Improvement of spatial locality: Design an optimised DOT implementation by transposing matrix A and/or B in order to get the best spatial locality. Estimate the number of level 1 cache misses of this "transp" version and fill the table with the estimated values (ignore the cost of transposing the matrix).

d) Measure and discuss the results: Measure the number of L1 data cache misses with `perf stat -e L1-dcache-load-misses -M cpi ./a.out`. Complete the table for matrix size of 512x512 (**always** using the `-O2` optimisation level) for the base and for the optimised (i.e., with the transpose) versions. What is the explanation for the performance differences between these two versions? Why is the gain not directly proportional to the improvement in cache misses?

N	Version	Time	CPI	#I (inst_retired.any)	L1_DMiss (estimated)	L1_DMiss	Miss/#I
512	base()						
	transp()						

e) (*) MM data locality analysis: Matrix multiplication is implemented with three nested loops. The algorithm has the particularity of producing correct results for any order of these loops. How many variations are possible by changing the order of the loops? What is the order that minimises the number of cache misses (i.e., with the best spatial locality), assuming a matrix larger than the L1 cache?

f) (*) Impact of matrix representation: Compare the MM "transp" implementation against an implementation using an array of pointers (also with the transpose). Take some metrics that give insights of the performance difference between versions.

a) An access to memory with spatial locality implies that the distance between data in memory units isnt big.

- A[i,k]: Exhibits strong spatial locality due to the addq $4096, %rax instruction, which accesses consecutive elements within a row.
- B[k,j]: Exhibits no spatial locality in this code snippet given row-major storage. The addq $8, %rdx instruction accesses elements separated by the row length of B.
- C[i,j]: No spatial locality within the inner loop. However, spatial locality will exist across calls to the dot product when calculating subsequent elements of the C matrix if it is stored in row-major order.

b) There are 3 cycles that travel the matrixes so:

- B always has misses: **n^3**
- A only has misses for each size of the read limit of each CPU block.

If we assume that the size of the machine block is 8 then, the number of misses in A is *(n^3)/8*.
Total is B + A = (8n^3)/8 + (n^3)/8 = **(9n^3)/8**.

c) The solution is transposing the B matrix and invert the acess order to the matrix itself. A x B_jk = A x B^T_k*j

d) The primary reason for the significant performance difference lies in how the two versions access memory, specifically matrix B.

N	Version	Time	CPI	#I	L1 Misses	Miss/#I
512	base()	0.5	0.913	110×10^7	142×10^6	0.129×10^6
512	trans()	0.177	0.507	837×10^6	17×10^6	0.014×10^6

Base Version: In the base version, when calculating C[i,j], the inner loop accesses elements of B along a column (B[k,j] for increasing k). If B is stored in row-major order (the standard in C/C++), this results in non-contiguous memory accesses. Each access to B[k,j] requires fetching a new cache line, leading to a high L1 cache miss rate. This is clearly shown by the high "L1 Misses" and "Miss/#I" values for the base version. **Transposed Version:** By transposing B, we effectively change its storage from row-major to columnmajor. Now, when calculating C[i,j], the inner loop accesses elements of the transposed B along a row. This results in contiguous memory accesses, exploiting spatial locality. The cache lines are utilized much more efficiently, leading to a drastically reduced L1 cache miss rate. This is reflected in the much lower "L1 Misses" and "Miss/#I" values for the transposed version.

The speedup isn't directly proportional to the reduction in cache misses because other factors contribute to execution time: increased instruction count in the transposed version, other cache levels (L2/L3), TLB misses, branch prediction, instruction-level parallelism, and the inherent limitations described by Amdahl's Law. These overheads limit the overall performance gain even with near-optimal cache utilization.

e) With three nested loops (traditionally i, j, and k), there are 3! (3 factorial) = 3 * 2 * 1 = 6 possible orderings: ijk, ikj, jik, jki, kij, kji
The kij order generally minimizes cache misses. In the kij order:

- The inner loop (j) iterates through consecutive elements of both C (row-wise) and B (row-wise), maximizing spatial locality for both matrices.
- The middle loop (i) iterates through rows of A and C.
- The outer loop (k) iterates through columns of A and rows of B.

