TESIS DE DOCTORADO

PARALLELIZATION AND OPTIMIZATION OF ITERATIVE SOLVERS ON HIGH PERFORMANCE ARCHITECTURES

Edoardo Emilio Coronado Barrientos

ESCUELA DE DOCTORADO INTERNACIONAL DE LA UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

PROGRAMA DE DOCTORADO EN INVESTIGACIÓN EN TECNOLOXÍAS DA INFORMACIÓN

SANTIAGO DE COMPOSTELA

2021

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

DECLARACIÓN DEL AUTOR DE LA TESIS

Parallelization and Optimization of Iterative Solvers on High Performance Architectures

Don Edoardo Emilio Coronado Barrientos

Presento mi tesis, siguiendo el procedimiento adecuado al Reglamento, y declaro que:

1. *La tesis abarca los resultados de la elaboración de mi trabajo.*
2. *En su caso, en la tesis se hace referencia a las colaboraciones que tuvo este trabajo.*
3. *Confirmo que la tesis no incurre en ningún tipo de plagio de otros autores ni de trabajos presentados por mí para la obtención de otros títulos.*
4. *La tesis es la versión definitiva presentada para su defensa y coincide con la versión enviada en formato electrónico.*

Y me comprometo a presentar el Compromiso Documental de Supervisión en el caso de el original no esté en la Escuela.

En Santiago de Compostela, 19 de julio 2021

Fdo. Edoardo Emilio Coronado Barrientos

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

AUTORIZACIÓN DEL DIRECTOR/TUTOR DE LA TESIS

Parallelization and Optimization of Iterative Solvers on High Performance Architectures

Don Antonio Jesús García Loureiro, Profesor Catedrático da Área de Electrónica da Universidade de Santiago de Compostela

INFORMA :

Que la presente tesis, se corresponde con el trabajo realizado por ***Don Edoardo Emilio Coronado Barrientos****, bajo mi dirección/tutorización, y autorizo su presentación, considerando que reúne los requisitos exigidos en el Reglamento de Estudios de Doctorado de la USC, y que como director/tutor de ésta no incurre en las causas de abstención establecidas en la Ley 40/2015.*

De acuerdo con lo indicado en el Reglamento de Estudios de Doctorado, declara también que la presente tesis doctoral es idónea para ser defendida en base a la modalidad de COMPENDIO DE PUBLICACIONES, en los que la participación del doctorando/a fue decisiva para su elaboración y las publicaciones se ajustan al Plan de Investigación.

En Santiago de Compostela, 19 de julio 2021

Fdo. Antonio Jesús García Loureiro

Director/a tesis

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

A la memoria de mi madre, Oralia Barrientos.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Fíate de Jehová de todo tu corazón, Y no te apoyes en tu propia prudencia. Reconócelo en todos tus caminos, Y él enderezará tus veredas.

Proverbios 3:5-6

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Agradecimientos

Durante el desarrollo de esta tesis, viví experiencias muy profundas. Algunas de ellas representaron grandes obstáculos, y otras se convirtieron en fuertes motivadores. Le agradezco principalmente a Dios Su aprobación, ayuda y protección para alcanzar esta meta.

Quiero agradecer también a mi familia por todo su apoyo, principalmente a mi padre, Emilio Coronado, por mantener su apoyo constante. Y a mi esposa Ángela Crujeiras por su paciencia y comprensión durante el desarrollo de esta tesis y su ayuda para corregir la parte en gallego de este documento.

Agradezco a mi director Antonio García por su apoyo, guía, consejo y su traducción en gallego necesarios para finalizar esta tesis. En este contexto también me gustaría agradecer al Departamento de Electrónica y al Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS) de la Universidad de Santiago de Compostela, por proveer los recursos y soporte técnico necesarios para la consecución de este proyecto. En especial a Fernando Guillén, Jorge Suárez, Diego Cougil y Rosa Hernández.

También me gustaría agradecer a todos los compañeros del CiTIUS que dedicaron parte de su tiempo para aconsejarme, corregirme, o motivarme, en especial a Guillermo Indalecio, Montserrat Fortes, Natalia Seoane, Pablo Quesada, Esteban Ferro, Alberto Suárez y Vanesa Graíño.

Deseo expresar mi agradecimiento a Adrian Jackson por aceptar ser mi anfitrión durante mi estancia en el Centro de Computación Paralela de Edimburgo (EPCC). En este contexto también me gustaría agradecer la ayuda que me proporcionó Catherine Inglis para los trámites administrativos y su guía para desplazarme en Edimburgo. A Rosa Filgueira por sus amables palabras y consejo. Y mi especial reconocimiento a Mario Antonioletti por su invaluable apoyo durante mi visita al EPCC y en la publicación de mi último trabajo.

19 de julio 2021

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Resumo

O obxectivo principal desta tese é desenvolver un formato óptimo de almacenamento en matriz dispersa e implementar kernels de computación eficientes que aceleren a execución do produto de matriz dispersa por vector (SpMV) en arquitecturas de computadoras modernas. O produto SpMV é un bloque de construción esencial para unha miríada de códigos de aplicación numérica, especialmente para solucionadores iterativos e simuladores numéricos. Mellorar o rendemento do produto SpMV é de especial interese para os investigadores, xa que é o principal pescozo de botella para os códigos onde se require. De feito, os kernels SpMV historicamente execútanse ao 10% ou menos do rendemento máximo en arquitecturas superescalares baseadas en caché [1]. De feito, optimizar o produto SpMV en arquitecturas de computadoras modernas non é unha tarefa trivial. Sobre todo, porque hoxe en día todos os computadores son paralelos [2], e aproveitar a maior parte da súa potencia de procesamento require que o programador teña coñecemento de paradigmas de programación paralelos, algoritmos paralelos eficientes e polo menos unha idea básica da arquitectura do dispositivo que se está apuntando.

No caso particular do produto SpMV, a natureza desordenada presente no patrón de dispersión das matrices dispersas xeradas por aplicacións da vida real, é o principal culpable de non ter un fluxo de control uniforme nos kernels SpMV. Os patróns de dispersidade desordenados tamén fan que os accesos consecutivos á memoria sexan difíciles de lograr polos subprocesos xerados no dispositivo acelerador. Estas dúas condicións son requisitos importantes para o procesamento paralelo eficiente de kernels en dispositivos de moitos núcleos [3]. Por tanto, reorganizar a estrutura de datos da matriz de tal maneira que esta regularidade póidase aplicar no deseño de novos formatos de almacenamento de matriz é primordial para implementar códigos numéricos eficientes nas computadoras modernas. Ademais, a implementación do kernel SpMV tamén debe ter en conta as especificacións técnicas do hardware

ao que se dirixe.

Esta tese comezou implementando versións personalizadas de solucionadores robustos de sistemas de ecuacións lineais nun simulador numérico para dispositivos semiconductores [4]. Implementáronse dous solucionadores personalizados en OpenCL [5]: o residuo mínimo xeneralizado flexible (FGMRes) [6] e o método gradiente biconxugado estabilizado precondicionado (BiCGStab precondicionado) [7]. As versións personalizadas probáronse utilizando o coprocesador Intel Xeon Phi 3120A [8] e a unidade de computación gráfica (GPU) NVIDIA Tesla S2050 [9]. Os obxectivos deste primeiro paso foron: familiarizar ao autor desta tese cun novo paradigma de programación paralela, comparar a precisión dos resultados numéricos utilizando dispositivos aceleradores para procesar fragmentos de código, e comparar o rendemento dos solucionadores personalizados coa implementación do solucionador FGMRes que se atopa na biblioteca PSPARSLIB [6]. Os resultados numéricos mostraron que existía unha pequena diferenza na precisión entre executar o código completo do simulador no servidor do sistema e executar o simulador no servidor do sistema e un dispositivo acelerador. Con todo, esta diferenza non foi significativa, e validouse a implementación de solucionadores personalizados xunto co primeiro obxectivo. Os resultados numéricos tamén mostraron que as funcións executadas no dispositivo acelerador requirían optimización porque as versións personalizadas non superaban a versión PSPARSLIB. Tamén exhibiron que o tamaño do problema requiría ser o suficientemente grande como para superar a inicialización de OpenCL e a sobrecarga do marco.

