Aceleración del Análisis de Series Temporales en el Procesador Intel Xeon Phi KNL

Iván Fernández[1], Alejandro Villegas[2], Eladio Gutiérrez[3] y Óscar Plata[4]

Resumen— **El análisis de series temporales es un campo de investigación de gran interés con innumerables aplicaciones. Recientemente, el método Matrix Profile, y particularmente una de sus implementaciones, el algoritmo SCRIMP, ha empezado a cobrar relevancia en este campo.**

En este trabajo analizamos la estructura y el rendimiento del algoritmo SCRIMP en el contexto de una arquitectura Intel Xeon Phi Knights Landing (KNL), que integra módulos de memoria HBM (High-Bandwidth Memory).

En este análisis combinamos diferentes técnicas para explotar el potencial de la arquitectura. Por un lado, explotamos la capacidad multihilo y vectorial de la arquitectura. Por otro lado, exploramos cómo ubicar los datos en la memoria para extraer el máximo rendimiento de la arquitectura de memoria híbrida disponible, haciendo uso tanto de la memoria 3D de alto ancho de banda como de la memoria convencional DRAM DDR4.

Palabras clave— **Serie Temporal, Detección de Anomalías, Paralelismo de Memoria Compartida, Memoria 3D de Alto Ancho de Banda, Intel Xeon Phi KNL.**

I. Introducción

El análisis de series temporales constituye una de las herramientas más importantes en la extracción de información sobre el comportamiento de fenómenos, con aplicabilidad en multitud de campos como como la genética [1], la sismografía [2], el transporte [3], la energía [4], etc.

Recientemente, *Matrix Profile* [5] se ha desarrollado como una potente herramienta de análisis de series temporales. Entre otras características, esta herramienta es capaz de detectar anomalías en una serie temporal. La figura 1 muestra una serie temporal y su Matrix Profile. Podemos observar un patrón periódico, aunque existe una anomalía entre los valores 100 y 120, aproximadamente.

El Matrix Profile de esta serie temporal devuelve valores bajos para la parte periódica de la serie y valores más altos donde aparece la anomalía.

Desde el punto de vista del rendimiento, la figura 2 muestra la intensidad aritmética observada en SCRIMP (número de operaciones aritméticas por byte de memoria accedido) frente al rendimiento medido en GFLOPS. En particular estas medidas han sido realizadas en el procesador multicore Intel Xeon Phi Knights Landing (KNL) [6] que integra memoria 3D de alto ancho de banda (tipo HBM) y módulos de memoria externos DRAM DDR4. El punto ① en la gráfica de la figura 2 representa el rendimiento de SCRIMP. Observamos cómo la baja intensidad aritmética hace que la aplicación no aproveche el potencial completo del procesador. En contraste, el rendimiento de SCRIMP está cerca del límite determinado por el ancho de banda de memoria de la plataforma, indicando que la memoria es el principal cuello de botella que limita el rendimiento de la aplicación.

Se pueden considerar dos opciones para mejorar el rendimiento de Matrix Profile. Una primera aproximación implica modificar el algoritmo para incrementar su intensidad aritmética. Esto podría mover el rendimiento al punto ② en la figura 2, pero estaría aún limitado por el ancho de banda de memoria. Si la intensidad aritmética se mejora lo suficiente, la aplicación podría incluso alcanzar el pico de rendimiento de la plataforma. No obstante, modificar el algoritmo requiere un gran esfuerzo y podría incluso no ser posible.

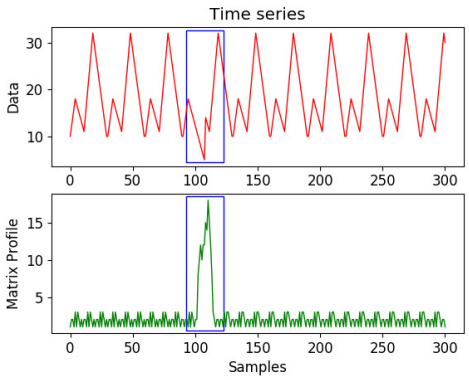

Fig. 1
Una serie con una anomalía y su Matrix Profile

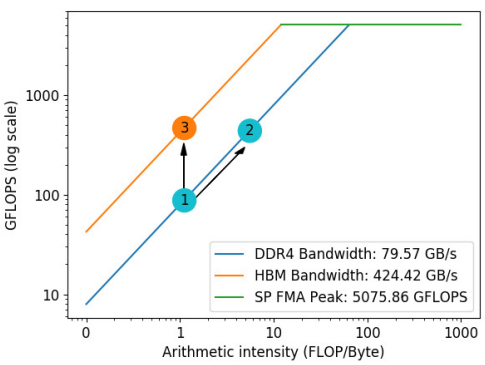

Fig. 2
Roofline del procesador KNL para SCRIMP

[1]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: ivanfv@uma.es.
[2]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: avillegas@uma.es.
[3]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: eladio@ac.uma.es.
[4]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: oplata@uma.es.

