Programación Multinúcleo y extensiones SIMD

Material de apoyo para la práctica 2

Arquitectura de Computadores
Universidade de Santiago de Compostela

Índice

Pautas de evaluación

- **Estructura y legiblidad general del informe. Grado de cumplimiento de las especificaciones del enunciado. Hasta 2 puntos.**
- **Apartados 1 y 2:**
  - **Experimentación*. Hasta 1.5 puntos.**
  - **Interpretación de los resultados**. Hasta 1.5 puntos.**
- **Apartado 3:**
  - **Experimentación. Hasta 1.5 puntos.**
  - **Interpretación de los resultados. Hasta 1.5 puntos.**
- **Apartado 4:**
  - **Experimentación. Hasta 1 punto.**
  - **Interpretación de los resultados. Hasta 1 punto.**

* Código correcto, experimentos adecuados y suficientes.

** Explicación clara y justificada, con longitud proporcionada a la complejidad de lo que se está explicando.

Criterios de entrega

- Es **indispensable** que los códigos entregados **compilen y se ejecuten** correctamente con el script proporcionado.

- Para que la entrega sea evaluada es necesario **entregar, como mínimo**, los siguientes apartados:
  - Apartado 1
  - Apartados 2.1, 2.2 y 2.3
  - Apartado 4.1

Optimización del uso de memoria caché

Optimización del uso de memoria caché

- **Métricas a reducir:**
  - Tiempo de búsqueda (hit time). Tasa de fallos (miss rate).
  - Penalización de fallos (miss penalty).

- **Métricas a aumentar:**
  - Ancho de banda de caché (cache bandwidth).

Todas las optimizaciones avanzadas buscan mejorar alguna de estas métricas.

Optimización del uso de memoria caché

Optimizaciones avanzadas que se aplican en los procesadores actuales:

- Cachés pequeñas y simples
- Predicción de vía
- Acceso segmentado a la caché
- Cachés no bloqueantes
- Cachés multi-banco
- Palabra crítica primero y reinicio temprano
- Mezclas en búfer de escritura
- Optimizaciones del compilador
- Lectura adelantada hardware

El programador puede aplicar manualmente algunas de las optimizaciones que aplica el compilador pero no puede actuar sobre otras.

Optimizaciones del compilador

- Mejoran la tasa de fallos
- Con complejidad baja para el hardware.
- **El desafío está en que el software permita realizarlas.**
- Trataremos de aplicar alguna para mejorar el código secuencial
- de la práctica 2.

Referencia sobre optimizaciones avanzadas de memoria cache:
Computer Architecture. A Quantitative Approach 5th Ed. Hennessy and Patterson. Secciones: 2.1, 2.2.

Reordenación de procedimientos

- **Objetivo:** Reducir los fallos por conflicto debidos a que dos procedimientos coincidentes en el tiempo se corresponden con la misma línea de caché.
- **Técnica:** Reordenar los procedimientos en memoria para separar los que acceden a datos en las mismas líneas caché.

Alineación de bloques básicos

- **Definición:** Un **bloque básico** es un conjunto de instrucciones que se ejecuta secuencialmente (no contiene saltos).
- **Objetivo:** Reducir la posibilidad de **fallos de caché** para código secuencial.
- **Técnica:** Hacer coincidir la **primera instrucción** de un bloque básico con la **primera palabra** de una línea de caché.

Linearización de saltos

- **Definición:** Un **bloque básico** es un conjunto de instrucciones que se ejecuta secuencialmente (no contiene saltos).
- **Técnica:** Si el compilador sabe que es probable que se tome un salto, puede cambiar el sentido de la condición e intercambiar los bloques básicos de las dos alternativas.
- Algunos compiladores definen extensiones para dar pistas al compilador. Ejemplo: gcc ("likely")

Distribución optimizada de datos en estructuras

- Cambiamos la ubicación de los campos para poner adyacentes los que se usan conjuntamente

```
struct datos {
  float coef, b, c;
  char id[20];
  float posicion[3];
     double v;
 };

struct datos ar[N];

for (i=0; i<M; i++)
 r += ar[ d[i] ].coef * ar[ d[i] ].v;
```

```
struct datos {
     double v;
  float coef, b, c;
  char id[20];
  float posicion[3];
};

struct datos ar[N];

for (i=0; i<M; i++)
r += ar[ d[i] ].coef * ar[ d[i] ].v;
```

Intercambio de bucles

- **Objetivo:** Mejorar localidad espacial.
- Dependiente del modelo de almacenamiento vinculado al lenguaje de programación (por filas o por columnas). Ejemplo: Fortran vs C.

Acceso con saltos

```
for ( int j=0; j<100; ++j) {
 for ( int i=0; i<5000; ++i) {
  v[ i ][ j ] = k * v[ i ][ j ];
 }
}
```

Accesos secuenciales

```
for ( int i=0; i<5000; ++i) {
 for ( int j=0; j<100; ++j) {
  v[ i ][ j ] = k * v[ i ][ j ];
 }
}
```

Fusión de bucles

- **Objetivo:** Mejorar localidad temporal.
- Puede reducir localidad espacial.
- Puede ser aplicada por el programador de manera manual.