O segundo paso desta tese foi estudar o rendemento dalgunhas operacións alxébricas lineais básicas utilizando OpenCL en dispositivos aceleradores [10, 11]. O rendemento das operacións AXPY, DOT e SpMV analizouse no coprocesador Intel Xeon Phi 3120A e na GPU NVIDIA Tesla S2050. Non se fixo ningún esforzo para mellorar o rendemento da operación AXPY debido ao seu carácter xa paralelo. Implementáronse dous enfoques para a operación DOT. O primeiro enfoque cargou dous valores dun vector, calculou o seu produto e realizou unha primeira redución de dous elementos e almacenou este resultado nun bloque na memoria local. Unha vez que se enchen todas as posicións dentro do bloque local, realizouse unha segunda redución baseada no direccionamiento secuencial para evitar conflitos de banco de memoria. O segundo enfoque para a operación DOT, foi un kernel de redución de dous pasos. O primeiro kernel asignará cada elemento de traballo xerado polo dispositivo para reducir varias posicións dunha matriz de resultados intermedia. E o segundo kernel usará un elemento de traballo para reducir aínda máis os resultados parciais calculados a partir do

primeiro kernel. Para a operación SpMV realizáronse dúas implementacións de acordo co traballo desenvolvido en [12, 13]. A primeira implementación coñécese como o kernel *escalar*. Asigna un elemento de traballo para realizar o produto escalar entre unha fila da matriz dispersa e o vector da dereita. O rendemento deste kernel vese moi afectado por matrices cuxas filas teñen un enchido moi desequilibrado, xa que isto xerará desequilibrios de carga de traballo entre subprocesos. O segundo kernel SpMV coñécese como kernel *vectorial*. Este kernel asigna 32 elementos de traballo para realizar o produto escalar entre cada fila da matriz dispersa e o vector da dereita. En primeiro lugar, cada elemento de traballo, dentro do grupo de traballo, carga unha entrada da matriz e un valor do vector e realiza o seu produto, a continuación, almacena o seu resultado nunha matriz na memoria local. Por último, realízase unha redución nos elementos desta matriz mediante o direccionamiento secuencial. O rendemento deste kernel depende en gran medida do número de elementos dentro das filas da matriz dispersa. Claramente, as matrices cuxas filas están moi poboadas serán máis adecuadas para este kernel. Os resultados numéricos mostraron que os kernels que usaban memoria local beneficiaban o rendemento do dispositivo NVIDIA, mentres que os kernels que evitaban o uso de memoria local desempeñábanse mellor no coprocesador Intel. Tamén mostraron que a GPU NVIDIA era máis rápida que o dispositivo Intel nos seguintes casos:

- para a operación AXPY, arranxos de discos con menos de 1 M de elementos,
- para o produto DOT, arranxos de discos con menos de 2 M de elementos,
- para o produto SpMV, arranxos de discos con menos de 10 M de elementos.

Estes resultados centraron esta tese na mellora do produto SpMV, xa que este último resultou ser a operación máis custosa en tempo dos solucionadores iterativos comúns. Por tanto, a investigación de varios estudos de vangarda sobre formatos optimizados foi o terceiro paso. Despois dunha coidadosa consideración, seleccionáronse catro formatos para identificar as estratexias exitosas empregadas no seu deseño, ou para ser utilizadas como competidores das nosas propostas co fin de validar os resultados desta tese. O primeiro esquema considerado é o formato de fila dispersa comprimida (CSR). Este formato é un dos esquemas de almacenamento máis populares para matrices dispersas en procesadores superescalares [14]. Ata o día de hoxe, o interese polo formato CSR mantívose ata o punto de que importantes bibliotecas numéricas aínda contan con versións optimizadas do mesmo no seu repertorio. As bibliotecas Intel MKL [15] e cuSPARSE [16] son dous exemplos destas bibliotecas. O formato CSR almacena explicitamente as entradas da matriz e os índices de columna nas matrices `val[]` e

col[], respectivamente. Este formato tamén require unha terceira matriz de punteiros de fila (row[]). Se NNZ é o número de elementos distintos de cero na matriz e NROWS é o número de filas da matriz, as matrices val[] e col[] teñen lonxitude NNZ e a matriz row[] ten dimensión NROWS+1. Os principais inconvenientes deste formato son os accesos non contiguo á memoria debido ao direccionamiento indirecto e as cargas de traballo pequenas e desequilibradas entre subprocesos debido ao enchido variable das filas. O formato CSR utilizouse nesta tese para comparar o seu rendemento.

O segundo esquema considerado foi o formato ELLR-T desenvolvido para GPU [17]. O formato ELLR-T proporciona unha matriz dispersa cunha estrutura de datos regular para favorecer a computación SpMV eficiente en máquinas vectoriais [18]. Este formato utiliza dous matrices rectangulares e unha matriz lineal para almacenar a matriz dispersa. E os parámetros T e BS para axustar o formato dunha matriz dada. A matriz val[] almacena os elementos distintos de cero da matriz, a matriz col[] almacena os seus respectivos índices de columna e a matriz rl[] almacena as lonxitudes de fila da matriz. As matrices val[] e col[] son de tamaño NROWS x RMAX, onde NROWS é o número de filas da matriz e RMAX é o número máximo de elementos distintos de cero por fila na matriz. A matriz rl[] é de dimensión NROWS. As filas cuxo número de elementos distintos de cero son inferiores a RMAX énchense con ceros. O parámetro T utilízase para indicar o número de subprocesos asignados para calcular o resultado de cada fila. Por tanto, cada fila divídese en conxuntos de elementos T. O parámetro BS para indicar o número do tamaño do bloque. Este formato ten as seguintes vantaxes: acceso á memoria global fusionado e aliñado, computación homoxénea dentro dun conxunto de 32 elementos de traballo, redución de cálculos inútiles e desequilibrio dos subprocesos dun conxunto de 32 elementos de traballo e alta ocupación para os elementos de traballo. Con todo, o recheo cero é o principal inconveniente do formato ELLR-T. Dado que unha matriz que ten polo menos unha fila moi poboada dará lugar a un gran espazo de memoria, o que fai imposible almacenar algunhas matrices. Nesta tese utilizouse o formato ELLR-T para comparar o seu rendemento.

O terceiro esquema importante considerado é o formato ELL-WARP (K1) desenvolvido para GPU. Este formato combina ideas doutros esquemas [19]: varía o número de elementos almacenados (formato ELL-R [14, 20]), reduce o requisito de almacenamento (formato SELL [21]), asigna moitos elementos de traballo para equilibrar a carga de traballo (formato ELLR-T [17]) e ordena as filas da matriz (formato pJDS [22]). O formato K1 reordena as filas da matriz en orde descendente segundo o seu número de elementos. Esta reordenación realízase para

empaquetar as filas cun número similar de elementos en segmentos denominados *bloques*. O tamaño dun bloque especifica o número de filas que contén e estableceo o parámetro `BLS`. Ter as filas empaquetadas nos bloques máis homoxéneos posibles, axuda a equilibrar a carga de traballo entre os grupos de traballo. Este formato utiliza cinco matrices para almacenar a matriz de partida. A matriz `val[]` almacena as entradas da matriz, a matriz `col[]` almacena os índices de columna, a matriz `nmc[]` garda o número máximo de columnas contidas en cada bloque, a matriz `blp[]` garda os punteiros á posición inicial de cada bloque e `permi[]` mantén o mapa de permutación inversa da matriz. A dimensión das matrices `val[]` e `col[]` depende de como se forman os bloques, pero a súa dimensión está preto do valor de `NNZ`. A dimensión das matrices `nmc[]` e `blp[]` son de tamaño `BLN`, o número de bloques necesarios para almacenar a matriz. E a matriz `permi[]` é de lonxitude `NROWS`.

O cuarto esquema é o formato SELL-C-σ desenvolvido para multiplataforma [23]. Este formato é equivalente ao formato K1, pero foi desenvolto de forma independente por outro grupo [19]. O formato SELL-C-σ divide a matriz en fragmentos de filas de igual tamaño. Cada fragmento ten `C` filas. A continuación, cada fila énchese con ceros para que coincida coa lonxitude da fila máis longa dentro do mesmo fragmento. Con iso todos os elementos nun fragmento almacénanse consecutivamente en orde principal de columna. O número de filas debe completarse nun múltiplo de `C`. Este formato utiliza as mesmas matrices e dimensións do formato K1. A principal diferenza entre este formato e o formato K1 é que este último foi desenvolvido para GPU e o primeiro foi desenvolvido para multiplataforma e presentou un núcleo para o coprocesador Intel Xeon Phi. O terceiro e cuarto formatos tamén se empregaron como competidores dos formatos propostos por esta tese.

Cada un dos formatos seleccionados foi un logro na optimización do rendemento de SpMV. Certamente, pódese argumentar que dous eventos importantes motivaron aos seus deseñadores para desenvolver estes esquemas. A introdución do modelo Compute Unified Device Architecture (CUDA [24]) en 2006 por NVIDIA para os seus GPU, e a introdución das Extensións Vectoriais Avanzadas de 512 bits (AVX-512 [25]) propostas por Intel en 2013 para as súas arquitecturas Knights Landing e Skylake [26]. Con todo, como se pode apreciar na exposición anterior, a maioría destes formatos foron desenvolvidos para GPU NVIDIA destacando unha preferencia por estes dispositivos sobre os dispositivos Intel Xeon, a pesar de que ambas plataformas utilízanse en computación de alto rendemento (HPC). Esta tendencia podería explicarse porque os programadores xa estaban utilizando GPU para a informática de propósito xeral e a introdución do modelo CUDA facilitou o uso de GPU porque os proble-

mas xa non requiren ser enmascarados como tarefas de gráficos por computadora [27].

Motivada pola falta de formatos orientados á arquitectura Intel Xeon e polo deseño cada vez máis complexo dos formatos de almacenamento actuais de última xeración, esta tese propoñía dous formatos de almacenamento para matrices dispersas que apoian a seguinte hipótese:

Se un formato de matriz dispersa non é complicado (en termos de matrices necesarias para conter unha matriz) e inclúe valores vectoriais en si mesmo, debería mellorar o rendemento do produto SpMV (medido en GFLOPS), e en consecuencia dos solucionadores iterativos, en arquitecturas paralelas modernas.

