Capítulo 1

Intel Modern Code: Programação Vetorial e Paralela em Arquiteturas Intel Xeon e Xeon Phi

Matheus S. Serpa – msserpa@inf.ufrgs.br[1]
Jean L. Bez – jean.bez@inf.ufrgs.br[2]
Eduardo H. M. Cruz – ehmcruz@inf.ufrgs.br[3]
Matthias Diener – mdiener@inf.ufrgs.br[4]
Marco A. Z. Alves – mazalves@inf.ufpr.br[5]
Philippe O. A. Navaux – navaux@inf.ufrgs.br[6]

[1]Matheus da Silva Serpa é atualmente bolsista da Intel Corporation e Mestrando no Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (UFRGS). É bacharel em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA), tendo sido bolsista de iniciação científica da FAPERGS por três anos, e recebido o Prêmio Destaques UNIPAMPA 2015 e o Prêmio Aluno Destaque da SBC (2016). Suas principais áreas de pesquisa são Arquitetura de Computadores e Computação de Alto Desempenho.

[2]Jean Luca Bez é atualmente bolsista do projeto High Performance for Energy (HPC4e) e Mestrando no Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (UFRGS). É bacharel em Ciência da Computação pela Universidade Regional Integrada do Alto Uruguai e das Missões - Erechim (URI), tendo sido bolsista de extensão durante 3 anos, e recebido o Prêmio Destaque Extensão (2012 e 2013) e o Prêmio Aluno Destaque da SBC (2015). Suas principais áreas de pesquisa são I/O Parallelo e Computação de Alto Desempenho.

[3]Eduardo Henrique Molina da Cruz se graduou em Ciência da Computação na Universidade Estadual de Maringa (UEM) e concluiu seu mestrado e doutorado na Universidade Federal do Rio Grande do Sul (UFRGS). Atualmente é pós-doutorando na UFRGS. Sua linha de pesquisa envolve arquitetura de computadores e sistemas operacionais e foca na otimização do acesso a memória em máquinas paralelas.

[4]Matthias Diener se graduou em Engenharia da Computação no Instituto Tecnológico de Berlin (TU Berlin) e concluiu seu mestrado e doutorado na Universidade Federal do Rio Grande do Sul (UFRGS). Atualmente é pós-doutorando na UFRGS. Sua linha de pesquisa foca em melhorar o desempenho e a eficiência energética em arquiteturas paralelas de memória compartilhada.

[5]Marco Antonio Zanata Alves se graduou em Ciência da Computação na Universidade Estadual de São Paulo (UNESP) e concluiu seu mestrado e doutorado na Universidade Federal do Rio Grande do Sul (UFRGS). Atualmente é Professor Adjunto na Universidade Federal do Paraná (UFPR). Sua linha de pesquisa envolve a eficiência energética de memórias cache em arquiteturas de alto desempenho.

[6]Philippe Olivier Alexandre Navaux é Professor Titular na Universidade Federal do Rio Grande do Sul (UFRGS) desde 1973. Se graduou em Engenharia Elétrica na UFRGS em 1970. Recebeu seu mestrado pela UFRGS em 1973 e seu doutorado pela Instituto de Tecnologia de Grenoble (INPG), na França, em 1979. Ele é o coordenador do Grupo de Processamento Paralelo e Distribuído (GPPD) na UFRGS e consultor de várias entidades de fomento nacionais e internacionais como DoE (US), ANR (FR), CNPq (BR), CAPES (BR), entre outras.

Resumo

Tradicionalmente o aumento de desempenho das aplicações se dava de forma transparente aos programadores devido ao aumento do paralelismo a nível de instruções e aumento de frequência dos processadores. Entretanto, este modelo não se sustenta mais. Atualmente para se ganhar desempenho nas arquiteturas modernas, é necessário conhecimentos sobre programação paralela e programação vetorial. Ambos paradigmas são tratados de forma lateral em cursos de computação, sendo que muitas vezes nem são abordados. Neste contexto, este minicurso objetiva propiciar um maior entendimento sobre os paradigmas de programação paralela e vetorial, de forma que os participantes aprendam a otimizar adequadamente suas aplicações para arquiteturas modernas. Como plataforma de desenvolvimento, serão utilizados processadores Intel Xeon e Intel Xeon Phi.

1.1. Programação em OpenMP

Open Multi-Processing (OpenMP) consiste em um padrão de programação paralela para arquiteturas de memória compartilhada [Chapman et al. 2008]. OpenMP utiliza a diretiva *#pragma*, definida no padrão da linguagem C/C++. Nesta seção, detalharemos todas as construções básicas definidas pelo OpenMP.