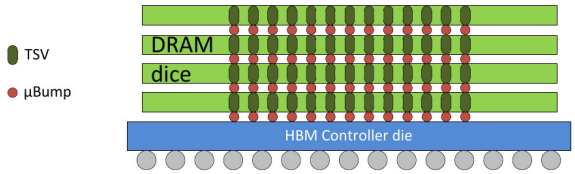

Fig. 3
Estructura de una memoria HBM

Otra aproximación consiste en explotar la memoria HBM integrada en el procesador. HBM es una inferfaz RAM de alto rendimiento para memorias 3D [7]. Como se muestra en la figura 3, esta organización de memoria consiste de varias capas de silicio conectadas verticalmente con TSVs (*Through-Silicon Vias*) y *microbumps*. Con esta organización, la memoria HBM ofrece un ancho de banda pico de unos 450 GB/s, en contraste con los 80 GB/s ofrecidos por la memoria DDR4. Usando adecuadamente la memoria HBM, combinada o no con la memoria DDR4, y asumiendo que la intensidad aritmética se mantiene inalterada, el rendimiento del algoritmo se puede llevar al punto ③ de la figura 2.

Este trabajo se centra en acelerar el algoritmo SCRIMP, cuyo cuello de botella principal es el ancho de banda de memoria, en el procesador KNL, que combina las tecnologías de memoria tipo HBM y DDR4. Nuestro análisis implica el aprovechamiento máximo de los recursos computacionales, tanto de los núcleos como de la jerarquía de memoria.

II. Antecedentes

A. Matrix Profile y SCRIMP

En esta subsección se describe sucintamente la estructura Matrix Profile [5], y el algoritmo SCRIMP [8].

Una *serie temporal* T es una secuencia de valores reales de longitud n. Llamamos t_i al i-ésimo valor de T. Una *subsecuencia* $T_{i,m}$ es un subconjunto de T empezando de la posición i y de longitud m. Dada una longitud m, un *Matrix Profile* P de una serie temporal T es un vector conteniendo la distancia entre cada subsecuencia de T y su subsecuencia más cercana (es decir, $P_i = d_{i,j}$, si $T_{j,m}$ es la subsecuencia más cercana a $T_{i,m}$).

El *Matrix Profile Index* I es un vector de índices donde $I_i = j$ si $P_i = d_{i,j}$. P contiene las mínimas distancias de las subsecuencias de T, mientras que I es un vector de índices a la localización de esas subsecuencias.

Para computar el Matrix Profile P necesitamos calcular la distancia $d_{i,j}$ para cada subsecuencia de T. Como las subsecuencias vecinas de $T_{i,m}$ son similares a esa subsecuencia, son ignoradas definiendo una zona de exclusión. La figura 4 muestra dos subsecuencias de una serie T y la zona de exclusión para una de ellas.

El objetivo del Matrix Profile es calcular la distancia para cada par de subsecuencias $T_{i,m}$ y $T_{j,m}$. Como se muestra en la figura 5, esas distancias pueden ser organizadas como una matriz.

No obstante, el tamaño de esa matriz es enorme para series temporales largas, y puede no ser posible almacenarla en memoria. Por ese motivo, los algoritmos basados en Matrix Profile solo guardan el vector de distancias mínimas P y sus índices en I.

El algoritmo SCRIMP computa la distancia euclídea $d_{i,j}$ de dos subsecuencias como:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{Q_{i,j} - \mu_i\mu_j}{m\sigma_i\sigma_j}\right)} \qquad (1)$$

En esta expresión, $Q_{i,j}$ es el producto escalar de las subsecuencias i y j, y puede ser calculada directamente, o si se conoce un producto escalar anterior, mediante la expresión $Q_{i,j} = Q_{i-1,j-1} - t_{i-1}t_{j-1} + t_{i+m-1}t_{j+m-1}$ (ver [9]). Esto implica que el cálculo de $Q_{i,j}$ crea una dependencia entre los elementos de la misma diagonal (figura 5). En este sentido, el producto escalar solo tiene que ser calculado para la primera fila (o columna) de la tabla, y los valores restantes pueden ser calculados usando los valores previos. Los símbolos μ_k y σ_k representan la media y la desviación estándar de la subsecuencia $T_{k,m}$, respectivamente, dado un tamaño de ventana m. Esos valores pueden ser calculados de antemano siguiendo la técnica introducida en [10].