Por iso, as principais achegas desta tese son as seguintes:

- A primeira proposta desta tese foi o novo formato AXC (A e X correspóndense a matriz e vector respectivamente en notación de álxebra lineal, e C significa liña de memoria caché) para realizar de maneira eficiente o produto SpMV na arquitectura Intel Xeon Phi utilizando OpenCL [28]. Este formato desenvolveuse seguindo as recomendacións que se atopan en [3]. Este traballo comparou o rendemento do produto SpMV utilizando os formatos AXC, CSR, ELLR-T e K1. Este estudo tamén incluíu a implementación dunha aplicación real utilizando os formatos anteriormente mencionados. As principais características deste traballo enuméranse a continuación:

  1. O formato AXC é un esquema moi simple. Utiliza só duas matrices para conter a matriz dispersa. A primeira matriz (`ax[]`) almacena as entradas da matriz e os seus valores vectoriais correspondentes de forma contigua en segmentos de datos denominados ladrillos. A segunda matriz (`brp[]`) garda os punteiros nas posicións iniciais de cada fila da matriz (`ax[]`).
  2. Cada grupo de datos ten unha lonxitude igual a `2 x HBRS`. `HBRS` significa medio tamaño de bloque e é igual a liña de memoria caché. Esta disposición de datos permite ao compilador explotar a utilización da memoria caché e a vectorización de rexistros de 512 bits do coprocesador Intel Xeon Phi dunha maneira altamente eficiente. Isto resolve o principal pescozo de botella de rendemento do coprocesador Intel Xeon Phi, que é a utilización da memoria caché de acordo con [28].
  3. A inclusión dos valores vectoriais na matriz `ax[]` fai que o formato AXC sexa robusto fronte aos accesos indirectos á memoria. Este feito confírmase polo

número de casos nos que o formato AXC supera aos seus competidores cun kernel OpenCL, é dicir, en 7 das 12 matrices. A maioría destas matrices teñen unha localidade espacial pobre debido aos seus patróns de dispersión aleatorios ou de punta de frecha.

4. O formato AXC probouse nunha aplicación real (solucionador CG) utilizando o enfoque de descarga. Esta proba requiría converter os formatos AXC, ELLR-T e K1 do formato CSR. Esta transformación mostrou que o formato AXC ten o tempo de conversión máis rápido debido á súa simplicidade.

5. A implementación de CG tamén requiría transferir datos entre o servidor e o dispositivo, o que aumentaba o número de operacións de memoria e obstaculizaba o rendemento de todos os solucionadores.

- A segunda proposta foi probar o formato AXC no coprocesador Intel Xeon Phi utilizando OpenMP e as instrucións Intel AVX-512 [29]. Este traballo desenvolveuse empregando o enfoque nativo, debido á falta de rendemento observada utilizando o enfoque de descarga. Este traballo comparou o rendemento de SpMV dos formatos AXC, CSR e SELL-C-σ. Neste estudo implementáronse dúas aplicacións reais: os solucionadores CG e BiCGStab. As principais características deste traballo enuméranse a continuación:

  1. O uso de funcións vectorizadas incorporadas, mellorou o rendemento do produto SpMV utilizando o formato AXC. Este feito evidenciouse polo número de casos en que o formato AXC superou aos seus competidores, é dicir, en 21 das 25 matrices.

  2. Os resultados numéricos confirmaron a resistencia de AXC aos direccionamentos indirectos de memoria, xa que logrou un factor de aceleración máximo de x6,8 para unha matriz (M07) cunha localidade espacial deficiente debido ao seu patrón de dispersión aleatoria.

  3. O enfoque nativo conduciu a menos operacións de memoria que o enfoque de descarga. Este feito provocou que o rendemento observado para os kernels SpMV trasladárase principalmente aos solucionadores CG e BiCGStab.

  4. O solucionador CG baseado en AXC superou aos seus competidores en 20 das 25 matrices. Tamén logrou un factor de aceleración máximo de x1,8 sobre a función

MKL para o formato CSR para unha matriz (M04) cunha localidade espacial pobre.

5. O solucionador BiCGStab baseado en AXC logrou un factor de aceleración de x1,9 sobre o seu competidor máis próximo para unha matriz (M07) cunha localidade espacial pobre.

- A terceira proposta foi o novo formato AXT (A e X correspóndense a matriz e vector respectivamente en notación de álxebra lineal, e T corresponde ao mosaico) para realizar eficientemente o produto SpMV nas arquitecturas Intel Xeon e NVIDIA utilizando as instrucións OpenMP e AVX-512 e CUDA respectivamente [30]. Este formato desarrollouse para ampliar o rango de aplicación do formato AXC a outras plataformas, como a plataforma NVIDIA. Os resultados nesta plataforma confirmaron o éxito deste logro. Este traballo probou o produto SpMV utilizando os formatos CSR, AXC e AXT en ambas plataformas.

  1. O novo formato AXT utiliza duas matrices para almacenar a matriz dispersa. Xa se introduciu a primeira matriz (`ax[]`) e a matriz auxiliar (`rwp[]` ou `hdr[]`) cuxa función depende da arranxo de datos xerada polo formato.
  2. O formato AXT utiliza tres parámetros para adaptarse a calquera dispositivo acelerador e optimizar o almacenamento de calquera matriz dispersa: o ancho medio do mosaico (`THW`), o modo (`MODE`) e o alto do mosaico (`TH`).
  3. O formato AXT pode xerar catro disposicións de datos diferentes, dependendo dos valores seleccionados para cada parámetro:
     a) a variante *AXTUH1* é o formato AXT con `MODE=UNC` e `TH=1`,
     b) a variante *AXTUH* é o formato AXT con `MODE=UNC` e `TH>1`,
     c) a variante *AXTCH1* é o formato AXT con `MODE=COM` e `TH=1`,
     d) a variante *AXTCH* é o formato AXT con `MODE=COM` e `TH>1`.
  4. A variante *AXTUH* foi a de mellor desempeño na plataforma Intel Xeon usando `TH=4` ou `TH=8` para a maioría dos casos. Logrou unha mellora do rendemento do 18,1% e logrou reducir a pegada de memoria un 5,1% sobre o formato AXC. En comparación co formato CSR, a variante *AXTUH* logrou unha mellora do rendemento do 44,3% e necesitou un 34,6% máis de memoria.

5. Esta variante non mostrou preferencia por un tipo específico de matriz porque logrou factores de aceleración sobresalientes sobre diferentes tipos de matrices. Para unha matriz cun patrón de dispersión de punta de frecha (M03), logrou un factor de aceleración de x2,41. Para unha matriz cun patrón de dispersión irregular e unha fila anormalmente grande (M07), alcanzou un factor de aceleración de x7,33. E para unha matriz dun patrón de dispersión diagonal de banda (M25), logrou un factor de aceleración de x2,73.
6. As variantes *AXTUH* e *AXTCH1* lograron o mellor rendemento na plataforma NVIDIA usando `TH=4` ou `TH=8` e `BS=512` ou `BS=1024` respectivamente para a maioría dos casos. A súa mellora de rendemento é inferior ao 10%, mentres que os seus requisitos de memoria representan un incremento de aproximadamente o 35% en comparación co formato CSR.
7. O formato AXT foi o de mellor desempeño para matrices con filas anormalmente grandes dentro dos seus rangos (por exemplo, M07, M20, M21 e M24). Alcanzou factores de aceleración desde x2,68 ata x378,50 para este tipo de matrices.
8. O formato AXT superou ao formato AXC na plataforma NVIDIA en todos os casos. Os factores de aceleración mínimo e máximo logrados polo formato AXT sobre o formato AXC foron x1,08 e x9,84 respectivamente. Este feito valida a extensión do rango de aplicación do formato AXT á plataforma NVIDIA.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Summary

The main objective of this thesis is to develop an optimal sparse matrix storage format and implement efficient computing kernels that accelerate the execution of the sparse matrix vector (SpMV) product on modern computer architectures. The SpMV product is an essential building brick for a myriad of numerical application codes, especially for iterative solvers and numerical simulators. Improving the performance of the SpMV product is of special interest for researchers, because it is the major bottleneck for codes where it is required. In fact, the SpMV kernels historically run at 10% or less of peak performance on cache-based superscalar architectures [1]. Indeed, optimizing the SpMV product on modern computer architectures is not a trivial task. Mostly, because nowadays all computers are parallel [2], and harnessing the most of their processing power requires that the programmer has knowledge of parallel programing paradigms, efficient parallel algorithms and a basic idea of the device architecture being targeted.

In the particular case of the SpMV product, the disordered nature present in the sparsity pattern of the sparse matrices generated by real life applications, is the major culprit for not having uniform control flow in the SpMV kernels. The disordered sparsity patterns also make consecutive memory accesses difficult to achieve by the threads spawned by the accelerator device. These two conditions are important requirements for efficient parallel processing of kernels on many-core devices [3]. Hence, rearrange the data structure of the matrix in such a way that this regularity can be enforced in the design of new matrix storage formats is paramount for implementing efficient numerical codes on modern computers. Furthermore, the SpMV kernel implementation also should take into account the technical specifications of the hardware being targeted.