1.1.1. Inclusão das funções da biblioteca

O padrão OpenMP, além das diretivas interpretadas diretamente pelo compilador, define uma série de funções através da biblioteca `omp.h`, que pode ser incluída através do seguinte código:

```
1 #ifdef _OPENMP
2 #include <omp.h>
3 #endif
```

A macro `_OPENMP` é utilizada para identificar se o compilador suporta o OpenMP. Desta forma, a biblioteca `omp.h` só é incluída caso haja suporte. As principais funções da biblioteca são:

int omp_get_num_threads() Retorna o número de *threads* ativas naquele momento da execução.

int omp_get_thread_num() Retorna o identificador da *thread*, também conhecido como *id*.

1.1.2. Iniciando um bloco de execução paralela

Para iniciar um bloco de execução paralela, o seguinte código deve ser utilizado:

```
1 #pragma omp parallel
2 {
3 }
```

O ambiente OpenMP irá alocar um determinado número de *threads*, e todas elas executarão as linhas de comando contidas entre as chaves . O número de *threads* varia, sendo responsabilidade do programador garantir que o resultado esperado seja atingido independente do número de *threads*.

A diretiva permite também compartilhar ou replicar variáveis conforme o código a seguir:

```
1 #pragma omp parallel shared(variables) private(variables)
2 {
3 }
```

A construção `shared` define que as variáveis são compartilhadas. A construção `private` define que as variáveis são privadas e que portanto devem ser replicadas na memória. Por padrão, as variáveis são do tipo `shared`.

Para exemplificar o que foi ensinado até agora, considere o seguinte código:

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main (int argc, char **argv)
5 {
6       int myid, nthreads;
7       #pragma omp parallel private(myid) shared(nthreads)
8       {
9             myid = omp_get_thread_num();
10            printf("myid: %i\n", myid);
11            if (myid == 0)
12                  nthreads = omp_get_num_threads();
13      }
14      printf("Havia %i threads na regiÃ£o paralela\n",
           nthreads);
15      return 0;
16 }
```

Nomeamos o arquivo como `hello.c`, e para compilar com o compilador Intel utilizamos a seguinte linha de comando:

```
1 icc -o hello hello.c -fopenmp
```

Para solicitar o número de *threads* à biblioteca, pode ser definida a variável de ambiente `OMP_NUM_THREADS` como exemplifica o código a seguir:

```
1 export OMP_NUM_THREADS=4
```

Neste exemplo, é solicitado que o ambiente de execução do OpenMP crie 4 *threads*. Ao executar este código, a seguinte saída é retornada no terminal:

```
1 ./hello
2 myid: 0
3 myid: 3
4 myid: 2
5 myid: 1
6 Havia 4 threads na região paralela
```

1.1.3. Sincronizando as threads

No exemplo anterior, utilizamos um comando condicional de forma explícita para selecionar que apenas a *thread* 0 atualizasse o conteúdo da variável *nthreads*. O OpenMP prevê a seguinte construção para realizar isto:

```
1 #pragma omp master
2 {
3 }
```

Tudo que estiver entre as chaves será executado apenas pela *thread* 0. Caso não seja necessário que a *thread* 0 que execute, mas apenas uma das *threads*, pode ser utilizada a seguinte construção:

```
1 #pragma omp single
2 {
3 }
```

Outra diferença entre o `single` e o `master` é que o `single` adiciona uma barreira implícita após seu término. Isto é, apesar de apenas uma *thread* executar o bloco `single`, todas as outras *threads* ficam aguardando a execução finalizar para prosseguir. Caso não seja necessária a barreira, deve-se adicionar a diretiva `nowait` ao comando:

```
1 #pragma omp single nowait
2 {
3 }
```

Barreiras podem ser incluídas de forma explícita com a seguinte linha de comando:

```
1 #pragma omp barrier
```

Como OpenMP foca em ambientes de memória compartilhada, muitas vezes é necessário acessar e atualizar variáveis compartilhadas. Por exemplo, considere um algoritmo que soma todos os elementos de um vetor. O seguinte código foi elaborado contendo as diretivas já apresentadas:

```
1 int sum(int *v, int n)
2 {
3       int i, sum, nthreads, id;
4       sum = 0;
5       #pragma omp parallel private(i, id)
6       {
7             #pragma omp single
8             nthreads = omp_get_num_threads();
9             id = omp_get_thread_num();
10            for (i=id; i<n; i+=nthreads)
11                  sum += v[i];
12      }
13      return sum;
14 }
```

Entretanto, este código não funciona como esperado devido às *condições de corrida*. Não há uma sincronização adequada entre as *threads*, fazendo que as sucessivas operações de leitura e escrita se sobreponham. A Tabela 1.1 demonstra um exemplo de tais sobreposições, considerando que há 2 *threads* e que todos os elementos do vetor são iguais a 1. Entre os tempos 1 e 4, tudo ocorre como esperado. Porém, nos tempos 5 e 6, as operações de ambas as *threads* se sobrepõe, fazendo literalmente que uma das operações de soma seja perdida, causando um resultado errado.

O OpenMP provê mecanismos para controlar acessos a tais regiões críticas, que acessam dados compartilhados. A primeira diretiva é a `critical`:

```
1 #pragma omp critical
2 {
3 }
```

Todo conteúdo colocado entre as chaves {} ocorre de forma atômica. Se alguma *thread* tenta entrar em uma região crítica enquanto outra *thread* já se encontra na região crítica, esta *thread* fica bloqueada aguardando a outra sair da região crítica. Caso haja diferentes recursos compartilhados, para que não haja interferência nas regiões críticas, é

Tabela 1.1: Exemplo de execução do código de soma dos elementos de um vetor com o problema das condições de corrida.

Tempo	Thread 0	Thread 1
1	Ler sum=0	
2	Escrever sum=1	
3		Ler sum=1
4		Escrever sum=2
5	Ler sum=2	Ler sum=2
6	Escrever sum=3	Escrever sum=3

possível estabelecer um identificador de regiões críticas:

```
1 #pragma omp critical(id)
2 {
3 }
```

O principal problema das regiões críticas é seu alto custo, pois faz com que as *threads* bloqueiem. Uma alternativa, para operações simples de lógica e aritmética, é o uso do `atomic`:

```
1 #pragma omp atomic
2 var1 += var2;
```

Neste exemplo, a operação de soma ocorre atomicamente. O `atomic` faz uso de suporte de *hardware* para realizar a operação de forma atômica, sendo mais eficiente que o `critical`.

O `critical` e `atomic` controlam o acesso à regiões críticas. Uma outra forma de sincronizar as *threads* é distribuir a carga de trabalho entre as as mesmas. Um dos principais focos do OpenMP é a paralelização de laços que não possuem dependências entre suas iterações. Anteriormente, isto foi feito de forma manual utilizando-se o número de *threads* e seus idenficadores. Isso pode ser feito de forma automática:

```
1 #pragma omp for
2 for (i=0; i<n; i++)
```

Em nosso exemplo de soma dos elementos de um vetor, o OpenMP provê a possibilidade de redução de um laço através de uma operação:

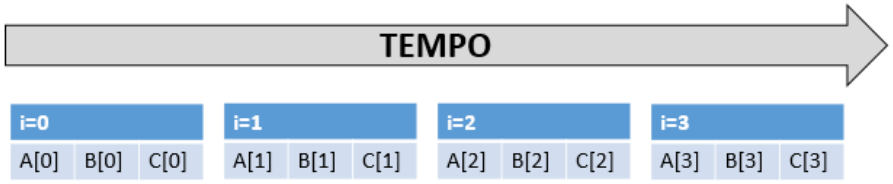
```
1 int sum(int *v, int n)
2 {
3       int i, sum;
4       sum = 0;
5       #pragma omp parallel for reduction(+:sum)
6       for (i=0; i<n; i++)
7             sum += v[i];
8       return sum;
9 }
```

1.2. Programação Paralela Vetorial (Intel AVX)

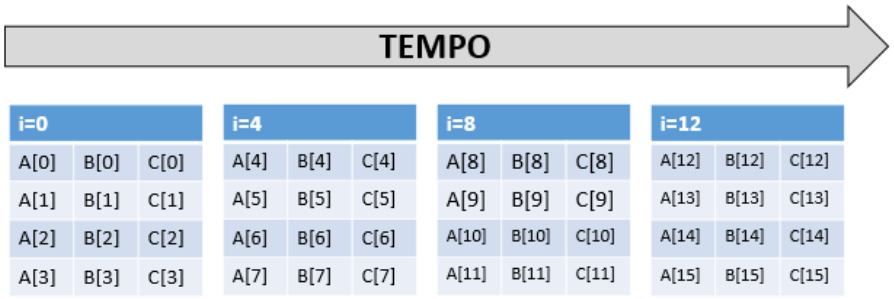
O paralelismo com execução vetorial se dá de forma diferente do explicado anteriormente. Enquanto na execução normal cada instrução opera em apenas um dado, na instrução vetorial a mesma operação é executada em vários dados de forma independente [Satish et al. 2012]. Considere o seguinte laço, que soma dois vetores e põe o resultado em um terceiro laço:

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

Como pode-se perceber, as iterações do laço são independentes. Supondo que haja instruções para ler e escrever 4 operandos na memória, e somar 4 operandos, pode-se visualizar o mesmo laço sendo operado vetorialmente da seguinte maneira (supondo *n*



(a) Execução normal.



(b) Execução vetorial com 4 elementos processados simultaneamente.

Figura 1.1: Diferença entre a execução normal de um laço e sua execução vetorial.

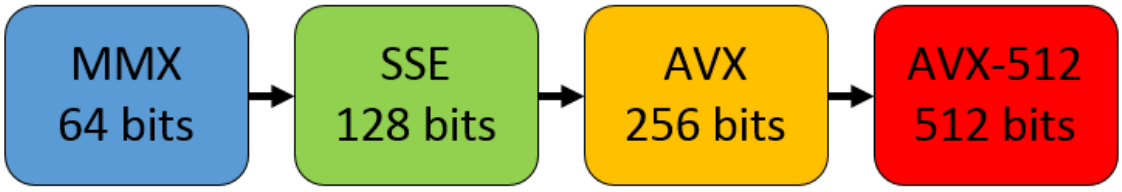


Figura 1.2: Evolução das instruções vetoriais na arquitetura x86.

múltiplos de 4):