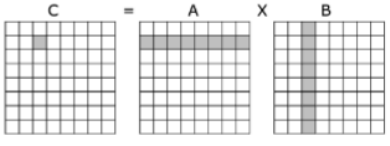
This order ensures that the most frequently accessed data (in the inner loop) is accessed contiguously in memory, leading to better cache utilization and fewer cache misses.

2.a) Assuming that the row of A fits in the cache, the misses for A are negligible. If the row of A does not fit in the cache, the number of misses for A stay the same. Therefore, we expect approximately a 4x reduction in cache misses related to accesses to B.

Exercise 2 - Temporal locality

a) Look at the base matrix multiplication code and identify matrix positions (only for matrix A and B) that are **reloaded** from memory in outer loops (note: this can also be identified looking at the DOT figure). Change the inner loop to take advantage of this temporal locality (e.g., to compute 4 dot products in the inner loop) and estimate the improvement in L1 cache misses for a large matrix size.

```
/* Cij =0 */
for i=0 to N-1
     for j=0 to N-1
          for k=0 to N-1
               C[i][j] += A[i][k] * B[k][j]
```



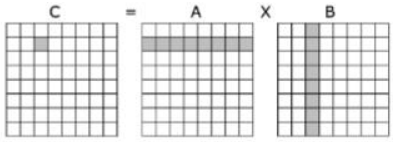

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j += 4) { // Increment j by 4
        for (int k = 0; k < N; k++) {
            C[i][j]   += A[i][k] * B[k][j];
            C[i][j+1] += A[i][k] * B[k][j+1];
            C[i][j+2] += A[i][k] * B[k][j+2];
            C[i][j+3] += A[i][k] * B[k][j+3];
        }
    }
}
```

FICHA 3

1.a) From the manually unrolled version: The innermost loop was unrolled by a factor of 4. This means that 4 iterations of the original loop are executed within one iteration of the unrolled loop. Evidence in the assembly In the unrolled version, 4 consecutive memory loads (vmovsd) and 4 fused multiply-add instructions (vfmadd231sd) occur before incrementing the loop index (add $4, %rax). The loop counter is incremented by 4 instead of 1, confirming the unrolling factor = 4. The estimated performance gain comes from the following benefits: Reduction in Loop Overhead: In the base version, the loop control instructions (add, cmp, jne) run every iteration. In the unrolled version, the loop control runs once every 4 iterations, reducing the overhead by approximately 75%. Instruction-Level Parallelism (ILP): Modern CPUs can execute multiple instructions simultaneously (superscalar execution). Unrolling exposes 4 floating-point operations (vfmadd231sd) within one loop iteration, allowing the CPU to process them in parallel. Estimated Gain: Theoretical performance improvement due to reduced loop overhead and increased ILP is typically 10–20%. Actual performance gain depends on: CPU architecture (e.g., pipeline, vectorization capabilities). Compiler optimizations (-O3 and -funroll-loops help further). Memory bandwidth and cache efficiency. For a conservative estimate, you can state: Performance gain is approximately 10–20%.

(*) Exercise 1 - Loop-Unrolling Review