This thesis began by implementing custom versions of robust solvers of systems of linear equations in a numerical simulator for semiconductor devices [4]. Two custom solvers were

implemented in OpenCL [5]: the Flexible Generalized Minimal Residual (FGMRes) [6] and the preconditioned BiConjugate Gradient Stabilized (preconditioned BiCGStab) [7] methods. The custom versions were tested using the Intel Xeon Phi 3120A coprocessor [8] and the NVIDIA Tesla S2050 Graphics Computing Unit (GPU) [9]. The objectives of this first step were: to familiarize the author of this thesis with a new parallel programming paradigm, to compare the precision of the numerical results using accelerator devices to process code snippets, and to compare the performance of the custom solvers against the implementation of the FGMRes solver found in the library PSPARSLIB [6]. The numerical results showed that there was a small difference in precision between running the complete code of the simulator on the system host and running the simulator on the system host and an accelerator device. However, this difference was not significant, and the custom solvers implementation along with the first objective were validated. The numerical results also showed that the functions executed on the accelerator device required optimization because the custom versions did not overperform the PSPARSLIB version. They also exhibited that the size of the problem required to be large enough to overcome the OpenCL initialization and framework overhead.

The second step of this thesis was to study the performance of some basic linear algebraic operations using OpenCL on accelerator devices [10, 11]. The performance of the operations AXPY, DOT and SpMV were analyzed on the Intel Xeon Phi 3120A coprocessor and on the NVIDIA Tesla S2050 GPU. There was no effort done to improve the performance of the AXPY operation due to its already parallel character. Two approaches were implemented for the DOT operation. The first approach loaded two values from a vector, calculated their product and performed a first two-element reduction and stored this result in a block on local memory. Once all the positions within the local block are filled, a second reduction were performed based on sequential addressing in order to avoid memory bank conflicts. The second approach for the DOT operation, was a two-step reduction kernel. The first kernel will assign each work-item spawned by the device to reduce several positions of an intermediate result array. And the second kernel will use one work-item to further reduce the partial results calculated from the first kernel. For the SpMV operation two implementations were done according to the work developed in [12, 13]. The first implementation is known as the *scalar* kernel. It assigns one work-item to perform the dot product between one row of the sparse matrix and the right-hand vector. The performance of this kernel is highly affected by matrices whose rows have very unbalanced population, because this will generate workload unbalances among threads. The second SpMV kernel is known as *vector* kernel. This kernel assigns 32

work-items to perform the dot product between each row of the sparse matrix and the right-hand vector. First each work-item, within the work-group, loads an entry from the matrix and a value from the vector and performs their product, then stores its result in an array on local memory. Finally, a reduction is performed on the elements of this array using sequential addressing. The performance of this kernel depends highly on the number of elements within the rows of the sparse matrix. Clearly, matrices whose rows are highly populated will be better suited for this kernel. The numerical results showed that kernels using local memory benefited the performance of the NVIDIA device while kernels avoiding local memory use performed better on the Intel coprocessor. They also showed that the NVIDIA GPU was faster than the Intel device on the following cases:

- for the AXPY operation, arrays with less than 1M of elements,
- for the DOT product, arrays with less than 2M of elements,
- for the SPMV product, arrays with less than 10M of elements.

These results focused this thesis on the improvement of the SpMV product, since the latter proved to be the most time expensive operation of common iterative solvers. Hence, researching several state-of-the-art studies on optimized formats was the third step. After careful consideration, four formats were selected for identifying successful strategies employed in their design, or to be used as competitors for our proposals in order to validate this thesis results. The first scheme considered is the Compressed Sparse Row (CSR) format. This format is one of the most popular storage schemes for sparse matrices on superscalar processors [14]. Up until this day, the interest in the CSR format has been maintained to the point that important numerical libraries still have optimized versions of it in their repertoire. The Intel MKL [15] and the cuSPARSE [16] libraries are two examples of these libraries. The CSR format explicitly stores the matrix's entries and the column indices in the `val[]` and `col[]` arrays, respectively. A third array of row pointers (`row[]`) is also required by this format. If `NNZ` is the number of nonzero elements in the matrix, and `NROWS` is the number of rows in the matrix, then the `val[]` and `col[]` arrays have length `NNZ` and the `row[]` array has dimension `NROWS+1`. The major drawbacks of this format are the non-contiguously memory accesses due to indirect addressing, and the small and unbalanced workloads among threads due to the variable population of the rows. The CSR format has been used in this thesis to compare its performance.

The second scheme considered was the ELLR-T format developed for GPUs [17]. The ELLR-T format provides a sparse matrix with a regular data structure in order to favor efficient SpMV computing on vector machines [18]. This format uses two rectangular arrays and one linear array to store the sparse matrix. And the parameters `T` and `BS` to tune the format for a given matrix. The `val[]` array stores the nonzero elements of the matrix, the `col[]` array stores their respective column indices, and the `rl[]` array stores the row lengths of the matrix. The arrays `val[]` and `col[]` are of size `NROWS x RMAX`, where `NROWS` is the number of rows of the matrix and `RMAX` is the maximum number of nonzero elements per row in the matrix. The array `rl[]` is of dimension `NROWS`. Those rows whose number of nonzero elements are inferior to `RMAX` are padded with zeros. The parameter `T` is used to indicate the number of threads assigned to compute the result of each row. Thus, each row is split in sets of `T` elements. The parameter `BS` is used to indicate the number of the block size. This format has the following advantages: coalesced and aligned global memory access, homogeneous computing within a set of 32 work-items, reduction of useless computation and unbalance of the threads of one set of 32 work-items and high occupancy for work-items. However, the zero padding is the major drawback of the ELLR-T format. Because a matrix that has at least one very populated row will lead to a large memory space, making impossible to store some matrices. The ELLR-T format has been used in this thesis to compare its performance.

The third important scheme considered is the ELL-WARP (K1) format developed for GPUs. This format combines insights from other schemes [19]: it varies the number of stored elements (ELL-R format [14, 20]), it reduces the storage requirement (SELL format [21]), assigns many work-items to balance the workload (ELLR-T format [17]), and it sorts the matrix's rows (pJDS format [22]). The K1 format reorders the matrix's rows in descending order according to their number of elements. This reordering is done to pack the rows with a similar number of elements into segments called *blocks*. The size of a block specifies the number of rows contained by it, and it is set by the parameter `BLS`. Having the rows packed in the most homogeneous blocks as possible, helps to balance the workload among work-groups. This format uses five arrays to store the matrix. The `val[]` array store the matrix's entries, the `col[]` array stores the column indices, the `nmc[]` array saves the maximum number of columns contained by each block, the `blp[]` array saves the pointers to the starting position of each block, and the `permi[]` keeps the inverse permutation map of the matrix. The dimension of the arrays `val[]` and `col[]` depends of how the blocks are formed, but their dimension are near to the value of `NNZ`. The dimension of the arrays `nmc[]` and `blp[]` are

of size BLN, the number of blocks needed to store the matrix. And the permi[] array is of length NROWS.

The fourth scheme is the SELL-C-σ format developed for multiplatform [23]. This format is equivalent to the K1 format, but it was developed independently by another group [19]. The SELL-C-σ format splits the matrix in equally sized chunks of rows. Each chunk has C rows. Then each row is padded with zeros to match the length of the longest row within the same chunk. Thereupon all elements in a chunk are stored consecutively in column-major order. The number of rows must be completed to a multiple of C. This format uses the same arrays, and dimensions, of the K1 format. The major difference between this format and the K1 format is that the latter was developed for GPUs and the former was developed for multiplatform and presented a kernel for the Intel Xeon Phi coprocessor. The third and fourth formats were also used as competitors of the formats proposed by this thesis.

Each of the formats selected was an accomplishment on the optimization of the SpMV performance. Certainly, it may be argued that two major events motivated their designers to develop these schemes. The introduction of the Compute Unified Device Architecture (CUDA [24]) model in 2006 by NVIDIA for their GPUs, and the introduction of the 512-bit Advanced Vector Extensions (AVX-512 [25]) proposed by Intel in 2013 for their Knights Landing and Skylake architectures [26]. However, as it can be appreciated from the above exposition, most of these formats were developed for NVIDIA GPUs highlighting a preference for these devices over the Intel Xeon devices, despite both platforms are used in High Performance Computing (HPC). This trend could be explained because programmers were already using GPUs for General Purpose computing and the introduction of the CUDA model facilitated the use of GPUs because problems do not longer require to be masked as computer graphics tasks [27].

Motivated by the lack of formats oriented to the Intel Xeon architecture and by the increasingly complex design of current state-of-the-art storage formats, this thesis proposed two storage formats for sparse matrices that support the following hypothesis:

If a sparse matrix format is uncomplicated (in terms of necessary arrays to contain a matrix) and includes vector values in itself, should improve the SpMV product performance (measured in GFLOPS), and consequently of iterative solvers, on modern parallel architectures.

Therefore, the main contributions of this thesis are the following:

- The first proposal of this thesis was the new AXC format (A and X stands for matrix and vector respectively in linear algebra notation, and C stands for cache memory lane) to perform efficiently the SpMV product on the Intel Xeon Phi architecture using OpenCL [31]. This format was developed following the recommendations found in [3]. This work compared the performance of the SpMV product using the AXC, CSR, ELLR-T and the K1 formats. This study also included the implementation of a real application using the formats previously mentioned. The main features of this work are listed below:

  1. The AXC format is a very simple scheme. It uses only two arrays to contain the sparse matrix. The first array (`ax[]`) stores the matrix's entries and their corresponding vector values contiguously in data segments called *bricks*. The second array (`brp[]`) saves the pointers to the starting positions of each row in the `ax[]` array.