La figura 6 muestra un pseudo código del algoritmo SCRIMP. Inicialmente, la media y la desviación estándar se precalculan (línea 2) además de inicializar el Matrix Profile y el vector de índices (línea 3). A continuación se calculan las diagonales (ver figura 5) (líneas 4–19).

La variable `Diagonals` es el conjunto de todas las diagonales que se necesitan para calcular P y I. Estas diagonales pueden elegirse aleatoriamente, permitiendo soluciones parciales o en orden, que permiten mayores optimizaciones [8]. Ambas opciones son exploradas en este trabajo. Cuando se procesa un elemento de la diagonal, necesitamos o bien computar el producto escalar si se trata del primer elemento (línea 7), o usar los resultados previos (línea 9). Una vez calculada la distancia (línea 11), se reemplaza la distancia previa si es menor (líneas 12–14). Como el Matrix Profile guarda la distancia $d_{i,j}$ entre dos subsecuencias $T_{i,m}$ y $T_{j,m}$, una segunda condición comprueba si $d_{j,i}$ necesita ser actualizada (líneas 15–17). Esta comprobación es necesaria porque no hay garantías de reciprocidad entre dos subsecuencias.

B. Intel Xeon Phi KNL

Como plataforma de experimentación hemos usado un SuperMicro Superserver 5038K-i [11], con un procesador Intel Xeon Phi 7210 (KNL) [12]. Esta arquitectura ofrece la combinación de tecnologías de memoria HBM con DDR4, lo que permite experimentar con la ubicación de datos en cualquiera de ellas, según sea más conveniente [13]. El procesador KNL incluye 64 núcleos Airmont (Atom) con soporte de 256 contextos hardware de ejecución (*Hyperthrea-*

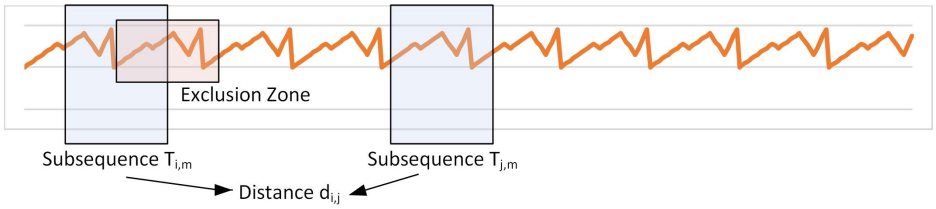

Fig. 4
Dos subsecuencias $T_{i,m}$ y $T_{i,m}$ de una serie dada T. La zona de exclusión de $T_{i,m}$ se ignora

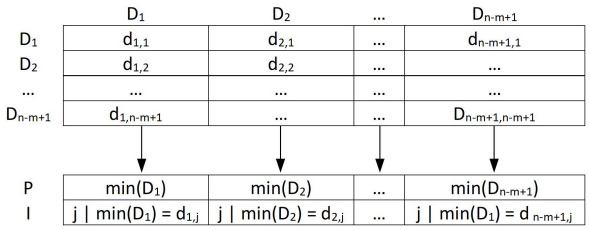

Fig. 5
Cálculo de P y sus índices I de la matriz de distancias

```
1 //Algoritmo SCRIMP
2 μ, σ = precalculateMeanSTD(T,m);
3 initialize(P, inf); initialize(I,0);
4 for k in Diagonals
5    for i=1 to length(k)
6       if i=1
7          q = dotProduct(T_{i,m},T_{diag,m});
8       else
9          q = q - t_{i-1}t_{j-1} + t_{i+m-1}t_{j+m-1};
10      endif
11      d = distance(q,μ_i,σ_i,μ_{i+k-1},σ_{i+k-1});
12      if d < P_i
13         P_i = d; I_i = i + k - 1;
14      endif
15      if d < P_{i+k-1}
16         P_{i+k-1} = d; I_{i+k-1} = i;
17      endif
18   endfor
19 endfor
20 return P,I;
```

Fig. 6
Pseudo código de SCRIMP

ding), esto es 4 contextos por núcleo. Además contiene una memoria 3D tipo HBM de 16GB integrada en el chip del procesador, ofreciendo un mayor ancho de banda que los módulos DDR4 externos. La memoria 3D se basa en DRAM multicanal (MCDRAM) y consta de cuatro bancos con un ancho de banda máximo agregado de más de 450 GB/s. La memoria DDR4, cuya capacidad es de 192 GB, proporciona 6 canales con un ancho de banda pico de 115.2 GB/s.