Bucles separados

```
for ( int i=0; i<rows; ++i) {
for ( int j=0; j<cols; ++j) {
  a[ i ][ j ] = b[ i ][ j ] * c[ i ][ j ];
}
}
for ( int i=0; i<rows; ++i) {
for ( int j=0; j<cols; ++j) {
  d[ i ][ j ] = a[ i ][ j ] + c[ i ][ j ];
}
}
```

Bucles fusionados

```
for ( int i=0; i<rows; ++i) {
for ( int j=0; j<cols; ++j) {
  a[ i ][ j ] = b[ i ][ j ] * c[ i ][ j ];
  d[ i ][ j ] = a[ i ][ j ] + c[ i ][ j ];
}
}
```

Acceso por bloques (blocking o tiling)

- En lugar de acceder filas o columnas enteras, las subdividimos en bloques y reusamos datos antes de que el bloque sea reemplazado de la cache.
- Requiere más acceso de memoria pero mejora la localidad en los accesos.
- Hay aumento de tamaño de código y de cálculos: no siempre se reducirá el tiempo de ejecución
- Elegir tamaño de bloque adecuado para que los datos utilizados en un momento dado quepan en la memoria caché.

Acceso por bloques (blocking o tiling)

Producto original

```
for ( int i=0; i<N; ++i) {
 for ( int j=0; j<N; ++j) {
   r=0;
   for ( int k=0; k<N; ++k) {
     r+= b[ i ][ k] * c[k ][ j ];
   }
   a[ i ][ j ] = r ;
 }
}
```

Producto con acceso por bloques

```
for ( bj=0; bj<N; bj+=bsize) {
  for (bk=0; bk<N; bk +=bsize) {
    for ( i=0; i<N; ++i) {
      for ( j=bj; j<min(bj+bsize,N); ++j) {
        r=0;
        for (k=bk; k<min(bk+bsize,N); ++k){
          r +=b[i ][ k] * c[k ][ j ];
        }
        a[ i ][ j ] += r;
      }
    }
  }
}
```

Lectura hardware adelantada de instrucciones

- También llamada prebúsqueda o prefetching.
- Las instrucciones presentan alta localidad espacial.

- **Instruction prefetching:** Lectura adelantada de instrucciones.
  - Lectura de dos bloques en caso de fallo.
  - Bloque que provoca el fallo.
  - Bloque siguiente.

- **Ubicación:**
  - Bloque que provoca el fallo → caché de instrucciones.
  - Bloque siguiente → búfer de instrucciones.

Lectura hardware adelantada de instrucciones

Ejemplo concreto del Pentium 4:

- **Data prefetching:** Permite lectura adelantada de páginas de 4KB a caché L2.
- **Se invoca lectura adelantada si:**
  - 2 fallos en L2 debidos a una misma página.
  - Distancia entre fallos menor que 256 bytes.

En los procesadores de Intel actuales se realizan precarga software y hardware.

Paralelismo a nivel de datos

Extensiones SIMD

Extensiones SIMD

- Los procesadores evolucionan año tras año.
- Los ciclos de reloj se fueron aumentando hasta que llego un punto donde la tecnología no fue capaz de llevar el ritmo. (Sobre 2010).
- Hubo que buscar alternativas para seguir mejorando:
  - Añadir más cores a cada procesador (Procesadores multinúcleo).
  - Paralelismo a nivel de datos (SIMD).