```
1 for (i=0; i<n i+=4)
2         a[i:4] = b[i:4] + c[i:4]
```

Em cada iteração do laço, carrega-se 4 operandos a partir da posição i dos vetores b e c, soma-se cada par $(b[i], c[i])$ de forma independente, e depois o bloco de 4 operandos é escrito no vetor a a partir da posição i. Este comportamento pode ser visualizado na Figura 1.1.

As instruções vetoriais já estão presentes há muitos anos nos processadores x86. A Figura 1.2 contém a evolução das instruções vetoriais x86. A cada etapa da evolução, aumenta-se a quantidade de dados processados por instrução, bem como o número de instruções vetoriais disponíveis. Neste documento, o foco são as instruções AVX. É importante ressaltar que, para maior eficiência, **os endereços acessados no laço em iterações sucessivas devem ser consecutivos**.

A fim de se entender como funciona, considere um código que realiza a soma da multiplicação dos elementos de um vetor:

```
1 double vsum (double *a, double *b, int n)
2 {
3       int i;
4       double r = 0;
5       for (i=0; i<n; i++)
6             r += a[i] * b[i];
7       return r;
8 }
```

Nas subseções a seguir, veremos algumas possibilidades de implementação.

1.2.1. Intrínsecas

A primeira implementação AVX a ser demonstrada é a manual, utilizando as intrínsecas do compilador da Intel. Para fins de simplificação, consideramos que o número de elementos nos vetores seja 4:

```
1 #include <immintrin.h>
2
3 double vsum(double *a, double *b, int n)
4 {
5       int i;
6       double r, partial[4];
7       __m256d ac, va, vb, mul;
8       ac = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
9       for (i=0; i<n; i+=4) {
10            va = _mm256_load_pd(&a[i]);
11            vb = _mm256_load_pd(&b[i]);
12            mul = _mm256_mul_pd(va, vb);
13            ac = _mm256_add_pd(ac, mul);
14      }
15      _mm256_storeu_pd(partial, ac);
16      r = partial[0] + partial[1] + partial[2] + partial
          [3];
17      return r;
18 }
```

A biblioteca `immintrin.h` inclui os tipos de dados e funções necessários. A descrição dos tipos e funções utilizados:

__m256d 4 pontos flutuantes to tipo `double` empacotados em uma variável de 256 bits.

__m256d _mm256_set_pd (double e3, double e2, double e1, double e0) Inicializa uma variável de 256 bits com 4 `doubles`.

__m256d _mm256_load_pd (double const * mem_addr) Carrega 32 bytes de dados a partir do endereço `mem_addr` e coloca na variável do tipo `__m256d`.

__m256d _mm256_mul_pd (__m256d a, __m256d b) Multiplica de forma independente os 4 pares de ponto flutuante contidos nas variáveis `a` e `b`.

__m256d _mm256_add_pd (__m256d a, __m256d b) Soma de forma independente os 4 pares de ponto flutuante contidos nas variáveis `a` e `b`.

void _mm256_storeu_pd (double * mem_addr, __m256d a) Salva os 32 bytes de dados da variável `a` a partir do endereço `mem_addr`.

A ideia desta implementação é, em cada iteração do laço, explicitamente processar 4 elementos do vetor. Caso não tivéssemos considerado o número de elementos no vetor (n) um múltiplo de 4, seria necessário processar o vetor vetorialmente até um valor k, tal que $k < n$ e k é múltiplo de 4. Para os elementos x, tal que $k < x < n$, deveria ser processado da forma tradicional, apenas 1 elemento por iteração.

Entretanto, a necessidade de conhecer as diretivas de baixo nível e ter que explicitamente codificar o código AVX desencoraja este tipo de programação. Além disso, caso seja alterado o tipo de dados de `double` para `float`, ou se o conjunto de instruções evoluir e suportar mais operandos por variáveis SIMD, é necessário recodificar o código. Nenhuma destas situações são desejáveis.

1.2.2. PRAGMA SIMD

Para que a situação anterior não seja necessária, foi introduzido o seguinte `pragma`:

```
1 #pragma simd
```

A lógica deste comando é semelhante ao `pragma omp for`, com a diferença que agora o paralelismo se dá vetorialmente. Ele também aceita a cláusula `reduction`. É responsabilidade do programador assegurar que as iterações são independentes.

O código anterior usando este `pragma` fica da seguinte maneira:

```
1 double vsum (double *a, double *b, int n)
2 {
3       int i;
4       double r = 0;
5       #pragma simd reduction(+:r)
6       for (i=0; i<n; i++) {
7             r += a[i] * b[i];
8       }
9       return r;
10 }
```

O `pragma` automaticamente considera o caso de n não ser múltiplo de 4.

1.2.3. PRAGMA SIMD com alinhamento de memória

Um dos principais pontos sobre vetorização é o alinhamento de memória. Alinhar memória significa fazer com que um determinado endereço de memória seja múltiplo de um determinado valor. Com o alinhamento correto, o *hardware* pode otimizar o acesso à memória [Lee et al. 2010]. Por exemplo, uma variável alinhada ao tamanho do seu tipo (exemplo um `double` alinhado em 8 bytes) sempre poderá ser armazenado na mesma linha de cache, jamais acarretando mais que uma falta por cache.

O compilador provê então diretivas para alinhamento de endereço. Por exemplo:

```
1 int partial[1024] __attribute__ ((aligned (64)));
```

Este código fará que o endereço inicial do vetor `partial` seja múltiplo de 64. Para alocação dinâmica, o compilador da Intel provê a seguinte função:

```
1 void* _mm_malloc (size_t size, size_t align )
```

O primeiro parâmetro é o tamanho em bytes, e o segundo parâmetro é o alinhamento, que deve ser uma potência de 2. Para desalocar a memória, deve-se utilizar a seguinte função:

```
1 void _mm_free (void *p)
```

Para instruções vetoriais, é recomendado alinhar os vetores em 64 bytes. Com os endereços de memória alinhados, um novo `pragma` pode ser utilizado para indicar ao compilador que os endereços de memória acessados estão alinhados:

```
1 #pragma vector aligned
```

O código então que soluciona o problema aqui tratado pode ser escrito da seguinte forma:

```
1 double vsum (double *a, double *b, int n)
2 {
3       int i;
4       double r = 0;
5       #pragma vector aligned
6       #pragma simd reduction(+:r)
7       for (i=0; i<n; i++)
8             r += a[i] * b[i];
9       return r;
10 }
```

1.2.4. Estudo de caso: multiplicação de matrizes

Como estudo de caso, será utilizado um algoritmo de multiplicação de matrizes, que é um dos exemplos mais comuns de aplicação de paralelismo. O código base de multiplicação de matrizes é o seguinte:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       double sum;
5       for (i=0; i<first_rows; i++) {
6             for (j=0; j<second_cols; j++) {
7                   sum = 0;
8                   for (k=0; k<first_cols; k++)
9                         sum += first[i*first_cols+k] *
                              second[k*second_cols+j];
10                  multiply[i*second_cols+j] = sum;
11            }
12      }
13 }
```