```
/* Cij =0 */
for i=0 to N-1
      for j=0 to N-1
           for k=0 to N-1
                C[i][j] += A[i][k] * B[k][j]
```



a) (*) Estimation of the impact of loop unrolling: Loop unrolling can be enabled in `gcc` with the option `-funroll-loops`. Look at the assembly generated with this option and compare with the assembly of the base matrix multiplication (e.g., code in lab guide 1, exercise 2c). How many times was the loop unrolled? What is the estimated performance gain of this optimisation?

b) (*) Measure and discuss the results: Fill the table with performance data collected with `perf stat -e L1-dcache-load-misses -M cpi ./a.out`. Calculate the performance gain between *base* and *unroll* and comment the results.

b) Performance Gain = (Execution Time of Base - Execution Time of Unroll) / Execution Time of Base

2.a) **Non-contiguous memory access for matrix A:** In the innermost loop, the code accesses A[i][k] repeatedly for each value of k, which means the accesses to matrix A are not contiguous in memory. This can lead to inefficient memory access patterns because modern processors and compilers optimize for sequential access to contiguous memory blocks. The lack of locality in memory access for A[i][k] reduces the possibility of vectorization.

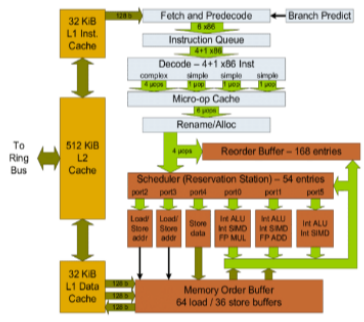
Exercise 2 - Vectorisation

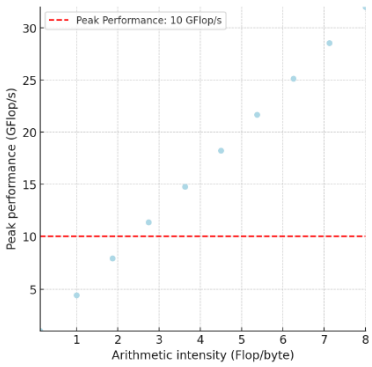
a) Identification of limitations: The compiler cannot vectorise the "base" DOT matrix multiplication code. Identify two reasons for this (suggestion: look at the matrix data access patterns in the DOT code, namely, reads from A; reads from B and writes to C).

b) Enabling vectorisation: The order of the loops can be changed without affecting the correctness of the results. Identify one (or more) order of the loops that support vectorisation. Implement the version that you selected and compile the code adding the options `-ftree-vectorize -msse4` (the second flag generates SSE4 vector instructions which supports vector operations of 128bits). How many elements are processed at once? Estimate potential gain of vectorisation by looking to the assembly with and without vectorisation (e.g., ratio of executed instructions).

Dependency on previous iteration's result for matrix C: The code performs a write operation on C[i][j] inside the innermost loop (C[i][j] += A[i][k] * B[k][j]). This operation involves reading and writing to the same location of the matrix C in every iteration. Since the write depends on the result of the previous iteration, this introduces a data dependency that hinders parallelization and vectorization. The compiler cannot safely execute multiple iterations of the loop in parallel because it cannot guarantee that the writes to C[i][j] won't interfere with each other across iterations.

Exercise 3 - Roofline Model





a) Peak Performance: Look at the figure above. How many operations can be performed at each cycle in this architecture? Identify, from the figure, constraints among operations that can be performed simultaneously. What is the floating point peak performance (with double data type), in GFlop per second, if the machine runs at 2.5 GHz and vector size is 256bits. How much time does it take to perform an 1024x1024 matrix multiplication at this GFlop/s rate?

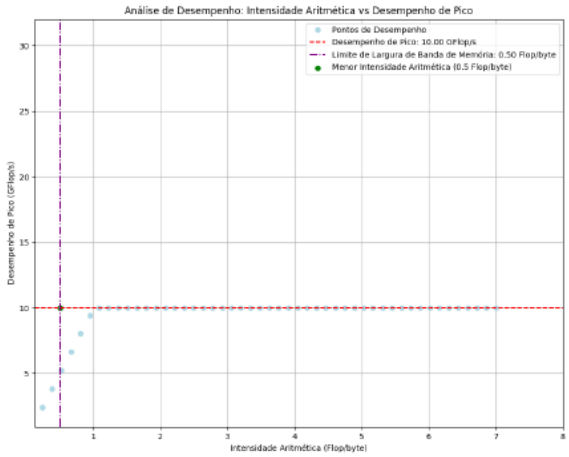