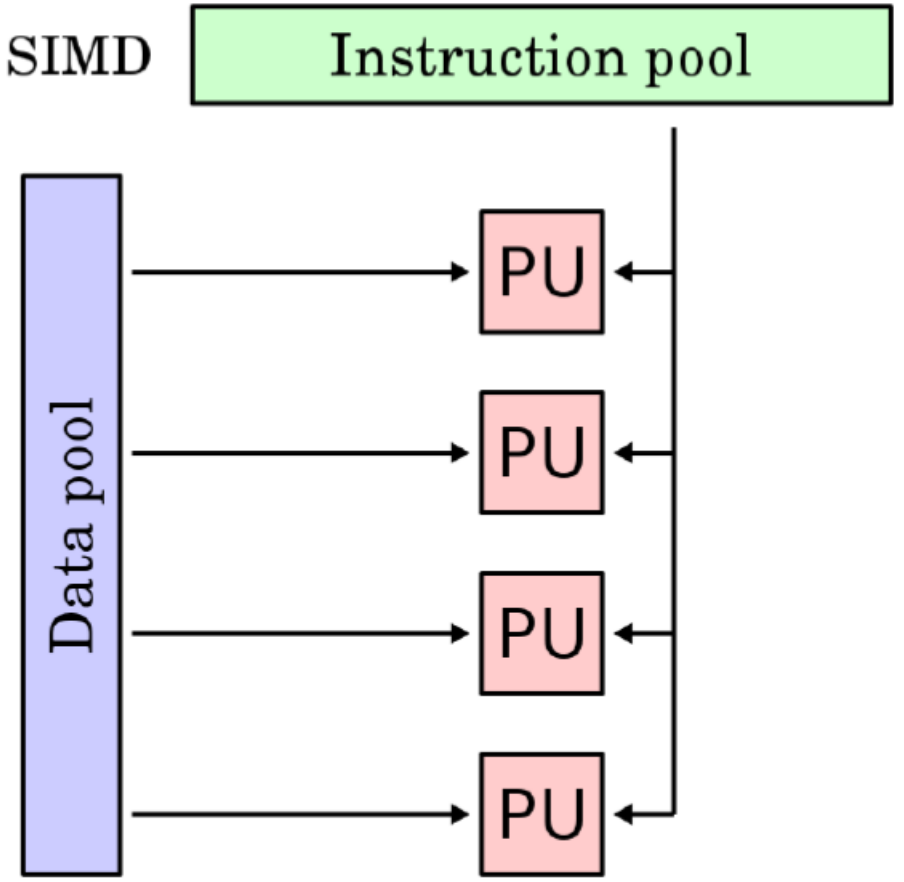
Extensiones SIMD

- Si aumentamos la frecuencia del procesador, nuestro programa ira más rápido por sí solo.
- Si el procesador tiene más unidades de paralelización, nuestro programa no tiene por qué mejorar. **REQUIERE UN PROCESO MANUAL**
- ¿Quién paraleliza un programa?
  - El hardware
  - El compilador
  - El programador

Extensiones SIMD

- SIMD (***S**ingle **I**nstruction, **M**ultiple**D**ata*)

- Mejora el rendimiento en las nuevas aplicaciones
  - Procesado de imagen
  - Tratamiento de vídeo
  - Procesamiento de audio
  - Modelado 3D