Se puede explotar paralelismo multihilo y vectorial, usando las extensiones SIMD con tecnología AVX-512. Estas extensiones permiten el cómputo vectorial de 16 operaciones de coma flotante de precisión simple u 8 de doble precisión, mediante dos procesadores vectoriales fuera de orden (VPU) disponibles por cada *tile*. El rendimiento máximo teórico del procesador KNL es de 6 TFLOPS en precisión simple y 3 TFLOPS en precisión doble.

III. Optimización de SCRIMP en KNL

SCRIMP es altamente paralelizable debido a que el cómputo de las diagonales se puede realizar independientemente. La forma más sencilla de acelerar el algoritmo en un sistema multi-core consiste en distribuir las diagonales entre los diferentes hilos de ejecución. No obstante, como las diagonales tienen diferente longitud, esa distribución puede llevar a un desbalanceo de carga. Una forma de solucionar este inconveniente consiste en crear una lista de diagonales para ser procesadas, y cada vez que un hilo acaba de computar una diagonal toma una nueva de la lista. De esta manera, los hilos están ocupados la mayor parte del tiempo. Puesto que existe la posibilidad de que varios hilos actualicen la misma posición del Profile, debemos establecer un mecanismo que resuelva los conflictos. En un primer lugar, desarrollamos una versión basada en operaciones atómicas, que debido al pobre rendimiento obtenido sólo mostraremos sus resultados por comparación. La solución que obtiene mejores resultados está basada en la privatización de los datos, que describimos en la siguiente subsección. También hemos realizado diversas optimizaciones en la implementación de SCRIMP para aumentar la intensidad aritmética y aprovechar las capacidades vectoriales de la plataforma.

A. Actualización de P y I

Las posibles actualizaciones concurrentes a las estructuras de datos P y I se pueden resolver o bien mediante exclusión mutua (*atomics*) o mediante privatización de los datos críticos. El rendimiento del primer enfoque está fuertemente influenciado por la sincronización. Además, aunque en la privatización la sincronización no es necesaria, necesita una cantidad extra de memoria.

Privatización del Profile y del vector de índices. Esta implementación está basada en la privatización de los accesos a las estructuras de datos compartidas. Nuestro objetivo es evitar el uso de métodos de exclusión mutua, en busca del máximo rendimiento. Esto se consigue expandiendo [14] el Profile, creando una réplica (fila) por hilo, como se muestra en la figura 7. Esto es computacionalmente seguro ya que la operación realizada por iteración es conmutativa y asociativa, es decir, es un bucle de reducción [15]. Aun siendo compartida, cada réplica puede ser vista como un almacenamiento privado durante el cálculo del Matrix Profile, de manera que cada hilo computa su Matrix Profile privado, guardando los resultados parciales en él. La única sincronización necesaria es una barrera para esperar que

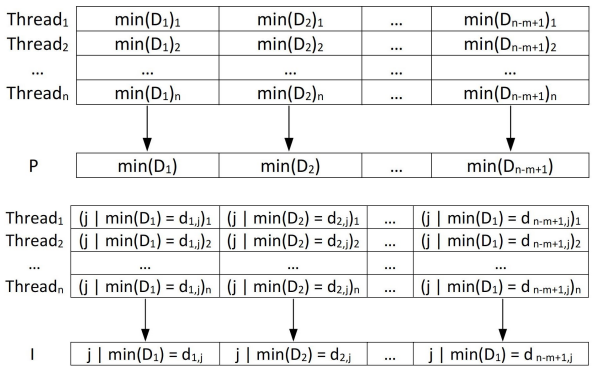

Fig. 7
Estructura expandida para P (arriba) e I (abajo)

```
1  /* Expandiendo estructuras privadas */
2  double profile_exp[nThreads][ProfileLength];
3  int profileIndex_exp[nThreads][ProfileLength];
4
5  void SCRIMP()
6  {
7    #pragma omp for schedule(dynamic)
8    for (j=0; j<numDiags; j++) {
9      for (i=j; i<diagLength; i++) {
10       /* distance is calculated here */
11       UpdateProfile(distance,i,j,threadID);
12     }
13   }
14   #pragma omp barrier
15   FinalReduction();
16 }
17
18 void UpdateProfile(distance, i, j, threadID)
19 {
20   id = threadID;
21   if (distance < profile_exp[id][j]) {
22     profile_exp[id][j] = distance;
23     profileIndex_exp[id][j] = i;
24   }
25   if (distance < profile_exp[id][i]) {
26     profile_exp[id][i] = distance;
27     profileIndex_exp[id][i] = j;
28   }
29 }
30
31 void FinalReduction()
32 {
33   #pragma omp for schedule(static)
34   for (col=0; col<ProfileLength; col++) {
35     for(row=0; row<numThreads; row++) {
36       if(profile_exp[row][col] < min_distance)
             {
37         min_distance = profile_exp[row][col];
38         min_index=profileIndex_exp[row][col];
39       }
40     }
41     profile[col] = min_distance;
42     profileIndex[col] = min_index;
43   }
44 }
```