b) Complete the graph, at the right-side of the figure, with this GFlop/s in the Y-axis by plotting a horizontal line representing this limit.

c) Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes). Assuming a machine with a memory bandwidth 20 GB/s, what is the lowest arithmetic intensity supported by this machine without being constrained by the memory bandwidth (note: the required memory bandwidth for a given arithmetic intensity can be obtained by dividing the peak performance by the arithmetic intensity). Write this point in the X-axis.

d) Memory Bandwidth Limitation: What happens if the application requests more bytes per floating point operation than the available memory bandwidth? Plot a line in the graph showing this limit.

3.a) **Instruction Queue:** It handles the instructions and can process 4 instructions per cycle (noted as "4 i.p.c."). **Scheduler (Reservation Station):** It has 5 ports capable of issuing different instructions in parallel.

Memory Order Buffer: Handles load/store operations, with a buffer for 64 load and 36 store operations. **Reorder Buffer:** Can handle 168 entries for reordering instructions. At least 4 instructions can be fetched per cycle (Instruction Queue).
Int ALU operations (including SIMD) can occur on multiple ports, possibly performing up to 3 operations per cycle if all integer units and SIMD ports are fully utilized. Floating-point operations (e.g., FP MUL) can be processed at 1 per cycle, given that there are fewer dedicated ports for floating-point operations than for integer operations. **Frequency:** 2.5 GHz **Vector Size:** 256 bits **Double-precision floating-point (64-bit)** requires 2 double-precision values per 256-bit vector. **Per cycle:** 2 operations per cycle per SIMD unit, with up to 2 SIMD units for floating-point operations in parallel (ports 4 and 5). Therefore, the number of floating-point operations per cycle (for double precision) is: 2 operations per cycle from each of the two SIMD units, so **4 operations per cycle**. Given the clock speed of 2.5 GHz: **Peak GFlop/s** = 4 operations per cycle × 2.5 GHz = **10 GFlop/s**.
A 1024x1024 matrix multiplication typically involves 2×1024^3 (2 billion) floating-point operations so 2×1024^3 = **2,097,152,000 floating-point operations**.
Time = Total operations ÷ Peak performance = 2,097,152,000 operations ÷ 10,000,000,000 operations/sec = **0.2097 seconds.**
b) resolvida na imagem
c) Required memory bandwidth = Peak performance/Arithmetic intensity
Peak performance: 10 GFlop/s (from earlier) **Memory bandwidth:** 20 GB/s
Arithmetic intensity = Peak performance/Memory bandwidth Since the peak performance is in GFlop/s, and the memory bandwidth is in GB/s, we need to convert the peak performance into the same unit (Flop/byte):
10GFlop/s = 10^9 Flops/s, 10 GB/s = 10^9 bytes/s



Arithmetic intensity = (10^9 bytes/s / 20^9 bytes/s) = 0.5 Flop/byte
d) ← imagem
FICHA 4:

Exercise 1- The basic OpenMP construct: the parallel region

Copy & paste the following code to a new file, compile, and run the code 2 or 3 times. This code contains a small loop that displays the loop iteration number and the identification number of the thread executing each iteration. Note that each thread has a private variable *id* and the OpenMP function *omp_get_thread_num()* returns a different number for each calling thread.