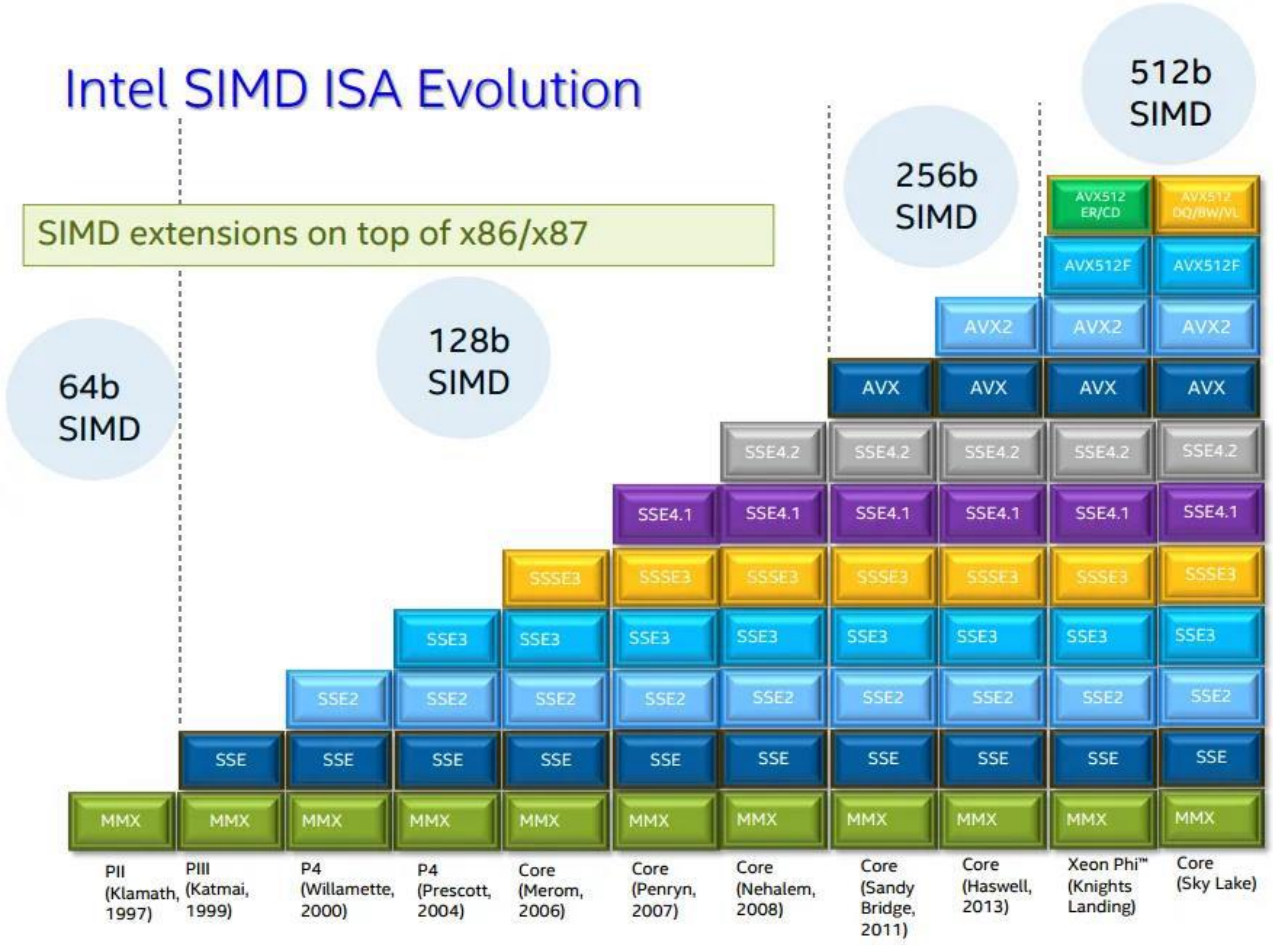


Extensiones SIMD

Permite realizar operaciones en paralelo

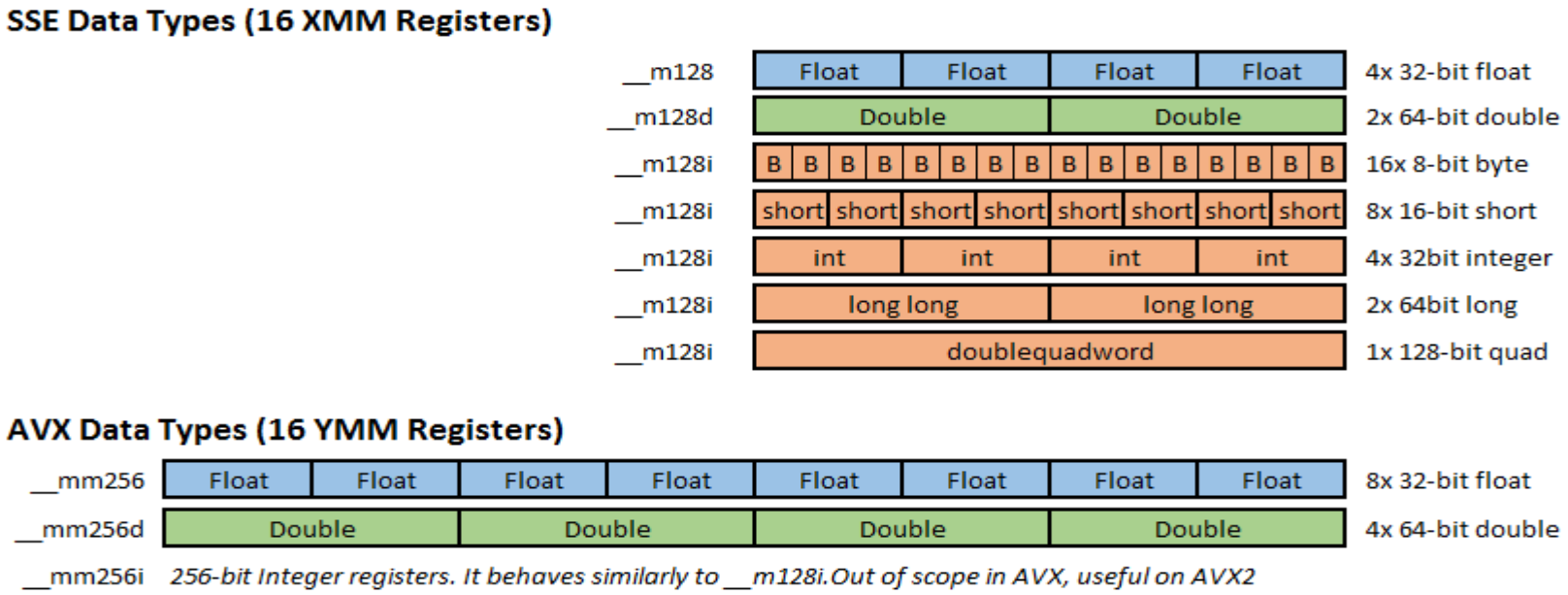
- Aritméticas (ADD, SUB, MUL, DIV, SQRT, MAX, MIN, RCP, etc)
- Lógicas (AND, OR, XOR, ANDN, etc)
- Comparaciones
- Shuffle
- Manipulación de Bits
- Funciones Matemáticas
- Criptografía
- Conversión
- Y muchas más….

Evolución


Intel SIMD ISA Evolution
SIMD extensions on top of x86/x87
64b SIMD
128b SIMD
256b SIMD
512b SIMD
MMX
SSE
SSE2
SSE3
SSSE3
SSE4.1
SSE4.2
AVX
AVX2
AVX512F
AVX512 ER/CD
AVX512 DQ/BW/VL
PII (Klamath, 1997)
PIII (Katmai, 1999)
P4 (Willamette, 2000)
P4 (Prescott, 2004)
Core (Merom, 2006)
Core (Penryn, 2007)
Core (Nehalem, 2008)
Core (Sandy Bridge, 2011)
Core (Haswell, 2013)
Xeon Phi™ (Knights Landing)
Core (Sky Lake)