Fig. 8
Privatización del Matrix Profile

todos los hilos acaben su computación privada. Una vez llegado a ese punto, realizamos una reducción por columnas en paralelo. Obsérvese que para el caso del vector de índices se puede realizar un procedimiento análogo (figura 7). La expansión involucra declarar dos matrices que pueden ser accedidas por los hilos: una correspondiente a las distancias y otra correspondiente a los índices, como muestra el código en la figura 8. Cada hilo actualiza las nuevas distancias en el Profile correspondientes a una fila basado en su identificador de hilo (líneas 20–28), y la reducción final se realiza en paralelo sin sincronización (líneas 33–43).

```
1  while(j < (ProfileLength - ARIT_FACT))
2  {
3    #pragma omp simd
4    for(int k=0; k<ARIT_FACT; k++) {
5        Q[k] = /* Q value based on j*/;
6    }
7
8    #pragma unroll (ARIT_FACT - 1)
9    for(int k=1; k<ARIT_FACT; k++) {
10       Q[k] += Q[k-1];
11   }
12
13   #pragma omp simd
14   for(int k=0; k<ARIT_FACT; k++) {
15       /* ξ= some calculations based on j */
16       distances[k] =  Q[k] + ξ;
17   }
18   j+=ARIT_FACT;
19 }
```

Fig. 9
Desenrollado de los bucles internos y vectorización

B. Aumento de la Intensidad Aritmética

Además de la paralelización del algoritmo, es posible reducir el tiempo de de ejecución aún más usando el soporte vectorial disponible en la plataforma KNL. Para este cometido, empaquetamos varios cálculos de distancias y actualizaciones en grupos, en vez de calcularlas individualmente. En particular, optimizamos el cálculo del producto escalar $Q_{i,j}$ de la ecuación (1). Debe tenerse en cuenta que hay dependencias de datos entre elementos consecutivos de una diagonal. Consecuentemente, esta parte de la computación será llevada a cabo en un bucle desenrollado y no vectorizado, y posteriormente usada para calcular la parte no dependiente de la computación que puede ser vectorizada dentro de otro bucle. La figura 9 muestra cómo se lleva a cabo la computación. En primer lugar, precalculamos el componente del valor que no tiene dependencias de datos (líneas 3–6) en un bucle vectorizado. Después de eso, podemos calcular los valores, teniendo en cuenta las dependencias y usando un bucle desenrollado (líneas 8–11). Finalmente, los valores de distancia derivan de los valores precalculados de nuevo en un bucle vectorizado (líneas 13–17).

Nótese que la constante *ARIT_FACT* es dependiente de la arquitectura y guía al compilador cuando genera el código máquina. En nuestro caso, esta constante está definida con un valor de 8, para aprovechar las instrucciones vectoriales de 512 bits (tenemos 8 números en coma flotante de doble precisión guardados en un mismo registro vectorial).

C. Política de Ubicación en Memoria

Con el objetivo de aprovechar el máximo ancho de banda de memoria disponible (HBM más DDR4), nuestro enfoque aloja las variables más frecuentemente accedidas en el espacio HBM, donde está disponible el mayor ancho de banda. Concretamente, HBM almacena tanto las medias y desviaciones estándar computadas para cada subsecuencia (parámetros μ y σ en la ecuación (1)), como las estructuras privatizadas de distancias mínimas e índices. No obstante, la serie original y el vector de distancias final junto a los índices se alojan en la memo-