```
#include<omp.h>
#include<stdio.h>

int main() {
    printf("master thread\n");
#pragma omp parallel num_threads(2)
    for(int i=0;i<100;i++) {
        int id = omp_get_thread_num();
        printf("T%d:i%d ", id, i);
    }
    printf("master thread\n");
}
```

a) Is the order of the output always the same across multiple runs? Why?
b) Is the order of the output **OF EACH** thread always the same? Why?
c) How is the loop execution distributed (i.e., scheduled) between threads?

1.a) Não, a ordem da saída não é sempre a mesma em várias execuções. Isso ocorre porque o código usa OpenMP para criar threads paralelas, e o agendamento dessas threads não é determinístico. O sistema operacional e o tempo de execução do OpenMP decidem como alocar o tempo de CPU para cada thread, o que pode variar de execução para execução. Como resultado, a intercalação da saída de diferentes threads pode mudar, levando a diferentes ordens de instruções impressas em cada execução.
b) Não, a ordem da saída de cada thread não é sempre a mesma. Isso ocorre porque o código usa OpenMP para criar threads paralelas, e o agendamento dessas threads é não determinístico. O sistema operacional e o runtime do OpenMP decidem como alocar o tempo de CPU para cada thread, o que pode variar de uma execução para outra. Como resultado, a intercalação da saída de diferentes threads pode mudar, levando a diferentes ordens de declarações impressas em cada execução.
c) Neste código, como nenhuma política de agendamento específica é definida, o OpenMP usa a política padrão, que geralmente é estática. No entanto, a ordem exata de execução das iterações ainda pode variar devido à natureza paralela e não determinística da execução das threads.

Exercise 2- Work sharing and synchronization

Successively introduce **one** of the following directives between the *#pragma omp parallel* and the "*for*" in code of exercise 1 and answer the following questions:

2.1. *#pragma omp for*

2.2. *#pragma omp master*

2.3. *#pragma omp single*

a) In 2.1/2.2/2.3, how is the loop execution distributed (i.e., scheduled) between threads?

b) In 2.1/2.2/2.3, the loop division is always the same?

2.4. *#pragma omp critical*

c) In 2.4, is the order of the output always the same? What kind of synchronization occurs?

2.a) 2.1. #pragma omp for: As distribuições do loop são feitas ao longo das threads. 2.2. #pragma omp master: O loop é executado apenas pela thread mestre. 2.3. #pragma omp single: O loop é executado apenas por uma thread, que pode ser qualquer thread na equipa.
b) 2.1. Não, a divisão do ciclo não é sempre a mesma. As iterações do ciclo são distribuídas entre as threads e a distribuição exacta pode variar em função da política de programação (estática, dinâmica, guiada) e do ambiente de execução. 2.2. Sim, a divisão do loop é sempre a mesma porque o loop é executado inteiramente pela thread mestre. Nenhuma outra thread participa da execução do loop. 2.3. Sim, a divisão do loop é sempre a mesma, no sentido de que apenas uma thread executa todo o laço. No entanto, a thread específica que executa o loop pode variar entre execuções.
c) Em 2.4, a ordem da saída não é sempre a mesma. A diretiva `#pragma omp critical` assegura que apenas uma thread de cada vez pode executar o bloco de código incluído, proporcionando exclusão mútua. Isso evita condições de corrida e garante que a saída de diferentes threads não seja intercalada. No entanto, a ordem em que as threads entram na secção crítica não é determinística, levando a variações na ordem de saída em diferentes execuções.

Exercise 3- Synchronization

3.1. Include a barrier inside the loop, after the *printf* statement *(#pragma omp barrier)*.

a) Is the order of the output always the same? What kind of synchronization occurs?

3.2. Include the directive *#pragma omp ordered* inside the loop, before the *printf* statement. Use the program developed in 2.1, also adding *ordered* to the *#pragma omp for*

b) The order of the output is always the same? What kind of synchronization occurs?

3.a) Não, a ordem da saída não é sempre a mesma. A diretiva `#pragma omp barrier` garante que todas as threads esperem na barreira até que todas as threads a tenham alcançado. Este ponto de sincronização garante que todas as threads completem cada iteração do loop antes que qualquer thread prossiga para a próxima iteração.
b) Não, a ordem da saída não é sempre a mesma. A diretiva `#pragma omp for ordered` garante que as iterações do loop sejam executadas na ordem especificada pelo índice do loop. No entanto, a ordem real de saída ainda pode variar porque a cláusula `ordered` garante apenas a ordem de execução dentro do laço, não a ordem das instruções de saída. A sincronização que ocorre é que cada thread espera sua vez de executar sua iteração na ordem correta, mas a impressão ainda pode ser intercalada.

Exercise 4 - Loop scheduling

Exploit the impact of the following of the loop scheduling options in program 2.1, by adding the *schedule* clause to the *for* directive:

4.1. *schedule(static)* and *schedule(static,10)*

4.2. *schedule(dynamic)* and *schedule(dynamic,10)*

4.3. *schedule(guided)*

a) In 4.1/4.2/4.3, how is the loop execution distributed (i.e., scheduled) between threads?

b) In 4.1/4.2/4.3, is the loop distribution (i.e., scheduling) always the same?

4.a) schedule(static): Divisão igual das iterações entre as threads. schedule(static,10): Partes de tamanho fixo de 10 iterações distribuídas de forma round-robin. schedule(dynamic): Atribuição dinâmica de iterações a threads. schedule(dynamic,10): Atribuição dinâmica de partes de tamanho fixo de 10 iterações. schedule(guided): Diminuição dos tamanhos dos pedaços atribuídos dinamicamente para equilibrar a carga e reduzir a sobrecarga.
b)

`schedule(static)` e `schedule(static,10)`

- `schedule(static)`: Sim, a distribuição do loop é sempre a mesma. As iterações são divididas igualmente entre as threads antes da execução.
- `schedule(static,10)`: Sim, a distribuição do loop é sempre a mesma. As iterações são divididas em pedaços de tamanho fixo de 10 e distribuídas de forma round-robin.

`schedule(dynamic)` e `schedule(dynamic,10)`

- `schedule(dynamic)`: Não, a distribuição do loop não é sempre a mesma. As iterações são atribuídas às threads dinamicamente à medida que elas completam seu trabalho atual.
- `schedule(dynamic,10)`: Não, a distribuição do loop não é sempre a mesma. As iterações são divididas em blocos de 10 e atribuídas a threads dinamicamente.

`schedule(guided)`

- `schedule(guided)`: Não, a distribuição do loop não é sempre a mesma. As iterações são divididas em partes que diminuem de tamanho ao longo do tempo e são atribuídas dinamicamente.

FICHA 5

Exercise 1 - Data sharing with OpenMP

Copy & paste the following code to a new file, compile and run it 2/3 times. The code contains a small loop that displays the current iteration number and the identification of the corresponding thread executing it, as well as a counter `w`, shared among all threads.