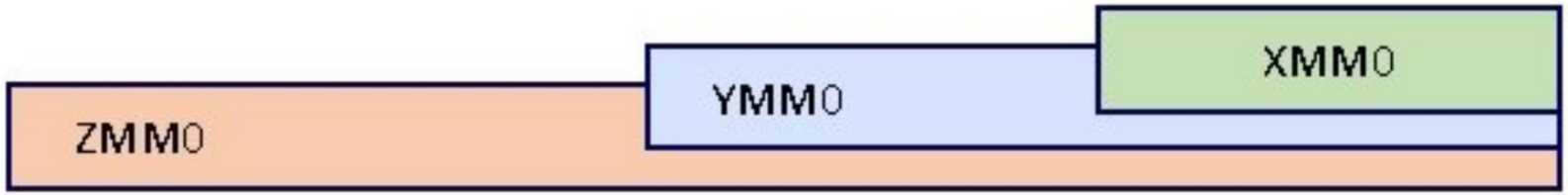
Registros

- SSE y AVX tienen 16 registros cada uno. En SSE son nombrados como XMM0-XMM15, en AVX YMM0-YMM15 y en AVX512 ZMM0-ZMM31 (añade otros 16 registros).
- Los XMM tienen una longitud de 128 bits, los YMM 256 bits y los ZMM 512 bits.



Registros

- SSE añade tres tipos de dato __m128 __m128d __m128i de tipo float, double e integer respectivamente.
- AVX añade tres tipos de dato __m256 __m256d __m256i de tipo float, double e integer respectivamente.
- AVX512 añade tres tipos de dato __m512 __m512d __m512i de tipo float, double e integer respectivamente.



- La parte inferior de los registros coincide con los registros de menor capacidad, por esa razón hay que tener cuidado al mezclar instrucciones de diferente longitud.

Instrucciones

- Cada tipo de extensión vectorial tiene su propia cabecera, podemos incluirlas por separado o usar **< immintrin.h >** y dejar que el compilador se encargue de seleccionar las que podemos usar.
- El nombre de las instrucciones se puede separar en 3 partes:
  - **Longitud de la instrucción:** _mm, _mm256, _ mm512
  - **Tipo de operación:** load, store, add
  - **Tipo de dato:** ps (float), pd (double), epi8, epi16, epi32, epi64 (integers)
    - **_mm_load_ps:** Carga 4 floats en un registro de 128 bits
    - **_mm256_add_pd:** Suma dos registros de 256 bits
    - **_mm512_store_epi32:** Guarda un registro de 512 bits como 16 enteros de 32 bits

Alineamiento

- Las funciones load y store requieren que la memoria este alineada con el registro vectorial.
- El compilador no realiza ningún tipo de comprobación sobre el alineamiento de las funciones. No alineado = segmentation fault
- Si no es posible alinear, existen versiones *loadu* y *storeu* que pueden crear registros sobre memoria no alineada. Requieren más ciclos

Compilador

- Los compiladores intentan preservar la portabilidad, ante todo.
- Las extensiones se activan mediante macros del preprocesador. No debemos definirlas manualmente, pero podemos usarlas para activar código más eficiente si el hardware lo soporta.
- En gcc podemos usar *gcc dM E --< dev null* para comprobar que macros esta definidas.
- Es posible compilar usando cualquier extensión, pero sin el hardware no se podrá ejecutar.
- Las optimizaciones O1 a ON de los compiladores solo hacen uso de las extensiones habilitadas.

Alineamiento

```
[uscecjlf@login210-19 ~]$ gcc -O3 -dM -E - < /dev/null | egrep "SSE|AVX"
#define __MMX_WITH_SSE__ 1
#define __SSE2_MATH__ 1
#define __SSE_MATH__ 1
#define __SSE__ 1
#define __SSE2__ 1
```

```
[uscecjlf@login210-19 ~]$ gcc -msse4.2 -dM -E - < /dev/null | egrep "SSE|AVX"
#define __SSE4_1__ 1
#define __SSE4_2__ 1
#define __MMX_WITH_SSE__ 1
#define __SSE2_MATH__ 1
#define __SSE_MATH__ 1
#define __SSSE3__ 1
#define __SSE__ 1
#define __SSE2__ 1
#define __SSE3__ 1
```

```
[uscecjlf@login210-19 ~]$ gcc -mavx2 -dM -E - < /dev/null | egrep "SSE|AVX"
#define __SSE4_1__ 1
#define __SSE4_2__ 1
#define __MMX_WITH_SSE__ 1
#define __SSE2_MATH__ 1
#define __AVX__ 1
#define __SSE_MATH__ 1
#define __AVX2__ 1
#define __SSSE3__ 1
#define __SSE__ 1
#define __SSE2__ 1
#define __SSE3__ 1
```