A primeira etapa é paralelizar com instruções vetoriais:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       double sum;
5       for (i=0; i<first_rows; i++) {
6             for (j=0; j<second_cols; j++) {
7                   sum = 0;
8                   #pragma simd reduction(+:sum)
9                   for (k=0; k<first_cols; k++)
10                        sum += first[i*first_cols+k] *
                              second[k*second_cols+j];
11                  multiply[i*second_cols+j] = sum;
12            }
13      }
14 }
```

Entretanto, esta implementação tem um problema, os endereços acessados dentro do laço pela expressão `second[k*second_cols+j]` não são consecutivos em sucessivas iterações, diminuindo o desempenho. Para solucionar isto, uma solução é modificar o algoritmo, de forma a inverter os laços das variáveis `j` e `k`:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       for (i=0; i<first_rows; i++) {
5             for (j=0; j<second_cols; j++)
6                   multiply[i*second_cols+j] = 0.0;
7       }
8       for (i=0; i<first_rows; i++) {
9             for (k=0; k<first_cols; k++) {
10                  #pragma simd
11                  for (j=0; j<second_cols; j++)
12                        multiply[i*second_cols+j] +=
                             first[i*first_cols+k] * second
                             [k*second_cols+j];
13            }
14      }
15 }
```

Agora, todos os endereços acessados no laço vetorizado são consecutivos ou, no caso de `first[i*first_cols+k]`, não se alteram. Com isto, o laço interno é paralelizado com instruções vetoriais.

O próximo passo é o alinhamento de memória. Além, para poder incluir as instruções de alinhamento de memória, é necessário não apenas alocar os dados com a função `_mm_malloc`, mas também que o valor das variáveis `first_cols` e `second_cols` seja múltiplo de 4. Isto deve ocorrer para que os endereços iniciais dos vetores acessados dentro do laço, `multiply[i*second_cols+j]` e `second[k*second_cols+j]`, sejam múltiplos de 4. É importante mencionar que, caso o tipo de dados fosse `float`, deveria ser múltiplo de 8.

Além de paralelizar o laço interno com instruções vetoriais, deve-se paralelizar o laço externo com OpenMP. Dessa forma, é possível aproveitar tanto o paralelismo a nível de *threads* quanto vetorização. A solução final do exercício é:

```
1 void matrix_mult (double *first, double *second, double *
    multiply, int first_rows, int first_cols, int
    second_cols)
2 {
3       int i, j, k;
4       for (i=0; i<first_rows; i++) {
5             for (j=0; j<second_cols; j++)
6                   multiply[i*second_cols+j] = 0.0;
7       }
8       #pragma omp parallel for private(i, j, k)
9       for (i=0; i<first_rows; i++) {
10            for (k=0; k<first_cols; k++) {
11                  #pragma vector aligned
12                  #pragma simd
13                  for (j=0; j<second_cols; j++)
14                        multiply[i*second_cols+j] +=
                             first[i*first_cols+k] * second
                             [k*second_cols+j];
15            }
16      }
17 }
```

1.3. Intel Xeon Phi

O Intel Xeon Phi representa um grande evolução para arquiteturas *many-core*. Ele permite a execução de instruções x86, e a execução do mesmos códigos utilizados em arquiteturas x86 tradicionais. Por exemplo, uma aplicação escrita para o Xeon Phi pode ser facilmente paralelizada com OpenMP, como qualquer máquina x86 traditional [Jeffers and Reinders 2013]. Por outro lado, outras arquiteturas *many-core*, como a GPU, requerem complexas APIs de programação, como CUDA [Cook 2012, Sanders and Kandrot 2010] e OpenCL [Stone et al. 2010]. Devido a isto, o Intel Xeon Phi pode ser adotado mais facilmente que dispositivos baseados nestas outras arquiteturas.

O primeiro processador, utilizando a arquitetura *Knights Corner* [Chrysos 2012], compreende um co-processador com vários núcleos e uma memória dedicada, e é conectado à máquina através da interface PCI express. A Figura 1.3 retrata como o co-processador Xeon Phi interage com o resto da máquina. Ele executa o sistema operacional Linux, separadamente do sistema operacional do hospedeiro. Pode haver vários co-processadores Xeon Phi instalados na mesma máquina. Uma aplicação pode ser executada inteiramente no Xeon Phi, or ser executada parcialmente no Xeon Phi e parcialmente no processador hospedeiro.

