Universidad de Costa Rica
Escuela de Ciencias de la Computación e Informática
CI-0119 Proyecto Integrador de Lenguaje Ensamblador y Arquitectura

Proyecto 2: segundo entregable

Integrantes

Juan Ignacio Pacheco Castro (B85841)

David Xie Li (B88682)

Profesor

Pablo Sauma Chacón

Grupo 4

Ciudad Universitaria Rodrigo Facio, Costa Rica
II semestre del 2020

Descripción de la solución

En este proyecto se implementó una función en lenguaje ensamblador que permite la rápida multiplicación de dos matrices planas (matriz almacenada en un único nivel: un único puntero) de datos tipo float (real de 4 bytes). Para ello se usaron los sets de instrucciones SSE investigados, especialmente las instrucciones: `movss, movaps, shufps, addss, addps, mulps, mulss, pxor.`

Primero que todo, se decidió utilizar el estándar de 64 bits y la función en ensamblador `quickMatrixMul` se implementó como un código independiente escrito en GAS en un archivo aparte `quickMatrixMul.s.`

La función de multiplicación de matrices flotantes (cuyo producto se almacena en la matriz resultado) mencionada anteriormente se llevó a cabo primero analizando los parámetros que recibía, los cuales eran 6, en el siguiente orden:

1. **Matriz NM:** la matriz **`A`** en la multiplicación de **`AxB = C`**, de tamaño **`n`** filas y **`m`** columnas. Este parámetro va en el registro **`%rdi.`**
2. **Matriz MP:** la matriz **`B`** en la multiplicación de **`AxB = C`**, de tamaño **`m`** filas y **`p`** columnas. Va en el registro **`%rsi.`**
3. **Matriz NP:** la matriz **`C`** donde se almacenará el resultado de la multiplicación de **`AxB`**, de tamaño **`n`** filas y **`p`** columnas. Va en el registro **`%rdx.`**
4. **n**: el tamaño de filas de la matriz **NM** y el de las filas de la matriz **NP**. Va en el registro **`%rcx.`**
5. **m**: el tamaño de columnas de la matriz **NM** y el de las filas de la matriz **MP**. Va en el registro **`%r8.`**
6. **p**: el tamaño de las columnas de la matriz **MP** y **NP**. Va en el registro **`%r9.`**

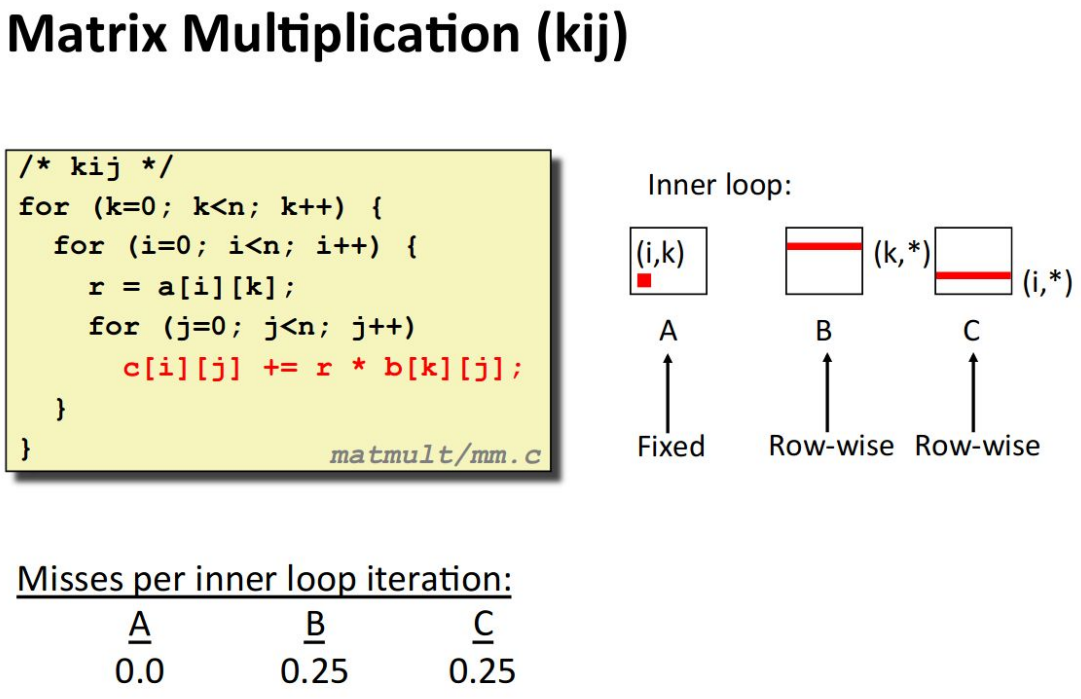
Posteriormente se determinó cuál era el mejor recorrido que se podía realizar para hacerlo lo más eficiente posible y para aprovechar la localidad de la caché, para ello se revisó la presentación vista en clases acerca del tema de la caché para llegar a que el recorrido más óptimo era el de `kij`. Por lo que llegamos a implementar primeramente la función en C para tener una plantilla para luego hacerlo en ensamblador, que se puede apreciar a continuación:

```
void quickMatrixMul(float* matrixNM, float* matrixMP, float*
```

```
matrixNP, unsigned n, unsigned m, unsigned p){
    unsigned int i, j, k;
    float r = 0;
    for (k = 0; k < m; ++k) {                // L1
        for (i = 0; i < n; ++i) {            // L2
            r = matrixNM[i*m + k];
            for (j = 0; j < p; ++j)          // L3
                matrixNP[i*p + j] += r * matrixMP[k*p + j];
        }
    }
}
```