Alineamiento

```
[uscecjlf@login210-19 ~]$ gcc -march=native -dM -E - < /dev/null | egrep "SSE|AVX"
#define __AVX512F__ 1
#define __AVX512BITALG__ 1
#define __SSE4_1__ 1
#define __AVX512VBMI2__ 1
#define __AVX512VBMI__ 1
#define __SSE__ 1
#define __SSE4_2__ 1
#define __AVX512IFMA__ 1
#define __MMX_WITH_SSE__ 1
#define __AVX512BW__ 1
#define __SSE2_MATH__ 1
#define __AVX__ 1
#define __AVX512VL__ 1
#define __AVX512CD__ 1
#define __AVX512VNNI__ 1
#define __AVX512VPOPCNTDQ__ 1
#define __SSE_MATH__ 1
#define __AVX512DQ__ 1
#define __AVX2__ 1
#define __SSSE3__ 1
#define __SSE3__ 1
#define __SSE2__ 1
```

Ejemplos

- Bucle básico

```
for (int i = 0; i < n; i++) {
    f3[i] = f1[i] * f2[i];
}
```

- Bucle desenrollado

```
for (int i = 0; i < n; i+=4) {
    f3[i] = f1[i] * f2[i];
    f3[i + 1] = f1[i + 1] * f2[i + 1];
    f3[i + 2] = f1[i + 2] * f2[i + 2];
    f3[i + 3] = f1[i + 3] * f2[i + 3];
}
```

Ejemplos

- Bucle vectorizado

```
for (int64 i = 0; i < n; i += 4) {
    __m128 a, b, c;
    a = _mm_load_ps(f1 + i);
    b = _mm_load_ps(f2 + i);
    c = _mm_mul_ps(a, b);
    _mm_store_ps(f3 + i, c);
}
```

- GCC optimizara los bucles con extensiones vectoriales con *O3* o *ftree vectorize*.

Referencias

- Intel® 64 and IA-32 Architectures Software Developer's Manuals (volumes 2A and 2B)
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
  - (última consulta 14 de marzo de 2023)
- Intel IntrinsicsGuide
  - https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
  - (última consulta 14 de marzo de 2023)
- Getting started with SSE programming
  - http://supercomputingblog.com/optimization/getting-started-with-sse-programming/
  - (última consulta 14 de marzo de 2023)
- Intel® C++ Compiler Classic Developer Guide and Reference. Development Reference Guides. Intrinsics for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Instructions
  - https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx-512-instructions.html
  - (última consulta 14 de marzo de 2023)

Programación paralela en memoria compartida

OpenMP

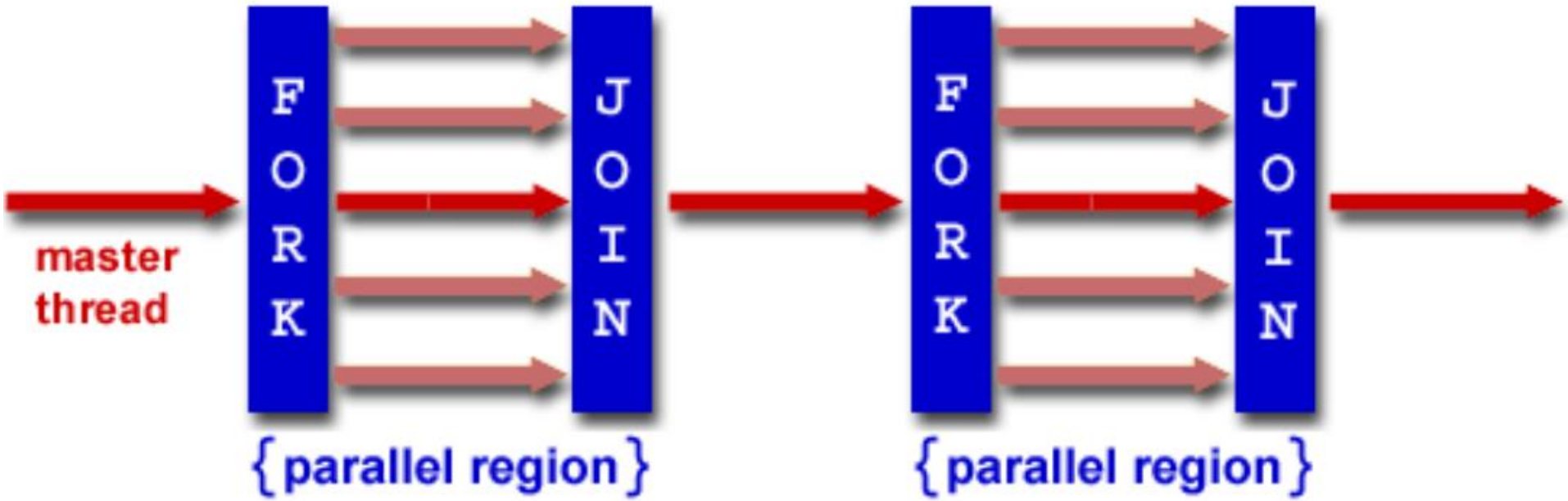
OpenMP

- Estándar industrial para **programación en memoria compartida**.
- Directivas de compilación, biblioteca de funciones y variables de entorno.
- Soportado para Fortran, C y C++.
  - 1998: versión 1.0 para C/C++.
  - 2002: versión 2.0 para C/C++.
  - 2008: versión 3.0.
  - 2013: versión 4.0.