```
#include<omp.h>
#include<stdio.h>

int main() {
    int w=10;
#pragma omp parallel
#pragma omp for
    for(int i=0;i<100;i++) {
        int id = omp_get_thread_num();
        printf("T%d:i%d w=%d\n", id, i, w++);
    }
    printf("w=%d\n", w);
}
```

Test the code as is (version 1.0) and then add & test the following clauses to the `#pragma omp for` directive:

1.1. `private(w)`
1.2. `firstprivate(w)`
1.3. `lastprivate(w)`
1.4. `reduction(+:w)`

a) How does the initial value of `w` vary with the use of the 1.0-1.4 directives?

b) Does the final value of `w` inside the loop vary with the use of the 1.0-1.4 directives?

c) Is the final value of `w` after the loop execution what would be expected?

1.a) 1.0: w is shared, leading to race conditions and unpredictable values. 1.1: w is private, initialized to an undefined value (often 0). 1.2: w is private, initialized to the original value (10). 1.3: w is private, initialized to an undefined value (often 0), with the last value copied back. 1.4: w is private, initialized to 0, with values combined using addition.
b) 1.0: w is shared, leading to race conditions and unpredictable values. 1.1: w is private, initialized to an undefined value (often 0), but retains the value from the shared variable. 1.2: w is private, initialized to the original value (10), leading to a final value of 49 inside the loop. 1.3: w is private, initialized to the original value (10), with the last value copied back, leading to a final value of 49 inside the loop. 1.4: w is private, initialized to 0, with values combined using addition, leading to a final value of 49 inside the loop.
c) 1.0: w=110 - Expected, as w is shared and incremented by all threads. 1.1: w=10 - Expected, as w is private and the main thread's w remains unchanged. 1.2: w=10 - Expected, as w is firstprivate and the main thread's w remains unchanged. 1.3: w=50 - Expected, as w is last private and the value from the last iteration is copied back. 1.4: w=110 - Expected, as w is reduced and the increments are combined.

Exercise 2 - Data races in OpenMP

Consider the following code that computes the dot product of two vectors:

```
#include<omp.h>
#include<stdio.h>
#define size 100000
double a[size], b[size];
int main() {
    // init vectors
    for(int i=0;i<size; i++) {
            a[i] = 1.0/((double) (size-i));
            b[i] = a[i] * a[i];
    }
    // compute dot product
    double dot = 0;
    for(int i=0;i<size; i++) {
        dot += a[i]*b[i];
    }
    printf("Dot is %18.16f\n",dot);
}
```

Parallelise the execution of the dot product (2nd `for-loop`) with the directive `#pragma omp parallel for`.

a) Does the result of the dot product differ from run to run? If so, why?

b) Is the result of the dot product affected by using different amounts of threads?
Suggestion: run the code with 2, 4, and 8 threads.

c) Adapt the parallelisation to produce the correct results using any of the OpenMP directives studied in the previous lab session (lab 5).
Validate the correctness of the implementation by testing with different number of threads.

d) Is it possible to achieve a better correct parallelisation using different directive(s)?

2.c) To ensure the correctness of the parallel reduction and avoid race conditions, you can use the #pragma omp critical directive to protect the update of the dot variable.
d) Yes, it is possible to achieve better and more efficient parallelization using the reduction clause in OpenMP. The reduction clause is specifically designed for parallel reduction operations and avoids the overhead of the critical section, leading to better performance.

(*) Exercise 3 – Parallelisation scalability

Consider the following code that computes an approximation of the value of *pi*:

```
double f( double a ) {
    return (4.0 / (1.0 + a*a));
}
double pi = 3.14159265358979323846264643;
int main() {
    double mypi = 0;
    int n = 1000000000; // number of points to compute
    float h = 1.0 / n;
    for(int i=0; i<n; i++) {
        mypi = mypi + f(i*h);
    }
    mypi = mypi * h;
    printf(" pi = %.10f \n", mypi);
}
```

Parallelise the code with OpenMP, using adequate directives to tackle potential data races. Measure the performance of the code for 2, 4, 8, and 16 threads, and calculate the speedup relative to its sequential execution. Does the performance improve as expected?

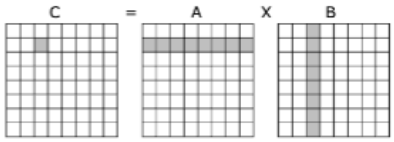
3. The performance improves when increasing the number of threads from 2 to 8. However, increasing the number of threads to 16 does not yield further improvement and actually results in a slight performance degradation.
Possible Reasons: *Overhead:* The overhead of managing more threads might outweigh the benefits. *CPU Utilization:* The system might not have enough physical cores to efficiently handle 16 threads, leading to context switching and other overheads. *Memory Bandwidth:* The memory bandwidth might become a bottleneck with more threads.

FICHA 6 – sem resolução, tudo culpa do cluster
FICHA 7 – apenas edição de código
FICHA 8

Exercise 1 – Distributed memory matrix multiplication with MPI collective operations

Download the matrix multiplication code from lab session 0 (base DOT version in the figure).

$$C_{ij} = DOT_{linha\_A_i, coluna\_B_j} = \sum_{k=0}^{n-1}(A_{ik} * B_{kj})$$