A arquitetura Knights Corner está ilustrada na Figura 1.4. É composto por núcleos, memórias cache, controladores de memória, um diretório de *tags* distribuído e um anel bidirecional. Cada núcleo tem caches L1 e L2 privadas. A memória cache é mantida coerente. Quando um acesso à cache L2 resulta numa falta na cache, um diretório de *tag* correspondente ao endereço da falta é acessado. Se a mesma linha de cache está presente

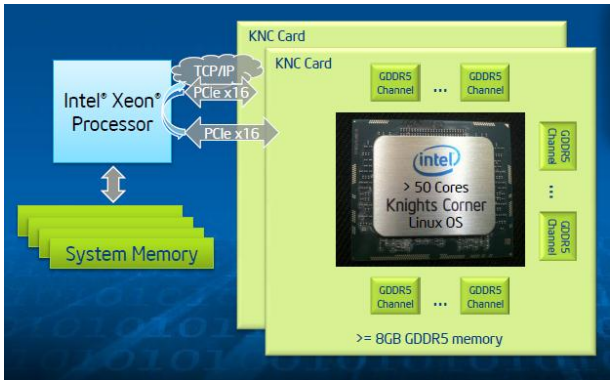


Figura 1.3: Coprocessador Intel Xeon Phi [Chrysos 2012].

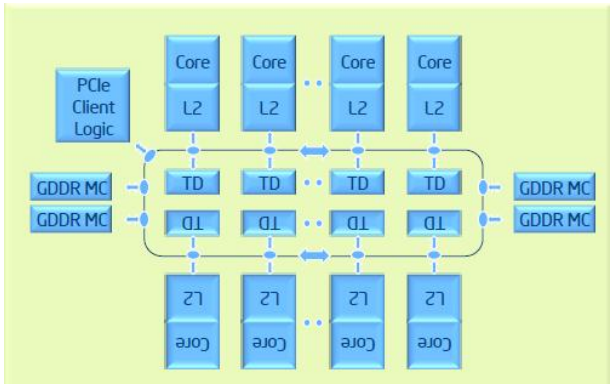


Figura 1.4: Arquitetura Knights Corner [Chrysos 2012].

em outra cache L2, os dados são redirecionados da outra cache. Caso contrário, o dado é buscado da memória primária utilizando o controlador de memória responsável pelo endereço da falta. Os endereços de memória são distribuídos de maneira uniforme entre os diretórios de *tag* e controladores de memória de forma a distribuir o tráfego.

Os núcleos implementam uma *pipeline* em ordem e são capazes de executar 4 *threads* em paralelo usando a técnica *Simultaneous Multithreading* (SMT). Desta forma, uma alta taxa de execução de instruções deve ser atingida. Uma das principais vantagens da arquitetura é sua unidade de processamento vetorial, que permite que a mesma operação seja realizada em vários operandos simultaneamente. O desempenho pode ser aumentado de forma acentuada se a aplicação usa instruções de processamento vetorial (AVX) de forma adequada. Até 16 operações de ponto flutuante de precisão simples, ou 8 operações de precisão dupla, podem ser realizadas na mesma instrução vetorial.

No restante desta seção, é explicado como programar o Xeon Phi.

1.3.1. Execução nativa

Como o Xeon Phi roda um sistema operacional próprio, uma versão do Linux, é possível executar aplicações nativamente dentro dele. Para tal, é necessário primeiro compilar o código fonte usando as *flags* adequadas:

```
1 icc -o <binary> <input.c> -mmic -static
```

Utiliza-se o `-mmic` para especificar ao compilador que o binário será executada nativamente no Xeon Phi. Utiliza-se o `-static` para que o *linker* inclua as bibliotecas no binários. Este último não é mandatório, porém sua ausência implicará que as bibliotecas dinâmicas devam também ser copiadas para dentro do Xeon Phi.

Com o binário gerado, é preciso copiá-lo para dentro do Xeon Phi. Há duas principais maneiras de se fazer isso:

1. O Xeon Phi executa, por padrão, um servidor SSH. Para acessá-lo, basta acessar `ssh mic0`. É possível também então transferir o binário via SCP.
2. O Xeon Phi permite montar diretórios remotos via NFS. Desta forma, é possível compartilhar pastas entre o Xeon Phi e a máquina hospedeira, sem necessitar de uma transferência explícita.

Independente da maneira escolhida, supondo que o binário esteja dentro do Xeon Phi, basta conectar via SSH no Xeon Phi e executar a aplicação, da mesma forma que se faz em uma máquina rodando Linux.

