UNIVERSIDADE DA CORUÑA
Facultade de Informática

ARQUITECTURA DE COMPUTADORES 2021-2022

GRADO EN INGENIERÍA INFORMÁTICA

IMPLEMENTACIÓN SIMD DEL PRODUCTO MATRIZ-VECTOR

AUTOR: Sergio Piñeiro Bermúdez

A Coruña, octubre de 2021

1. INTRODUCCIÓN

En esta memoria se va a comentar las diferentes implementaciones del producto matriz-vector realizado con instrucciones SIMD. Se tomará como referencia las diferentes implementaciones realizadas en el repositorio matriz-Vector, que se adjuntará en la entrega.

En este repositorio se exponen diferentes implementaciones del producto matriz-vector, donde se emplean dos tipos de instrucciones: instrucciones SSE que emplean registros de 128 bits e instrucciones AVX que emplean registros de 256 bits. Para cada tipo de instrucciones se implementa el producto de tres maneras diferentes. En un caso, sólo se vectoriza el lazo interno del producto, empleando instrucciones hadds (sumas horizontales en el vector). En un segundo caso, se vectorizan los dos lazos utilizando también instrucciones hadds. Por último, se logra vectorizar los dos lazos sin utilizar instrucciones hadds, sino que se utilizan shuffles y otras instrucciones de ordenamiento junto con adds para realizar el producto. Cabe destacar que todas las implementaciones soportan que sean ejecutadas con cualquier tamaño de matriz y vector, ya que se emplea la técnica de padding para que esto pueda ocurrir, de tal manera que la matriz y el vector a multiplicar se escalan siempre a un tamaño múltiplo del número de floats que caben en un registro.

A continuación, se comentarán todos los casos analizándolos y comparándolos para lograr tener una comparación en función del tiempo que tarda cada implementación en realizar el producto de una matriz por un vector. Las mediciones de tiempo se realizan en un portátil MSI GV62 7RE con procesadores Intel i7-7700HQ sobre el sistema operativo Ubuntu 20.04 LTS. Cabe mencionar que los tiempos obtenidos son aproximaciones, ya que para poder comparar tiempos a gran precisión tendríamos que ejecutar numerosas veces la parte del código que se vectoriza y calcular una media de ese tiempo. Lo que sí se realizó para calcular los tiempos es la ejecución repetidas veces (en torno a 5) de cada caso, hallando posteriormente la media, pero no es comparable con el nivel de precisión que obtendríamos con el método comentado anteriormente.

2. ESTUDIO DE LAS IMPLEMENTACIONES

En primer lugar, se va a medir el tiempo que tarda en realizar un producto de una matriz de 30,000 x 30,000 elementos por un vector de 30,000 elementos con un factor alfa igual a 4, sin realizar ningún tipo de vectorización, para que sirva de punto de referencia. Ejecutando dicho producto, se obtiene que tarda en torno a 1s y 40ms.

A continuación, se ejecuta el mismo producto, pero en este caso utilizando vectorización automática. De esta manera, se obtiene que tarda en torno a 1s y 20ms. Vemos como se reduce un poco el tiempo, pero el factor de mejora es prácticamente mínimo, tratándose de 1,02 veces más rápido utilizando vectorización automática.

Obteniendo estos tiempos de referencia, se va a analizar en primer lugar las implementaciones realizadas con instrucciones SSE, comenzando por el caso en el que sólo se vectoriza el lazo interno con instrucciones haads. Lo que se hace es iterar el lazo interno de 4 en 4 de tal manera que se van obteniendo registros de 4 floats equivalentes a 4 elementos de cada columna de la matriz. Para cada registro de estos 4 elementos de la columna se cargan 4 elementos del vector, para poder multiplicarlos entre sí. Así se van obteniendo los resultados de cada fila los cuales se van guardando en un array. Con los resultados de cada fila en el array, falta sumar todos los elementos de cada registro. Para ello se van cogiendo los resultados de 2 en 2 filas y aplicando instrucciones hadds se van obteniendo los resultados. Con esta implementación, se tarda en torno a 240 ms. Vemos que ahora el tiempo se reduce considerablemente, pues esta implementación es 4,33 veces más rápida que la versión sin vectorizar y 4,25 veces más rápida que la versión de vectorización automática que realiza el compilador. La mejora respecto a la versión sin vectorizar era esperada debido a que ahora estamos utilizando paralelismo a nivel de datos que nos permite ejecutar varias multiplicaciones al mismo tiempo sobre diferentes datos. Lo sorprendente es que también superemos con creces a la vectorización automática del compilador, que se debe a que no consigue mejorar en gran medida la ejecución sin vectorización.

A continuación, se analiza el caso en el que se vectorizan los dos lazos. En este caso, también se itera de 4 en 4 el lazo externo, de tal manera que ahora sólo se cargará una vez los 4 elementos del vector para 4 registros con 4 elementos de las columnas de la matriz. Es decir, en esta implementación, se van cogiendo de 4 en 4 filas y se multiplica por los 4 elementos correspondientes del vector. Ahora si miramos el tiempo que tarda, obtenemos que tarda en torno a 235 ms. Nos encontramos con una leve mejora que tiene sentido porque antes se cargaban los 4 elementos del vector para cada 4 elementos de la columna, en cambio ahora sólo se carga una vez los 4 elementos del vector para cada 4 registros con 4 floats cada uno.