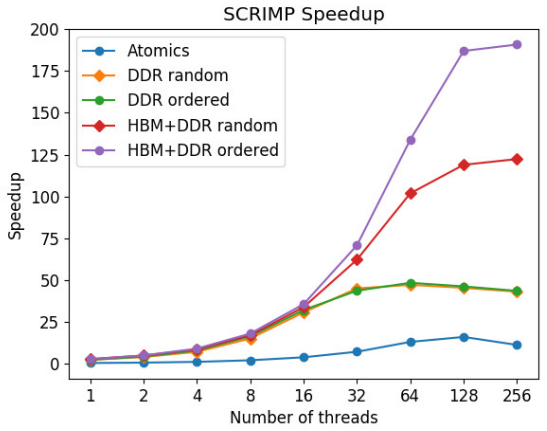

Fig. 10
Aceleración de SCRIMP en el Xeon Phi KNL

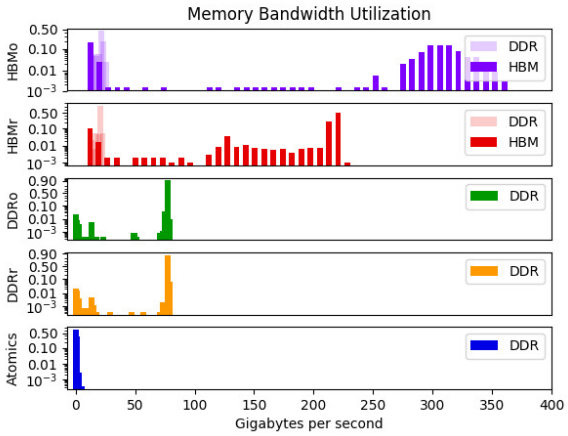

Fig. 11
Utilización del ancho de banda de memoria por SCRIMP en el Xeon Phi KNL

ria DDR4. Usando este enfoque, la memoria DDR4 sirve la serie temporal a los hilos mientras el cálculo de las diagonales es llevado a cabo (solo lecturas); después de eso, la memoria DDR4 es usada otra vez para guardar los resultados finales obtenidos en la última fase de reducción.

IV. Evaluación Experimental

A. Entorno Experimental

Los experimentos se han llevado a cabo sobre el procesador Intel Xeon Phi 7210 descrito en la sección II-B. Los códigos han sido compilados con el compilador Intel C++ v.18.0.2, con optimizaciones `-O3 -xHost` para generar el código con el mejor conjunto de instrucciones disponible (en este caso, AVX-512 cuando sea posible). Los resultados presentados están basados en el valor promedio de 10 ejecuciones.

B. Resultados de Aceleración

En primer lugar, hemos llevado a cabo los experimentos relacionados con la aceleración y uso de ancho de banda con una longitud fija tanto de serie temporal como de ventana, usando la implementación descrita en la sección III además de la versión atómica. Hemos medido el uso de ancho de banda variando el número de hilos en la implementación que nos proporciona los mejores resultados desde el punto de vista del rendimiento. Finalmente, hemos ejecutado nuestra implementación basada en en privatización de estructuras variando la longitud de la serie temporal, el tamaño de la ventana y la política de ubicación en memoria, proporcionando una comparación con los trabajos previos.

En la figura 10 se muestran los distintos resultados obtenidos. Este caso corresponde a una serie temporal de 2^{17} elementos y un tamaño de ventana de 1024.

Además de la versión atómica, hemos evaluado cuatro versiones de la versión privatizada. Por un lado, hemos probado tanto la combinación de la memoria HBM más la DDR4 como usar la memoria DDR4 solamente. Por otro lado, cuando computamos las diagonales, consideramos tanto orden aleatorio como secuencial.

La aceleración observada para estas configuraciones se muestra en la figura 10. Estableciendo un orden aleatorio da la ventaja de la propiedad *anytime*, que permite al usuario parar la computación en cualquier parte de ella, obteniendo una versión parcial (no exacta) de los resultados. La desventaja de ese enfoque es que es esperado un menor reuso de datos en las cachés, y esto explica por qué computar las diagonales en orden secuencial brinda un mejor rendimiento.

Más aún, los dos casos en los que solo se usa DDR4 (tanto orden aleatorio como secuencial) obtienen un rendimiento similar, creciendo significativamente hasta los 16 hilos y empezando a decrecer a partir de 64. A partir de ese punto, el ancho de banda DDR4 no es suficiente para servir a los hilos los datos que necesitan.

Por contra, los casos en los que se usa HBM mantienen el crecimiento con el número de hilos, y a partir de 16 hilos quedan bastante lejos de los casos con sólo DDR4. También ha de considerarse que el número de núcleos físicos disponibles es 64, y emplear más hilos implica el uso de *hyperthreading*. Este hecho resulta en una menor incremento del rendimiento si se compara con el caso de un hilo por núcleo. Obsérvese que la ejecución de un solo hilo es más rápida que la secuencial, debido al incremento en la intensidad aritmética obtenida a través del uso de las instrucciones AVX-512 en la mayor parte de los cálculos, como se explica en la sección III-B.