1.3.2. Offload de código

Outra maneira de rodar aplicações dentro do Xeon Phi é fazendo *offload* de código. Desta maneira, um binário executando na máquina hospedeira, faz um desvio da execução para o Xeon Phi, que, depois de executado parte do código, retorna a execução para a máquina hospedeira. Além do código binário do programa, os dados precisam ser também transferidos. Considere a seguinte função de soma dos elementos de um vetor:

```
1 double sum (double *v, int n)
2 {
3       int i;
4       double sum = 0.0;
5       for (i=0; i<n; i++)
6             sum += v[i];
7       return sum;
8 }
```

Para ser executada esta soma no Xeon Phi, temos que inserir a diretiva `pragma offload`:

```
1 double sum (double *v, int n)
2 {
3       int i;
4       double sum = 0.0;
5       #pragma offload target(mic) in(v:length(n))
6       {
7             for (i=0; i<n; i++)
8                   sum += v[i];
9       }
10      return sum;
11 }
```

A seguir a explicação das principais diretivas (algumas não estão no exemplo acima):

pragma offload Determina que o próximo bloco será executado no coprocessador, e não no processador hospedeiro.

target(mic) Especifica qual coprocessador irá executar.

in(ponteiro:length(n)) Especifica que os *n* elementos a partir do *ponteiro* deverão ser transferido do hospedeiro para o coprocessador antes do início da execução do bloco. O ambiente automaticamente considera o tipo dos dados para calcular o tamanho em *bytes*.

out(ponteiro:length(n)) Especifica que os *n* elementos a partir do *ponteiro* deverão ser transferido do coprocessador para o processador hospedeiro após o término da execução do bloco. O ambiente automaticamente considera o tipo dos dados para calcular o tamanho em *bytes*.

inout(ponteiro:length(n)) Tem o mesmo resultado de fazer *in* e *out* com os mesmos parâmetros.

Só é necessário especificar o conteúdo apontado por ponteiros nas diretivas *in* e *out*. Variáveis normais são automaticamente transferidas pelo ambiente quando são acessadas dentro da região com `offload`. Além disso, a região interna do `offload` pode ser normalmente paralelizada com OpenMP e instruções vetoriais, conforme as seções anteriores.

Exemplo com OpenMP:

```
 1 double sum (double *v, int n)
 2 {
 3       int i, nthreads;
 4       double sum = 0.0;
 5       #pragma offload target(mic) in(v:length(n))
 6       {
 7             #pragma omp parallel for reduction(+:sum)
 8             for (i=0; i<n; i++)
 9                   sum += v[i];
10       }
11       return sum;
12 }
```

Exemplo com instruções vetoriais:

```
1 double sum (double *v, int n)
2 {
3       int i;
4       double sum = 0.0;
5       #pragma offload target(mic) in(v:length(n))
6       {
7             #pragma vector aligned
8             #pragma simd reduction(+:sum)
9             for (i=0; i<n; i++)
10                  sum += v[i];
11      }
12      return sum;
13 }
```

Como exercício final, deve-se programar para executar no Xeon Phi, usando `offload`, o código de multiplicação de matrizes apresentado na Seção 1.2.4. O código final é:

```
1 void matrix_mult (myfloat *first, myfloat *second,
     myfloat *multiply, int first_rows, int first_cols, int
      second_cols)
2 {
3       #pragma offload target(mic) in(first:length(
           first_rows*first_cols)) in(second:length(
           first_cols*second_cols)) out(multiply:length(
           first_rows*second_cols))
4       {
5             int i, j, k;
6             for (i=0; i<first_rows; i++) {
7                   for (j=0; j<second_cols; j++)
8                         multiply[i*second_cols+j] = 0.0;
9             }
10            #pragma omp parallel for private(i, j, k)
11            for (i=0; i<first_rows; i++) {
12                  for (k=0; k<first_cols; k++) {
13                        #pragma vector aligned
14                        #pragma simd
15                        for (j=0; j<second_cols; j++) {
16                              multiply[i*second_cols+j]
                                   += first[i*first_cols+k]
                                    * second[k*second_cols+
                                   j];
17                        }
18                  }
19            }
20      }
21 }
```

1.4. Conclusão

Neste curso, foram apresentadas novas técnicas de programação a serem aplicadas nas novas arquiteturas. Tais técnicas são primordiais para extrair desempenho dos novos sistemas. As técnicas apresentadas permitem fazer uso dos múltiplos núcleos de arquiteturas *multicore* e *manycore*. Permitem também que cada núcleo, individualmente, possa aproveitar o paralelismo vetorial.

Referências

[Chapman et al. 2008] Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

[Chrysos 2012] Chrysos, G. (2012). Intel Xeon Phi X100 Family Coprocessor - the Architecture.

[Cook 2012] Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

[Jeffers and Reinders 2013] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi coprocessor high-performance programming*. Newnes.

[Lee et al. 2010] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460.

[Sanders and Kandrot 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[Satish et al. 2012] Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., and Dubey, P. (2012). Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society.

[Stone et al. 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73.