De primero está el ciclo con el contador k. Este contador k representa la suma (que es la multiplicación de las dos celdas respectivas de las matrices operando) que se va agregando a cada celda en la matriz resultante. Dentro de k tenemos el ciclo cuyo contador es `i`. Este contador `i` representa la fila de la primera matriz. Por último tenemos el ciclo más interno que es `j`. El contador `j` representa la columna de la segunda matriz y la columna de la matriz resultante.

Para entender mejor cómo funciona esta multiplicación refiérase a la siguiente imagen:



(Bryant and O'hallaron, 2015, p.31)

Como se puede ver en la imagen, en el ciclo más interno (que es el que más veces se ejecuta) se recorren los datos en fila por lo que el miss rate de la caché se reduce considerablemente.

Luego para adaptar la solución al uso de instrucciones SSE, se diseñó el siguiente pseudocódigo, en donde se aprovechan los registros `%XMM` de 128 bits para poder almacenar 4 flotantes de 32 bits consecutivos, y la capacidad de leer múltiples valores a la vez y operar sobre estos al mismo tiempo con una sola instrucción (las operaciones de suma y multiplicación son las que nos interesan), para así poder hacer que la multiplicación de matrices sea mucho más rápida.

```
quickMatrixMulSSE (float* matrixNM, float* matrixMP, float* matrixNP, unsigned n, unsigned
m, unsigned p):
1. Cree los enteros i, j, k;
2. Cree el flotante r  y asígnele 0;
3. Para k = 0 menor que m, avanzando k de 1 en 1 ejecute:
4.    Para i = 0 menor que n, avanzando i de 1 en 1 ejecute:
5.       Asígnele a r el valor de matrixNM[i*m + k];
6.       Para j = 0 menor que p, avanzando de 4 en 4 ejecute:
7.          Si p - j es menor que 4 entonces:
8.             Para j menor que p, avanzando de 1 en 1 ejecute:
9.                Asigne el próximo valor matrixNP[i*p + j] += r * matrixMP[k*p + j];
10.         Caso contrario ejecute:
11.          Asigne los próximos 4 valores matrixNP[i*p + j] += r * matrixMP[k*p + j];
```

Note que en las líneas que dicen que van de 4 en 4, se utilizan instrucciones como `movaps` (para mover 4 flotantes a un registro %xmm o hacia el arreglo después de efectuar la multiplicación y suma), `addps` (para sumar los 4 valores de 2 registros %xmm), `mulps` (para multiplicar 4 valores de 2 registros `%xmm`).

Desafíos encontrados y problemas resueltos

En general fue un desafío el aprender a usar SSE. Poderlo interpretar correctamente los resultados que se querían y tener los modelos mentales correctos. Por ello, se escribieron programas pequeños donde se veía si el resultado mostrado de las operaciones realizadas correspondía con lo esperado. Algunas de las operaciones realizadas fueron mover 4 flotantes

de un arreglo a otro, sumar y multiplicar 4 valores consecutivos de 2 registros `%xmm` entre ellos.

Otro desafío que se enfrentó fue la multiplicación de matrices con un eficiente uso de la caché. Esta multiplicación a primera vista no se entiende como calcula la multiplicación, puesto que no es natural para nosotros los humanos (el recorrido `kij`). Para entender se corrió el algoritmo a mano en primer lugar y luego se implementó en C++ donde iba imprimiendo en consola los cambios que ocurrían.