  2. Each brick of data has a length equal to `2 x HBRS`. `HBRS` stands for half brick size and is equal to the cache memory lane. This data arrangement lets the compiler exploit the cache memory utilization and the 512-bit registers vectorization of the Intel Xeon Phi coprocessor in a highly efficient way. This solves the main bottleneck performance of the Intel Xeon Phi coprocessor, which is the cache memory utilization according to [28].

  3. The inclusion of the vector values in the `ax[]` array makes the AXC format robust against indirect memory accesses. This fact is confirmed by the number of cases where the AXC format outperforms its competitors with an OpenCL kernel, that is, in 7 out of 12 matrices. Most of these matrices have poor spatial locality due to their random or arrow-head sparsity patterns.

  4. The AXC format was tested in a real application (CG solver) using the offload approach. This test required to convert the AXC, ELLR-T and K1 formats from the CSR format. This step showed that the AXC format has the fastest conversion time due to its simplicity.

  5. The CG implementation also required to transfer data between host and device which increased the number of memory operations and hindered the performance of all the solvers.

- The second proposal was to test the AXC format on the Intel Xeon Phi coprocessor using OpenMP and the Intel AVX-512 instructions [29]. This work was developed using the native approach, because of the lack of performance observed using the offload approach. This work compared the SpMV performance of the AXC, CSR, and SELL-C-σ formats. Two real applications were implemented in this study: the CG and BiCGStab solvers. The main features of this work are listed hereunder:

    1. The use of built-in vectorized functions, improved the performance of the SpMV product using the AXC format. This fact was evidenced by the number of cases the AXC format outperformed its competitors, that is, in 21 out of 25 matrices.
    2. The numerical results confirmed the AXC resilience to memory indirections, because it achieved a maximum speedup factor of x6.8 for a matrix (M07) with poor spatial locality due to its random sparsity pattern.
    3. The native approach led to fewer memory operations than the offload approach. This fact caused that the performance observed for the SpMV kernels were mostly translated to the CG and BiCGStab solvers.
    4. The CG AXC-based solver outperformed its competitors in 20 out of 25 matrices. It also achieved a maximum speedup factor of x1.8 over the MKL function for the CSR format for a matrix (M04) with poor spatial locality.
    5. The BiCGStab AXC-based solver achieved a speedup factor of x1.9 over its closest competitor for a matrix (M07) with poor spatial locality.

- The third proposal was the new AXT format (A and X stands for matrix and vector respectively in linear algebra notation, and T stands for tiled) to perform efficiently the SpMV product on the Intel Xeon and the NVIDIA architectures using OpenMP and AVX-512 instructions and CUDA respectively [30]. This format was developed to extend the range of application of the AXC format to other platforms, such as the NVIDIA platform. The results on this platform confirmed the success of this accomplishment. This work tested the SpMV product using the CSR, AXC and AXT formats on both platforms.

    1. The new AXT format uses two arrays to store the sparse matrix. The first array was already introduced (`ax[]`) and the auxiliary array (`rwp[]` or `hdr[]`) varies its function depending on the data arrangement generated by the format.

2. The AXT format uses three parameters to adapt itself to any accelerator device and optimize the storage of any sparse matrix: the tile's half width (`THW`), the mode (`MODE`), and the tile's height (`TH`).
3. The AXT format can spawn four different data arrangements, depending on the selected values for each parameter:
   a) the *AXTUH1* variant is the AXT format with `MODE=UNC` and `TH=1`,
   b) the *AXTUH* variant is the AXT format with `MODE=UNC` and `TH>1`,
   c) the *AXTCH1* variant is the AXT format with `MODE=COM` and `TH=1`,
   d) the *AXTCH* variant is the AXT format with `MODE=COM` and `TH>1`.
4. The *AXTUH* variant was the best performer on the Intel Xeon platform using `TH=4` or `TH=8` for most cases. It achieved a performance improvement of 18.1% and it managed to reduce the memory footprint a 5.1% over the AXC format. Compared to the CSR format, the *AXTUH* variant achieved a performance improvement of 44.3% and needed 34.6% more memory.
5. This variant showed no preference for a specific type of matrix because it achieved outstanding speedup factors over different type of matrices. For a matrix with an arrowhead sparsity pattern (M03), it achieved a speedup factor of x2.41. For a matrix with an irregular sparsity pattern and an abnormally large row (M07), it achieved a speedup factor of x7.33. And for a matrix a band diagonal sparsity pattern (M25), it achieved a speedup factor of x2.73.
6. The *AXTUH* and *AXTCH1* variants achieved the best performance on the NVIDIA platform using `TH=4` or `TH=8` and `BS=512` or `BS=1024` respectively for most cases. Their performance improvement is lower than 10%, while their memory requirements represent an increment of approximately 35% compared to the CSR format.
7. The AXT format was the best performer for matrices with abnormally large rows within their ranks (e.g., M07, M20, M21 and M24). It reached speedup factors from x2.68 up to x378.50 for this type of matrices.
8. The AXT format outperformed the AXC format on the NVIDIA platform in all cases. The minimum and maximum speedup factors achieved by the AXT format over the AXC format were x1.08 and x9.84 respectively. This fact validates the extension of the range of application of the AXT format to the NVIDIA platform.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation

Nowadays, computers have become an essential tool for everyone because they can process general data at astonishing speeds. Engineers and researchers have specially profited from their ability to perform fast calculations, and deliver clear data visualization. This development has enabled them to address real problems that were too complex to be solved by analytical means. An example of this fact is found in the application of the Navier-Stokes equations. Despite they do not have an analytic solution, the discipline known as Computational Fluid Dynamics (CFD) solves them numerically for studies of the aerodynamic properties of modern aircrafts [32]. The amalgamation of computers, mathematical models and numerical methods is a new field of knowledge known as *computational mechanics*. A wide range of notable industries relies on the computational mechanics simulation capabilities to design and manufacture their high-end products [33]. Consequently, optimizing numerical simulation in modern computers is paramount for today's industrial and scientific activities.

In order to optimize a numerical simulation code (from here onwards *simulator*) it is important to understand its procedure. A simulator solves the Partial Differential Equations (PDEs) that emerge from a mathematical model of a process using a numerical method and a computer. Basically, the selected numerical method transforms the PDEs to algebraic equations. Then, the system of linear equations is solved by a linear equation solver (from here onwards *solver*) on a computer [33]. This description expose four main parts in a simulator: the mathematical model, the numerical method, the solver and the computer. The mathematical model is inherent to the problem at hand. However, each remaining part offers various

options for selection. This work focuses on the last two areas of a simulator.

Despite the numerical method is out of the scope of this work, it is worthwhile to mention that there are three extended methods for transforming PDEs to a system of linear algebraic equations: the Finite Difference Method (FDM), the Finite Element Method (FEM) and the Finite Volume Method (FVM). Being the FEM the most general and widespread method [6].

It is essential to know the characteristics of a system of linear equations before choosing its solver from the numerous algorithms available. Since a matrix associated with a system of linear equations upholds its properties in a much compacted manner, this work uses the matrix form of a system:

$$\mathbf{Ax} = \mathbf{b}, \tag{1.1}$$

where **A** is known as the *coefficient matrix*, **x** is the *unknown vector* and **b** is the *right-hand vector*. A matrix can be distinguished as: *full* or *sparse*. A full matrix has the number of non zero values in the same magnitude as the number of all its entries. In a sparse matrix the number of zero values dominates over the number of non zero values. Typically, the coefficient matrices that arise from simulators are very large and sparse [6]. An unfortunate selection of the storage scheme used to contain a sparse matrix could lead to larger memory requirements and, consequently, a larger number of null operations due to zero padding. Thus, in a simulator the storage format is fundamental for an efficient memory handling and optimum performance.

The algorithms for solving systems of linear equations (from here onwards *system*) are essential for numerical analysis, and can be classified in two wide categories: *direct methods* and *iterative methods* [34]. The direct methods require a finite number of operations to obtain the solution of the system. These methods depend on the size of the system being solved, which, despite the simplicity of their conception, make them inefficient in terms of operations and memory needed. Examples of these solvers are: Gaussian Elimination (GE), the LU factorization, the Cholesky decomposition and so on [34]. The iterative methods start with an approximated solution and perform a virtually infinite number of processes to refine it until a certain criteria is satisfied [34]. Among the advantages of these algorithms are their numerical robustness, fast implementation and easy parallelization [6]. The Conjugate Gradient (CG), the BiConjugate Gradient (BiCG), the BiConjugate Gradient Stabilised (BiCGStab) and the Generalized Minimum Residual (GMRes) methods belong to the last category. Due to the

advantages exposed above, this thesis focuses on iterative methods because they are preferred over direct methods in most simulators.

Particularly, this thesis focuses on the optimization of the CG and BiCGStab solvers. Algorithms **??**, **??** and **??** show that these solvers require to perform the Sparse Matrix Vector (SpMV) product. The SpMV product is of especial interest because it is the most time expensive operation within these algorithms [31, 29, 30]. The performance of the SpMV product is strongly related to the storage format used to contain the matrix and the code implementation of the operation [12, 13]. Hence, this work proposes two new formats and their optimal implementation on modern architecture computers.