```
foreach line of A
      foreach column of B
        C[line,column] = DOT[line of A,column of B]
```

a) Identify a strategy to implement the matrix multiplication with MPI:

1. Each process will compute a subset of the elements of matrix C. Identify the main alternatives to partition the computation of this matrix among processes.
2. Select the simplest alternative to partition the computations of C. What data from matrix A and matrix B is required to perform the computation of each part of C?

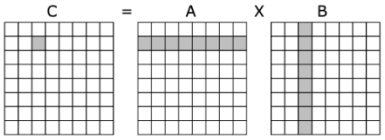
1.a) 1. The core idea is to divide the computation of the resulting matrix C among the available processes. Here are the main partitioning strategies:
Row-wise partitioning: Each process is responsible for calculating a subset of rows of matrix C. This is a common and often efficient approach.
Column-wise partitioning: Each process calculates a subset of columns of matrix C. This is similar in concept to row-wise partitioning.
Block-wise partitioning: The matrix C is divided into smaller square or rectangular blocks, and each process is responsible for calculating one or more of these blocks. This offers greater flexibility and can be more efficient for larger numbers of processes, but it also introduces more complexity in terms of data distribution and gathering.
2. The simplest partitioning strategy is row-wise partitioning. Let's break down the data requirements:
Distribution of C: If we have P processes, each process p (where 0 <= p < P) will compute size/P rows of C (assuming size is divisible by P). The rows assigned to process p will be from row p * (size/P) to (p + 1) * (size/P) - 1.
Data needed from A: To compute its assigned rows of C, each process p needs all rows of matrix A. Therefore, we need to distribute the entire matrix A to all processes. This implies that each process will store a copy of the entire matrix A.
Data needed from B: To compute its assigned rows of C, each process p needs all columns of matrix B. Therefore, we also need to distribute the entire matrix B to each process. This implies that each process will store a copy of the entire matrix B.

FICHA 9 – só alterações em código
FICHA 10

(*) Exercise 2 – Matrix multiplication (MM) on GPU

Consider the DOT version of the MM code from lab session 1.

```
/* Cij =0 */
for i=0 to N-1
  for j=0 to N-1
    for k=0 to N-1
       C[i][j] += A[i][k] * B[k][j]
```



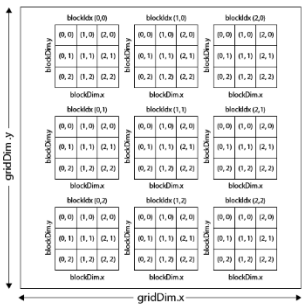
a) Develop a (simple) CUDA based implementation of this code. In this implementation each thread computes an element of the resulting C matrix. A 2D thread structure can be specified by (using a 2D grid of thread blocks of 16x16):

```
dim3 threadsPerBlock (16, 16);   // block of 16x16 threads
dim3 blocksPerGrid (N/16, N/16); // grid of N/16xN/16 blocks
mmKernel<<< blocksPerGrid, threadsPerBlock >>> (da, db, dc);
```

With this thread organization the MM kernel becomes (almost) trivial:

```
__global__
void mmKernel (float *a, float *b, float *c) {
   int i = blockIdx.y*blockDim.y+threadIdx.y;
   int j = blockIdx.x*blockDim.x+threadIdx.x;
     ...
   c[i*N+j] = ...
}
```

CUDA Grid



b) Change the previous implementation to use thread blocks of size 256x1. The MM performance improves with this thread organization? Why?

2.b) For large matrix multiplications, this layout (256 × 1) can lead to better performance compared to 16×16, primarily due to enhanced memory coalescing and reduced warp divergence. However, for small or moderately sized matrices, the performance difference might be negligible. To maximize performance, additional optimizations (e.g., shared memory tiling, loop unrolling) should be employed.