A la hora de implementar las instrucciones SSE se enfrentó ante el problema de que `P` podría no ser un múltiplo de 4. Cuando `P` era un múltiplo de 4 funcionaba, ya que las instrucciones SSE no generaban accesos inválidos a memoria porque al ser de 128 bits, son capaces de almacenar 4 flotantes consecutivos y operar sobre todos los datos e ir de 4 en 4 y por ello "finaliza correctamente".

Para resolverlo se creó el caso excepcional al que iba a entrar una vez que j (contador que iba por los números de p) resultara ser menor que p y distinto de 0. Una vez que entramos a este caso se dejan las instrucciones SSE que permite operar de 4 en 4 y se pasa a operar con las instrucciones de SSE que van de 1 en 1 (la instrucción de SSE en vez de terminación ps pasa a ser terminacion ss).

Asimismo, tuvimos un problema a la hora de realizar el salto en ensamblador, para que en el caso de que quedaran 1, 2 o 3 elementos por procesar (resultado de la condición de si p - j < 4 en el ciclo interno del algoritmo), saltara a un ciclo auxiliar para que procese esos elementos de 1 en 1 en lugar de hacerlo de 4 en 4. Por lo que los primeros intentos no saltaban a la sección de código que queríamos que saltara.

Para resolverlo, ya que nuestra condición de salto era que `p - j < 4,` donde `p - j` se encontraba en el registro `%r10d,` por lo que para hacer la comparación se hizo primero un `cmpl $4,%r10d,` y luego para la condición de salto llegamos a que nos salía más fácil hacer la inversa, por ello usamos el `jge`, ya que después de haber hecho la comparación, si `p - j` resulta ser mayor o igual que 4, entonces que saltara a la sección de código que procesa de 4 en 4, lo cual es el objetivo, y, en caso contrario, que prosiga a ejecutar el código siguiente, que corresponde al procesamiento de un elemento a la vez hasta terminar los elementos de la fila.

Otro de los problemas enfrentados es que a la hora de hacer la función en ensamblador, se usaron instrucciones como `movq, cmpq, addq` en lugar de `movl, cmpl, addl` para trabajar sobre tipos de datos de 4 bytes, lo cual generó algunos valores

indeseados, ya que se estaba trabajando con más bits de la cuenta. Además, almacenamos los valores de los índices `i, j` y `k` en el Stack, de manera continua, de forma que al usar las instrucciones (terminadas en q) que trabajan a nivel de 64 bits, se sobreescribian los valores de `i, j` y `k` en cada actualización, produciendo comportamientos adversos en el programa y errores de segmentación.

Para resolver el problema, simplemente se cambió el `q` de las instrucciones a `l`, en los casos en los que se trabajaba con tipos de datos de 4 bytes. Y además en lugar de usar los registros `%rax` por ejemplo, se usó el `%eax` de 32 bits.

Desglose de las tareas realizadas por cada integrante del grupo

- **David Xie**

➢ Implementación en C++ de las funciones `loadMatrix` y `compare`, sin manejo de excepciones.

➢ Aportes al primer entregable junto con la investigación del funcionamiento de las instrucciones SSE.

➢ Implementación del ciclo for `i, j` en ensamblador.

➢ Implementación en C++ de la función `quickMatrixMul` para tenerla como guía en su implementación en ensamblador.

➢ Aporte en la implementación de instrucciones SSE para paralelizar 4 multiplicaciones de datos.

➢ Aporte al caso especial cuando no se puede paralelizar con instrucciones SSE 4 datos de una misma fila de MP debido a que no quedan datos.

- **Juan Ignacio Pacheco**

➢ Aportes al primer entregable con investigación del funcionamiento de las instrucciones SSE. Además investigacion sobre la manera optima del uso de la caché en una multiplicación de matrices.

➢ Manejo de errores.

➢ Aporte cooperativo en lógica a la hora del uso de SSE para paralelizar 4 multiplicaciones de datos.

➢ Implementación de la sección del uso de instrucciones SSE para el procesamiento de un solo flotante en el caso de que no se puedan tomar 4 consecutivos.

➢ Implementación del ciclo for `k` en ensamblador.

➢ Cálculo del flotante `r`.

Referencias

R.E. Bryant, D.R. O'Hallaron (2015). *Cache Memories.* Recuperado el Noviembre 11 de 2020 en: https://mv1.mediacionvirtual.ucr.ac.cr/pluginfile.php/1201273/mod_resource/content/1/12-cache-memories.pdf