Since all computers are now parallel [2], parallel programming has become a must in the skill set of all programmers. Parallelism is the path to optimal performance and use of modern computer architectures. Parallelism is available in many ways: vector instructions, multicore processors, manycore processors, coprocessors and Graphical Processing Units (GPUs) [2]. This thesis focuses on the optimization of iterative solvers in three distinct types of platforms: multi and manycore processors and GPUs. The relevance of these platforms is evidenced in their use by major supercomputers [35]. In order to effectively exploit the parallel capabilities of these platforms the following parallel programming standards were used for code implementation: Open Computing Language (OpenCL) [5], OpenMP [36], Compute Unified Device Architecture (CUDA) [24], and the Advanced Vector Extensions (AVX-512) Intel intrinsic instructions [25].

In summary, this work improves the performance of the SpMV product, a key operation of iterative solvers, by proposing two new storage formats and their optimal implementation in modern computer architectures. The optimization of the SpMV product impacts directly the performance of iterative solvers, whose use is well extended in numerical simulation. However, the range of application of an optimal SpMV product extends beyond numerical simulation, since it is also important in eigenvalue calculations because they are needed in shift-and-invert techniques [37].

1.2 Hypothesis and objectives

As exposed on the motivation (Section 1.1), this thesis focus on the design of the sparse matrix storage formats and the parallel implementations of the SpMV product on modern computer equipment, as a mean to improve the performance of the SpMV product. Hence the hypothesis

of this work is formulated hereunder:

If a sparse matrix format is uncomplicated (in terms of necessary arrays to contain a matrix) and includes vector values in itself, should improve the SpMV product performance (measured in GFLOPS), and consequently of iterative solvers, on modern parallel architectures.

In consequence, the previous hypothesis generates the following main objective:

- Explore different studies on optimization of simulators, or iterative solvers, on modern parallel computer architectures based mainly on proposing new sparse matrix storage formats. This will highlight: bottleneck points, successful strategies and useful tools that serve as the starting point to develop an optimal sparse matrix storage scheme for parallel architectures.

The main objective is decomposed on the following secondary objectives:

- Implement custom iterative solvers using OpenCL to test a simulator performance. There are several purposes of this objective: learn and apply the OpenCL parallel language, test the accuracy of the results using different devices, identify main bottlenecks changing the parallel paradigm, and compare the performance between the custom variants and the solver from an established library such as the PSPARSLIB. This objective is tackled in the paper:
  - E. Coronado-Barrientos, A.J. García-Loureiro, G. Indalecio and N. Seoane, Implementation of numerical methods for nanoscaled semiconductor device simulation using OpenCL, *In Proceedings of the 2015 Spanish Conference on Electron Devices*, CDE 2015, IEEE, 2015.
- Conduct a study on the performance of the basic algebraic operations that are commonly present in iterative solver algorithms on different accelerator devices. This goal aims to: identify the strengths and weaknesses of different accelerator devices, identify the most time expensive operation, explore different implementation of operations using OpenCL. The following papers address this objective:
  - E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Study of basic vector operations on Intel Xeon Phi and NVIDIA Tesla using OpenCL, *Annals of Multicore and GPU Programming*, 2(1):66–80, 2015.

  - E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Implementation and performance analysis of the AXPY, DOT, and SpMV functions on Intel Xeon Phi and NVIDIA Tesla using OpenCL, *In Second Congress on Multicore and GPU Programming*, PPMG 2015, pages 9–17, University of Extremadura, 2015.

- Develop a new sparse matrix format that requires the minimum of arrays necessary to contain a sparse matrix in order to optimize the SpMV performance on the Intel Xeon Phi coprocessor using OpenCL. And test the proposal on an iterative solver. This step intents to: compare the performance of the new proposal, the AXC format, against other formats, develop a Performance Model for the AXC format, test the offload approach for workload assignment, and test the OpenCL language as an optimization tool for the Intel Xeon Phi coprocessor. This goal is achieved in the following paper:

  - E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, AXC: A new format to perform the SpMV oriented to Intel Xeon Phi architecture in OpenCL, *Concurrency and Computation: Practice and Experience*, 31, 2018.

- Test the SpMV performance of the AXC format on the Intel Xeon Phi coprocessor using OpenMP and Intel AVX-512 vectorized instructions. Also, test the AXC format on iterative solvers. This objective seeks to: compare the performance of the new proposal, the AXC format, against other formats, test the native approach for workload assignment, and test the OpenMP and Intel AVX-512 instructions combination as an optimization option for the Intel Xeon Phi coprocessor. The following paper addresses this goal:

  - E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Improving Performance of Iterative Solvers with the AXC Format Using the Intel Xeon Phi. *The Journal of Supercomputing*, 74(6): 2823–2840, June 2018.

- Develop a new sparse matrix format that requires the minimum of arrays necessary to contain a sparse matrix in order to optimize the SpMV performance on the Intel Xeon and NVIDIA platforms using OpenMP, Intel AVX-512 instructions and CUDA respectively. This objective aims to: widen the range of utilization of the AXC format, improve the performance of the AXC format on the Intel Xeon platform, reduce the memory footprint of the AXC format. This objective is tackled in the following paper:

– E. Coronado-Barrientos, M. Antonioletti and A.J. García-Loureiro, 2021, A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA, *Advances in Engineering Software*, 156:102997, 2021.

1.3 Methodology and tools

This section provides the experimental framework for all the studies conducted to validate the sparse matrix formats proposed in this thesis. The first step in this thesis roadmap was to explore and analyse the state-of-the-art papers focused on optimizing simulators and popular iterative solvers by improving the SpMV product and the sparse matrix format. While doing this, there also were some initial tests performed using OpenCL to understand the parallel programming paradigm for Single Instruction Multiple Data (SIMD) machines, and to identify the main bottleneck on the sparse algebraic operations. Once successful strategies were identified, the new proposals were made using different parallel approaches. In order to avoid the overhead due to small workloads and to obtain statistical valid results, several iterations of each test were performed. The specific number of iterations for each test is shown in the Numerical Results section of chapters 2, 3 and 4.

The metrics used to validate the numerical results are shown in Section 1.3.1. This thesis targeted different architectures (e.g. multicore CPUs, manycore devices, GPUs and coprocessors) which are briefly described on Section 1.3.2. Different architectures, imply different parallel programing models, these are commented in Section 1.3.3. Section 1.3.4 addresses the suite of matrices used in this work. Lastly, Section 1.3.5 shows additional required tools used to process data outputs.

1.3.1 Metrics

This section describes the most important metrics used to evaluate the computational and storage efficiency of the formats proposed in this thesis.

SpMV performance

The performance of the SpMV product was calculated according to the following formula:

$$P = \frac{2 \times NNZ}{T}, \tag{1.2}$$

where NNZ is the number of non zero entries of the sparse matrix, the quantity $2 \times NNZ$ represents the total number of floating point operations needed to perform the SpMV product, T is the execution time needed to perform the operation measured in seconds, and P is the performance measured in FLOPS. The aim of the formats is to have the highest value for this metric.

Storage occupancy

The storage occupancy of a format is defined as:

$$\beta = \frac{NNZ}{SE}, \tag{1.3}$$

where NNZ was introduced in Equation 1.2, SE is the number of stored elements by the format, and β is the storage occupancy a dimensionless quantity. The aim of all formats is to achieve $\beta = 1$, which means that the format does store only the non zero elements of the matrix, avoiding any storage overhead.

1.3.2 Devices

In this thesis was used the following architectures: two multicore processors (Intel Core i7-3770, Intel Xeon Gold 6148), two manycore processors (Intel Xeon Phi 3120A and 7120P), and two GPUs (NVIDIA Tesla S2050 and V100). These devices are described in more detail down below.

Intel Xeon Phi 3120A

The Intel Xeon Phi 3120A is a coprocessor based on the Knights Corner (KNC) microarchitecture. It has 57 cores with a 512-bit vector arithmetic unit, which allows the use of the Intel AVX-512 instructions set. Each core has an L1 of 32 KBytes and an L2 of 512 KBytes cache memory. The maximum core frequency is 1.1 GHz. This coprocessor has a maximum memory of 6 Gbytes and 240 GBytes/s bandwidth [8]. It can be programmed with OpenMP and OpenCL parallel models. Since the 3120A admits fused multiply–add (FMA) operations, the theoretical peak performance can be calculated as: 2 operations x 57 cores x 8 (SIMD length) x 1.1 GHz = 1003 GFLOPS.

Intel Xeon Phi 7120P

The Intel Xeon Phi 7120P coprocessor shares the same technology and features of the 3120A counterpart. However, the 7120P coprocessor has: 61 cores running at a maximum frequency of 1.33 GHz, a maximum memory of 16 Gbytes and a bandwidth of 352 GBytes/s [38]. Its theoretical peak performance is: 2 operations x 61 cores x 8 (SIMD length) x 1.33 GHz = 1298 GFLOPS.

Intel Core i7-3770

The Intel Core i7-3770 is a desktop multicore processor. It has 4 cores running at at a maximum frequency of 3.90 GHz. Each core can run up 2 hardware threads for a total of 8 threads. The maximum memory is 32 GBytes type DDR running at 1333 MHz with a maximum bandwidth of 25.6 GBytes/s [39]. This processor allows OpenCL, OpenMP and AVX instructions programming. Its peak performance is: 2 x 8 x 8 x 3.9 = 499 GFLOPS.