Como último caso de la implementación SSE, tenemos el caso en el que evitamos utilizar instrucciones hadds en el momento de juntar los resultados de cada fila. Para este caso lo que se realiza es una transposición de los elementos de los resultados para que podamos utilizar simplemente adds en vez de hadds. Para ello se cogen de 4 en 4 los registros resultado y mediante instrucciones unpack y shuffles se consigue obtener los elementos transpuestos que se van sumando a un registro final que contendrá las sumas individuales de cada fila. Si medimos el tiempo que tarda en ejecutarse esta implementación obtenemos un tiempo en torno a 235 ms. No obtenemos una mejora respecto a la implementación anterior ya que no supone un aumento de rendimiento el no utilizar hadds y sí realizar una transposición.

Llegado a este punto, se procede a analizar las implementaciones realizadas con instrucciones AVX. Estas instrucciones utilizan registros de 256 bits por lo que ahora podremos tener hasta 8 floats por registro. Se comienza analizando el primer caso en el que sólo se vectoriza el lazo interno y se utilizan instrucciones hadds. En este caso la diferencia primordial a la anterior implementación es manejar unos registros más grandes, lo que produce que cambie el sistema de suma de todos los elementos de los registros resultados. Pues para este caso aplicando varios hadds no obtenemos la suma completa, sino que tenemos que emplear la instrucción permute para poder provocar que se sumen los elementos correspondientes a cada fila mediante hadds. Además, tenemos que utilizar un

cast a registros de 128 bits para poder obtener las sumas finales y escribirlas en memoria. Si miramos el tiempo que tarda este caso, obtenemos un tiempo de entorno a 234 ms. Prácticamente no obtenemos una mejora respecto al caso anterior donde utilizábamos instrucciones SSE, cosa que puede parecer rara ya que ahora podemos realizar hasta 8 multiplicaciones a la vez con diferentes datos. Lo que ocurre en este caso, es que pude ser que las cachés se vean perjudicadas al utilizar unos registros más grandes, por lo que no podemos apreciar una mejora de tiempo respecto al caso anterior. Estamos pagando una penalización caché que impide mejorar los tiempos.

Pasando a la implementación en la que se vectorizan los dos lazos, en este caso ocurre algo parecido a lo que nos ocurría con las instrucciones SSE. Ahora, sólo se carga una vez los 8 elementos del vector que servirán para multiplicar con cada registro de 8 elementos de 8 filas distintas. Es decir, por cada 8 registros correspondientes a 8 elementos de 8 filas distintas sólo vamos a cargar un registro de 8 elementos del vector, mientras que antes por cada registro de 8 elementos de la matriz teníamos que cargar también los 8 elementos del vector cada vez. Esto se consigue simplemente iterando de 8 en 8 el lazo exterior. Si medimos el tiempo que tarda esta implementación, obtenemos un tiempo en torno a 232 ms. Estamos mejorando el tiempo anterior levemente ya que ahora realizamos menos cargas que anteriormente.

Por último, se analiza el último caso en el que no se utilizan instrucciones hadds. Como ahora manejamos registros más grandes, el sistema de suma de los elementos de los registros resultado se nos complica un poco más que para las instrucciones SSE. En este caso vamos cogiendo registros resultado de 8 en 8 y en grupos de 4 en 4 se va trabajando de 2 en 2 registros, de tal manera que mediante unpack y shuffles vamos a ir transponiendo los elementos de los registros de tal manera que nos vayan quedando los resultados de las multiplicaciones de la misma columna en los mismos registros, para posteriormente sumarlos con adds. Luego tenemos que emplear un cast y un permute para poder trabajar con registros de 128 bits y poder escribirlos en memoria. Si medimos el tiempo que tarda en ejecutarse esta implementación obtenemos un tiempo en torno a 232 ms. Como podemos ver, no se obtiene una mejora significativa respecto a la implementación anterior, ya que no supone un beneficio no emplear instrucciones hadds y sí realizar una transposición de los elementos.

3. CONCLUSIÓN

Con esta práctica podemos observar cómo beneficia la utilización de vectorización en estos programas donde hay que realizar numerosas operaciones sobre diferentes datos. Lo que más llama a la atención es que utilizando registros más grandes, de 256 bits, en el caso de instrucciones AVX, no se consiga una mejora respecto a utilizar registros de 128 bits con las instrucciones SSE. Probablemente si aumentásemos la capacidad de tamaño de las cachés o disminuyésemos el tiempo de lectura/escritura en los dispositivos de almacenamiento, sí notaríamos la mejora que se espera al poder realizar más operaciones al mismo tiempo. Aun así, sí queda notoriamente demostrado que la vectorización mejora, con grandes resultados, el tiempo de ejecución del producto.