C. Resultados de Ancho de Banda de Memoria

El uso del ancho de banda de memoria fue medido con la herramienta Intel VTune [16]. La figura 11 muestra los tiempos de ejecución normalizados para cada uso de ancho de banda dada una serie temporal aleatoria de 2^{18} elementos, que permite resultados más precisos con esta herramienta. Hemos usado un tamaño de ventana de 1024, 256 hilos y las mismas cinco configuraciones que en la subsección anterior.

TABLA I
SCRIMP usando solo DDR4 en orden aleatorio

m	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
1024	5.65s	24.18s	119.70s	579.01s	2599.26s
2048	5.51s	23.84s	119.51s	590.99s	2615.70s
4096	5.32s	23.42s	118.71s	592.58s	2622.94s
8192	4.92s	22.54s	116.38s	586.66s	2637.59s
16384	4.21s	20.66s	111.77s	577.15s	2611.05s

TABLA II
SCRIMP usando solo DDR4 en orden secuencial

m	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
1024	5.62s	23.49s	104.26s	468.75s	2063.70s
2048	5.51s	23.29s	104.09s	472.22s	2100.94s
4096	5.32s	22.82s	103.97s	473.78s	2148.62s
8192	4.93s	21.94s	101.25s	474.86s	2088.82s
16384	4.24s	20.24s	97.98s	464.04s	2072.78s

TABLA III
SCRIMP usando HBM más DDR4 en orden aleatorio

m	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
1024	1.99s	8.38s	37.37s	153.61s	568.52s
2048	2.17s	9.22s	40.72s	169.43s	614.55s
4096	2.17s	9.35s	41.03s	173.04s	631.99s
8192	2.03s	9.07s	40.43s	173.02s	639.72s
16384	1.82s	8.48s	39.14s	167.54s	637.03s

TABLA IV
SCRIMP usando HBM más DDR4 en orden secuencial

m	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
1024	1.28s	4.88s	20.71s	88.19s	380.82s
2048	1.27s	4.86s	21.13s	90.09s	397.05s
4096	1.24s	4.88s	20.77s	90.17s	406.58s
8192	1.18s	4.79s	20.46s	91.64s	402.10s
16384	1.03s	4.48s	19.88s	90.43s	411.40s

En la versión de atómicos el ancho de banda usado es muy pequeño, puesto que la mayor parte del tiempo los hilos están compitiendo por los *locks*. Los dos casos correspondientes al uso exclusivo de la memoria DDR4 presentan un alto uso del ancho de banda teniendo en cuenta el máximo de este tipo de memoria. Este máximo no es suficiente para que el orden secuencial, al tomar las diagonales, sea más ventajoso.

Con respecto a las implementaciones basadas en la combinación de HBM con DDR4, podemos observar que, mientras un orden aleatorio de las diagonales usa un alto ancho de banda de memoria, el orden secuencial alcanza el máximo para este tipo de memoria la mayor parte del tiempo.

D. Sensibilidad a la Longitud de la Serie Temporal y el Tamaño de la Ventana

Las tablas I a IV muestran la sensibilidad de las implementaciones cuando se varía la longitud de la serie temporal y el tamaño de la ventana. Todos estos tests usan el número de hilos que proporcionan el menor rendimiento. Para llevar a cabo estos experimentos, hemos definido series temporales con tamaños representativos. En particular, usamos series temporales de 2^{17}, 2^{18}, 2^{19}, 2^{20} y 2^{21} elementos, puesto que son los valores más usados en la literatura [8], además de los tamaños de ventana más comunes (m).

La tabla I muestra los resultados de ejecutar SCRIMP con DDR4 exclusivamente y estableciendo un orden aleatorio para las diagonales. Como sugieren los resultados, no hay una clara correlación entre el tamaño de la ventana y el tiempo de ejecución, pero depende del número de elementos que caben en las cachés. En caso de que establezcamos un orden secuencial para computar las diagonales, obtenemos tiempos de ejecución similares a los del caso anterior, como muestra la tabla II. No hay beneficios significativos en el rendimiento para series temporales cortas, pero para más largas podemos obtener hasta un 25 % de mejora en los tiempos de ejecución.