Intel Xeon Gold 6148

The Intel Xeon Gold 6148 is a processor based on the Skylake microarchitecture. It has 20 cores running at a maximum frequency of 3.70 GHz. Each core can support 2 hardware threads for a total of 40 threads per processor. The maximum memory is 768 GBytes type DDR4 at 2666 MHz with a maximum bandwidth of 119.21 GBytes/s [40]. This processor can be programmed using OpenCL or OpenMP. Its theoretical peak performance is: $2 \times 20 \times 8 \times 3.7$ = 1184 GFLOPS.

NVIDIA Tesla GPU S2050

The NVIDIA Tesla S2050 is a GPU based on the Fermi architecture. This GPU has a total of 448 cores running at a maximum frequency of 574 MHz. The maximum memory is 3 GBytes and it also has 148 GBytes/s bandwidth. The peak performance is 514 GFLOPS using double precision arithmetic. This GPU can be programmed using: DirectX, OpenGL, OpenCL, Vulkan and CUDA [9].

NVIDIA Tesla V100 PCIe GPU

The NVIDIA Tesla V100 PCIe GPU is based on the Volta architecture. It has 5,120 CUDA cores running at 1,230 GHz. This GPU has 32 GBytes of memory and 897 GBytes/s of

bandwidth. It has a peak performance of 7 TFLOPS using double precision. This GPU shares the same programming languages of the S2050 GPU: DirectX, OpenGL, OpenCL, Vulkan and CUDA [41].

1.3.3 Parallel programming models

This section briefly describes the three programming models used in this thesis. There are two ways in how an accelerator device can be used: the *native* and the *offload* approaches. The native approach allows the accelerator device to behave as a shared memory processor that does not require special instructions to transfer data since all information is already contained on it. The offload approach sees the accelerator device as an auxiliary extern processor where some parts of the main program can be offloaded for execution. OpenMP was used to program the Intel Xeon Phi 7120P and the Intel Xeon Gold 6148 processors with the native approach. OpenCL was used to program the Intel Xeon Phi 3120A, the Intel Xeon Phi 7120P processors and the NVIDIA S2050 GPU with the offload approach. Lastly the NVIDIA V100 GPU was programmed in CUDA.

OpenMP

OpenMP is an Application Programming Interface (API) comprised of a set of compiler directives and a library of subroutines used to describe parallelism to applications. OpenMP is designed for shared memory multiprocessors, that is, devices where all processors are able to directly access all the memory in the machine. There are two main benefits for parallel programming using directives: the first is that directives can be treated as comments and ignored for language translators where the API is not installed. The second is that it allows an incremental approach to parallelism starting from a sequential code. The directives of OpenMP are offered for inclusion in the Fortran, C and C++ languages. OpenMP extensions fall into one of three categories: control structures, data environment, and synchronization constructs [42].

The control structures modify the flow of control in a program. OpenMP uses the basic fork/join model (Figure 1.1), and the control structures are in charge of generating (fork) new threads or change the execution control to another thread. OpenMP offers two directives for controlling parallelism: *parallel* and *do*. The parallel directive encloses a block of code and creates a set of threads to execute this block concurrently. The do directive splits the iterations of a loop among multiple concurrent threads [42].

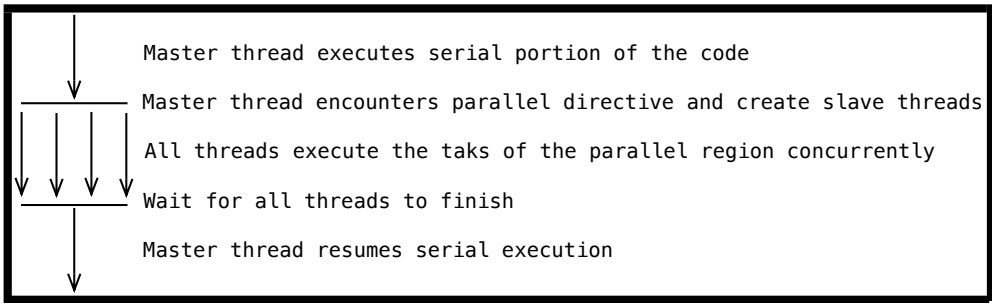

Figure 1.1: OpenMP basic execution fork/join model.

The data environment constructs are for thread communication. OpenMP executes a program with an initial thread of control associated to a data environment. This initial thread is referred as *master thread*. When a master thread encounters a parallel construct new threads and data environments are created. Each thread has its own stack for subroutines. Thus, threads within a parallel region can invoke and execute subroutines safely without interfering in other stacks. Threads in a parallel region can share variables or have their own private copies. For this purpose, a variable can have one of three attributes: *shared*, *private* or *reduce*. A shared variable has a single storage location in memory and can be accessed for all threads in the parallel region. A private variable has multiple locations in memory, one within each data environment of each thread in the parallel region. The attribute reduce is used for variables that are the result of a reduction operation, hence a reduce variable has shared and private storage behaviours [42].

The synchronization constructs are designed to coordinate the execution of multiple threads. There are two ways of synchronization: the *critical* directive and the *barriers*. The critical directive ensures that only one thread access exclusively a shared variable. On the other hand, barriers set waiting points where all threads should arrive, before continuing the execution stream [42].

The construction and initialization of a parallel region add extra runtime. This extra runtime is called *parallel overhead*. OpenMP has the *scheduling* mechanism to reduce the parallel overhead by improving the workload distribution among threads. A loop can have one of four schedule types: *static*, *dynamic*, *guided*, or *runtime*. The static loop schedule assigns a *chunk* of iterations to each thread. The dynamic loop schedule assigns chunks of iterations dynamically among threads at runtime. The guided type, sets the first chunk size to a fixed

value, then the size of each successive chunk decreases exponentially, down to a minimum size of chunk. The runtime schedule enables the selection of the scheduling type at runtime based on the value of the variable `omp_schedule` [42].

OpenCL

Nowadays, HPC equipments are conformed by devices of different architectures that have little similarity between them. The OpenCL language was developed as a common interface to avoid developers having to learn multiple languages to program different architectures. OpenCL has a set of data types, data structures and functions to augment C and C++. OpenCL has three main advantages over other parallel APIs: *portability*, *standardized vector processing* and *parallel programming* [43].

OpenCL has the philosophy "write once, run on everything". This means that every written OpenCL routine can be compiled and run on any compliant device, whether it is a GPU or multi/many core processor [43].

Most of modern processors have vector processing capabilities. However, ANSI C/C++ does not define a basic vector data type. This, generates a problem, vector instructions are vendor specific. For example, Intel has its AVX instructions and NVIDIA requires PTX instructions. OpenCL lets the user code vector routines and the compiler will produce the right platform instructions [43].

Parallel programming distributes tasks among several processing elements to be performed simultaneously. In OpenCL terms, these tasks are called *kernels*. Kernels are sent to OpenCL compliant devices by *host applications*. A host application is a regular C/C++ *program* running on the user system or *host*. Host applications manage connected devices through a container called *context*. A kernel is a function selected from a program. In order to process a kernel, this is dispatched, along with argument data, to a structure called *command queue*. A host tells a device what to do though the command queue. When a kernel is enqueued the corresponding device will execute it (Figure 1.2).

OpenCL can configure different devices to process different tasks on different data. This means that OpenCL provides full *task-parallelism*, while other parallel APIs only enable *data-parallelism* [43].

The device and memory models are other important elements of the OpenCL framework, these have the following components:

- **Compute unit**: a processing core contained within a device.

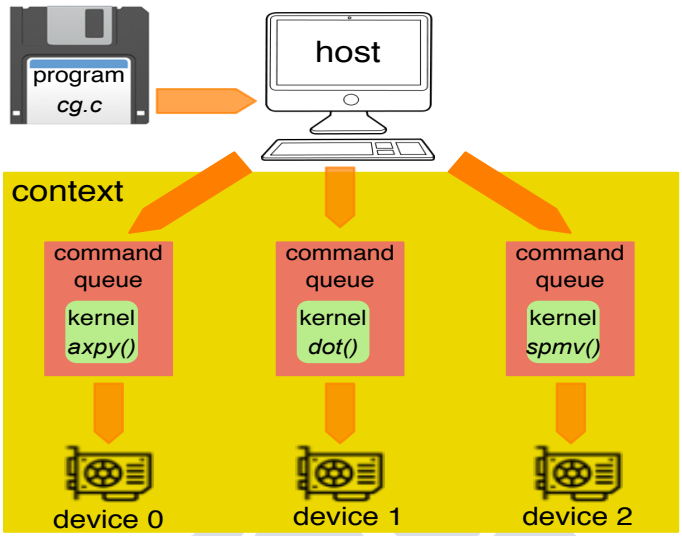

Figure 1.2: OpenCL framework.

- **Work-item**: is an individual kernel execution with a specific set of data.
- **Work-group**: is a set of work-items that access the same processing resources (e.g. local memory).
- **Global memory**: write/read memory that stores data for the entire device.
- **Constant memory**: read memory that stores data for the entire device.
- **Local memory**: stores data for work-items in a work-group.
- **Private memory**: stores data for a single work-item.

The Figure 1.3 shows how these components are related among themselves.