Modelo Fork-Join

- Modelo de paralelismo basado en hilos.
- **Paralelismo explícito:** El programador debe indicar las partes del código paralelizables
- **Modelo fork-join:**
  - Hilo "maestro" ejecutando código en serie.
  - Fork: Se crean los hilos para operar en paralelo.
  - Join: terminación síncronizada de los hilos y continuación de la ejecución serie en el hilo maestro.

Modelo Fork-Join



Estructura general del código

```
#include <omp.h>
main () {
int var1, var2, var3;
Serial code
..
//Beginning of parallel section. Fork a team of threads. Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3){
Parallel section executed by all threads
...
...
All threads join master thread and disband
}
Resume serial code
}
```

Directiva parallel

#pragma omp parallel [clause ...]

private (list)

shared (list)

num_threads(int)

.....

{ Bloque de código }

Directiva parallel

- El código del bloque estructurado es ejecutado por un grupo de hilos.
- El hilo maestro pasa a ser el hilo con ID 0.
- Existe una barrera implícita al final de la sección paralela. Sólo el hilo 0 seguirá a partir de ese punto.
- **Número de hilos:** Opción num_threads(int).

Directiva parallel

- **Private (list):** lista de variables privadas a cada hilo del equipo. Valor sin inicializar.
- **Shared(list):** lista de variables compartidas entre todos los hilos.
- Por defecto las variables declaradas antes de entrar en parallel son shared y las variables declaradas dentro da zona parallel son privadas.

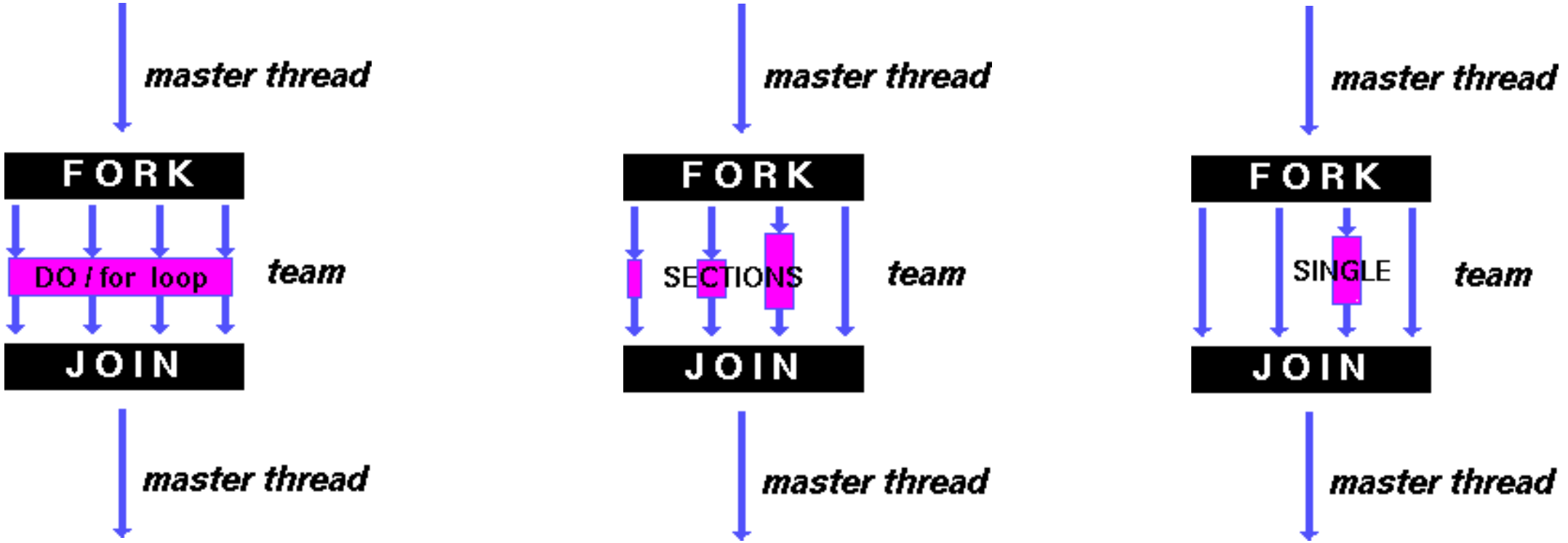
Ejemplo de parallel

```
#include <omp.h>
#define k 8
main () {
  int nthreads, tid;
  /* Inicia el conjunto de hilos con copias privadas de las variables */
  #pragma omp parallel private(nthreads, tid) num_threads(k){
  /* Obtener e imprimir el rango del hilo*/
  tid = omp_get_thread_num();
  printf("Hola desde el hilo = %d\n", tid);
  /* código para el hilo master */
  if (tid == 0) {
     nthreads = omp_get_num_threads();
     printf("Número de hilos= %d\n", nthreads);
  }
} /* sincronización final (join) y terminación */ }
```

Construcciones para división de trabajo

- **for**: reparte iteraciones de un blucle entre el equipo de hilos (paralelismo de datos).
- **Sections**: conjunto discreto de secciones a repartir entre el equipo de hilos (paralelismo funcional).
- **Single**: serializa una sección de código (ejecución única por parte de un hilo).
- **Barrera implícita al final de la construcción.**

Construcciones para división de trabajo



Construcción for

#pragma omp for

Bucle for a paralelizar

Los hilos ejecutan concurrentemente diferentes iteraciones del bucle. Es esencial tener en cuenta esto para garantizar que los resultados sean correctos cuando una iteración del bucle depende de resultados de iteraciones anteriores.