También hemos evaluado la combinación de HBM con DDR4. Si usamos la memoria HBM para alojar las variables más usadas y un orden aleatorio para las diagonales, obtenemos desde un 2.8x hasta un 4.6x de aceleración con respecto al uso exclusivo de la DDR4, dependiendo del tamaño de la ventana y de la longitud de la serie temporal. Las series más largas obtienen más beneficio del uso de la HBM que las más pequeñas, como se muestra en la tabla III. Los experimentos usando el orden secuencial para las diagonales y la combinación de las memorias HBM más la DDR4 se muestran en la tabla IV. En este caso, obtenemos hasta un 58 % de mejores tiempos de ejecución que la misma configuración con orden aleatorios de las diagonales.

V. Conclusiones

En este trabajo hemos presentado una implementación eficiente del algoritmo SCRIMP de Matrix Profile en una arquitectura Xeon Phi KNL. En esta implementación explotamos los múltiples núcleos, la vectorización y el uso agregado de ancho de banda de memorias HBM y DDR4.

Hemos probado dos enfoques diferentes. En primer lugar, hemos propuesto la paralelización de las diagonales en SCRIMP, distribuyéndolas dinámicamente en los núcleos. En segundo lugar, mediante privatización reducimos la presión de sincronización entre hilos, mejorando así la escalabilidad. Además, hemos incrementado la intensidad aritmética de nuestras implementaciones usando operaciones vectoriales. Finalmente, hemos propuesto la distribución de los datos en los espacios DDR4 y HBM, alojando los datos privatizados y los más frecuentemente usados en la HBM y los datos compartidos de solo lectura en la DDR4.

Los experimentos muestran mejoras en el rendimiento en hasta 190× con respecto a la ejecución secuencial usando un Xeon Phi 7210 (KNL) de 64 núcleos. Finalmente, la implementación usando tanto HBM como DDR4 es capaz de ejecutarse hasta 5x más rápido que la solución basada solo en DDR4, probando los beneficios del uso de HBM para problemas limitados por el ancho de banda.

Agradecimientos

Este trabajo se ha realizado gracias a la financiación ofrecida por los proyectos TIN2016-80920-R del Ministerio de Economía, Industria y Competitividad, y P12-TIC-1470 de la Junta de Andalucía.

Referencias

[1] Zu-Guo Yu and Vo Anh, "Time series model based on global structure of complete genome," *Chaos, Solitons & Fractals*, vol. 12, no. 10, pp. 1827–1834, 2001.

[2] Clara E Yoon, Ossian O'Reilly, Karianne J Bergen, and Gregory C Beroza, "Earthquake detection through computationally efficient similarity search," *Science advances*, vol. 1, no. 11, pp. e1501057, 2015.

[3] Li Li, Xiaonan Su, Yi Zhang, Yuetong Lin, and Zhiheng Li, "Trend modeling for traffic time series analysis: An integrated study," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 6, pp. 3430–3439, 2015.

[4] "REFIT: Smart homes and energy demand reduction," `www.refitsmarthomes.org/index.php/data`, Accedido 2 Mayo 2019.

[5] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh, "Matrix Profile I: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *16th IEEE International Conference on Data Mining (ICDM)*, 2016, pp. 1317–1322.

[6] James Jeffers, James Reinders, and Avinash Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*, Morgan Kaufmann, 2016.

[7] Gabriel H Loh, "3D-stacked memory architectures for multi-core processors," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 453–464, 2008.

[8] Yan Zhu, Chin-Chia Michael Yeh, Zachary Zimmerman, Kaveh Kamgar, and Eamonn Keogh, "Matrix Profile XI: SCRIMP++: Time series motif discovery at interactive speeds," in *18th IEEE International Conference on Data Mining (ICDM)*, 2018.

[9] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn Keogh, "Matrix Profile II: Exploiting a novel algorithm and GPUs to break the one hundred million barrier for time series motifs and joins," in *16th IEEE International Conference on Data Mining (ICDM)*, 2016, pp. 739–748.

[10] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012, pp. 262–270.

[11] "Superserver 5038k-i specs.," `www.supermicro.com/products/system/tower/5038/SYS-5038K-I.cfm`, Accedido 2 Mayo 2019.

[12] "Intel Xeon Phi 7210 specs.," `https://ark.intel.com/products/94033`, Accedido 2 Mayo 2019.

[13] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[14] P. Feautrier, "Array expansion," in *2nd ACM International Conference on Supercomputing (ICS)*, 1988, pp. 429–441.

[15] E. Gutiérrez, O. Plata, and E. L. Zapata, "A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors," in *14th International Conference on Supercomputing (ICS)*, 2000, pp. 78–87.

[16] "Intel VTune website," `https://software.intel.com/en-us/vtune`, Accedido 2 Mayo 2019.