Construcción for: ejemplo

```
#include <omp.h>
#define n 1000
#define k 8

main () {
 int i;
 float a[n], b[n], c[n];
 for (i=0; i < N; i++)
   a[i] = b[i] = i * 1.0;
 #pragma omp parallel shared(a,b,c) private(i) num_threads (k){
 #pragma omp for
 for (i=0; i < n; i++)
   c[i] = a[i] + b[i];
 }
}
```

Ejemplo problemático

```
#include <omp.h>
#define k 8
main () {
int i, n;
float a[100], b[100], result, my_result,p_result[k];
n = 100;
result = 0.0;
for (i=0; i < n; i++) {
  a[i] = i * 1.0; b[i] = i * 2.0;
 }
#pragma omp parallel private(i,my_result) num_threads(k){
#pragma omp for
for (i=0; i < n; i++) // bucle for a paralelizar
  my_result = my_result + (a[i] * b[i]);
  p_result[omp_get_thread_num()]=my_result;
}
for(i=0;i<k;i++) result=result+p_result[i]; // no se paraleliza
  printf("Final result= %f\n",result);
}
```

Sections

```
#pragma omp sections [clause ...]{
#pragma omp section
{bloque estructurado}
#pragma omp section
{bloque estructurado}
} (/* Barrera implícita */
```

Ejemplo: Sections

```
#include <omp.h>
#define N 1000
main (){
  int i;
  float a[N], b[N], c[N];
  for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
  #pragma omp parallel shared(a,b,c) private(i){
  #pragma omp sections{
  #pragma omp section
  for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
  #pragma omp section
  for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
} } }
```

Single

#pragma omp single [clause ...]

{bloque estructurado}

Ejecución del bloque estructurado en un único hilo

Ejemplo con single

```
#include <omp.h>
#define n 1000
#define k 8
main () {
  int i;
  int a[n], b[n], c[n],part_count[k],count=0; int my_count=0;
  for (i=0; i < N; i++)
  {a[i] = rand(); b[i]= rand();}
  #pragma omp parallel shared(a,b,c) private(i,my_count) num_threads (k){
  #pragma omp for
  for (i=0; i < n; i++){
     c[i] = a[i] + b[i];
     if (c[i]==0) my_count++;}
   part_count[omp_get_thread_num()]=my_count++;
   #pragma omp single{
   for(i=0;i<k;i++) count=count+part_count[k];
      printf("count=%d \n",count);}
  }
}
```

Directivas de sincronización

- Directiva Master.
- Directiva Critical.
- Directiva Barrier.
- Directiva Atomic.

Master

- Indica que el bloque estructurado sólo debe ejecutarlo el hilo maestro
- No existe barrera implícita. El resto de hilos ingoran el bloque.

```
#pragma omp master
        structured_block
```

Critical

- **Serializa el código, utilizar con precaución**
- Especifica una zona de código (sección crítica) que debe ser ejecutada únicamente por un hilo cada vez.
- Se puede asignar un nombre a la sección crítica. Las secciones con el mismo nombre actúan como una única sección crítica.

#pragma omp critical [name]

{bloque estructurado}

Ejemplo de Critical

```
#include <omp.h>
#define k 8
main() {
 int x,y;
 x = 100; y=100;
 #pragma omp parallel shared(x,y) num_threads (8){
  #pragma omp critical{
  x = x + 1;
  y = y - 1;
  }
 }
}
```

Barrier y Atomic

- Barrera de sincronización del grupo de hilos

#pragma omp barrier

- Implementación eficiente de una sección crítica simple (asignación del resultado de una operación aritmética a una variable). Ninguna escritura por parte de otro hilo interfiere en la evaluación de la expresión

#pragma omp atomic

expression_statement

Ejemplo con Atomic

```
#include <omp.h>
#define k 8
 main() {
  int x[100];
  #pragma omp parallel shared(x) num_threads (k){
  #pragma omp for
  for (i=0; i < 100; i++){
    #pragma omp atomic
    x[rand()] += i;
  }
 }
}
```

¡Puede no ser eficiente!
Serialización del código

Funciones auxiliares

- **int omp_get_num_threads(void):** devuelve el número de hilos ejecutando una región paralela.
- **int omp_get_thread_num(void):** número natural que identifica el hilo en el equipo.
- **int omp_get_num_procs(void):** número de procesadores disponibles.
- **omp_set_num_threads(int n_threads):** ajusta el número de threads que deben ejecutarse en paralelo.