The global memory is the largest and slowest memory region on the device. Data is transferred directly between the host memory and the device global memory. The constant memory has the same characteristics of the global memory, except it has read-only access. The local memory is faster and commonly much smaller than global memory. Local memory can be accessed by all work-items in a work-group. Finally, the private memory is the fastest and smallest memory region, it can be only accessed by individuals work-items [43].

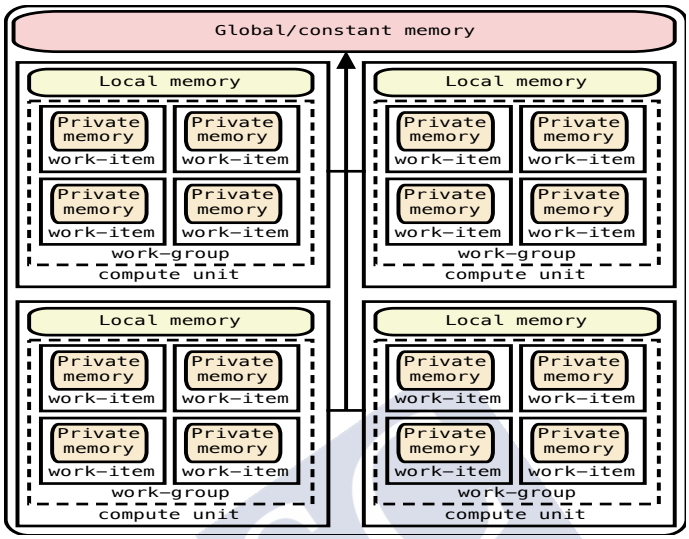

Figure 1.3: Schematic representation of the OpenCL device and memory models.

CUDA

The CUDA architecture was introduced by NVIDIA in 2006 with its GeForce 8800 GTX GPU. NVIDIA intended this new architecture to let its GPUs to perform general purpose computing. One of the advantages of CUDA was that developers no longer required to mask their problems as graphic tasks in order to be processed in a NVIDIA GPU [27].

The CUDA programing model is very similar to the OpenCL model. It lets the user to run applications in heterogenous systems by marking code snipets with a set of C extensions. As in OpenCL, a *host* is a CPU and its memory and a *device* is a GPU and its memory. An important remark is that starting with CUDA 6, NVIDIA introduced the *unified memory* model which bridges the gap between the host and device memory spaces [44]. A key component of CUDA is the *kernel*. The kernel is the code to be run on the device. It can be written as a sequencial program. All code can be put in a single file, but the host code is written in ANSI C and the device code is written in CUDA C. The `nvcc` compiler will generate the executable code for the host and the device.

When a kernel is launched from the host to the device, a large number of *threads* are generated. The organization of threads is very important to a kernel's performance. All threads generated by a kernel are called *grid*. All threads in a grid share the global memory space. A

grid is made of *blocks* of threads. All threads in a block can cooperate with each other using: block local synchronization and/or block local shared memory. CUDA organizes grids and blocks in three dimensions [44]. Figure 1.4 shows an example of a thread structure of a 2D grid containing 2D blocks.

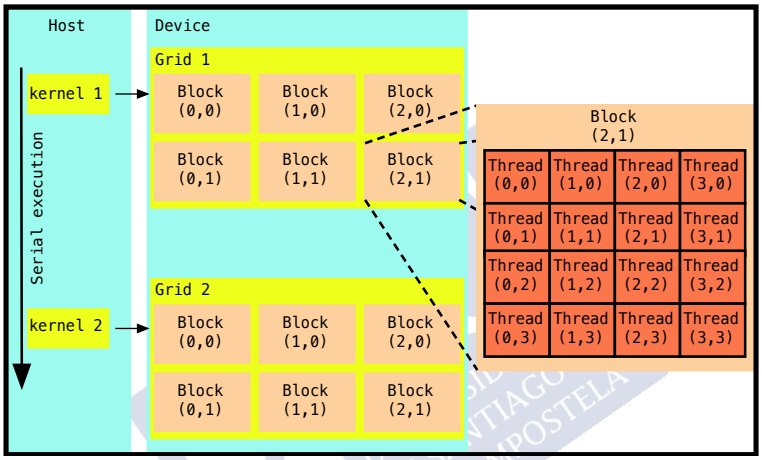

Figure 1.4: Schematic of a CUDA programming elements.

The GPU architecture is composed of a scalable array of *Streaming Multiprocessors* (SMs). Each SM have 32 fundamental compute units called *cores* (Figure 1.5). This enables a single SM to run hundreds of threads concurrently, thus a single GPU can run thousands of threads at the same time. Once a kernel is launched, the threads within a block run concurrently in the assigned SM. Several blocks can be assigned to a SM depending on its resources availability. CUDA uses the Single Instruction Multiple Thread (SIMT) parallel model to run threads in groups of 32 threads known as *warps* [45]. All threads in a warp excute an instruction concurrently. The SIMT model is similar to the Single Instruction Multiple Data (SIMD) parallel model. Both models broadcast an instruction to multiple computing units. The difference lies in that the SIMD requires that all components in a vector execute together, whereas the SIMT allows that the threads belonging to the same warp execute independently [44].

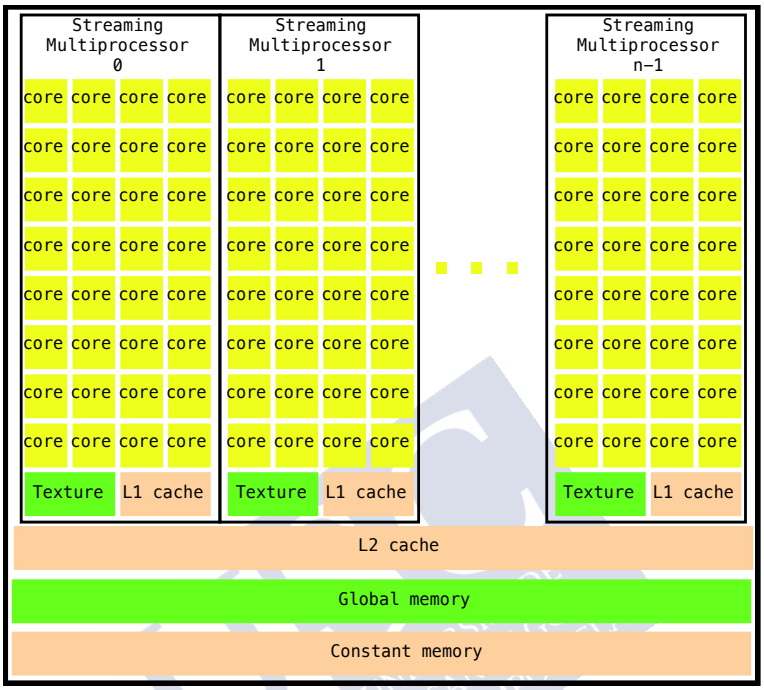

Figure 1.5: NVIDIA general GPU with n SMs.

Figure 1.5 also shows the CUDA memory model, these concepts are briefly described hereunder:

- **Global memory**: is the largest and slowest memory region. It is located on device and its space can be accessed for any SM.
- **Local memory**: resides in the same region as global memory, so it has the same characteristics. Variables that do not fit in the registers are placed in this memory region.
- **Shared memory**: is used as the CPU L1 cache, so it is faster than global memory, but it is programmable. Shared memory can be accessed by all the cores within the same SM.
- **Constant memory**: is statically declared read-only memory space. It and can be accesed by all kernels in the same compute unit. It performs best when all threads in a

warp read from the same memory address, which make it perfect to declare constant coefficients for a formula.

- **Texture memory**: is a read-only memory space that is designed to perform best for 2D spatial locality. If threads in a warp use this memory space for other than 2D data, texture memory can be slower than global memory.

1.3.4 Matrices

Tables **??**, **??** and **??** show the matrices, and their characteristics, used in this thesis. Most of these matrices belong to Williams, Boeing and Janna collections that can be consulted in [46]. Matrices E001, E002, E003, E004 and E005 are generated by a 3D FEM semiconductor device simulator that uses domain decomposition as one of their strategies for parallel execution. All matrices are squared, that is, the matrices have the same number of rows and columns. Since the formats proposed in this work focus on general matrices, the selected matrices have different sparsity patterns. There are two main structured sparsity patterns identified in these matrices: banded matrices and arrowhead matrices. Banded matrices are sparse matrices whose non zero entries are confined to a diagonal band (e.g cant, mac_econ_fwd500, mc2depi). Arrowhead matrices are sparse matrices containing zeros in all entries except for the last column, last row and main diagonal (e.g. E001, E002, E003). Remaining matrices that can not be categorized in the those categories are considered to have a random sparsity pattern (e.g. scircuit, webbase_1M, fullChip).

1.3.5 Other tools

All tests were coded using the C language under the Linux Operating System (OS). Most of these tests were compiled with the GNU Compiler Collection (gcc), except those tests where the Intel Xeon architecture was used with the native approach, in those tests the Intel Compiler Classic (icc) was used. Tests involving CUDA were compiled using the NVIDIA CUDA Compiler (nvcc). The output from these tests generated a large amount that required efficient handling. For this purpose, and because of its powerful data manipulation features python [47] was used to read and visualize data. Specifically, the matplotlib [48] package was used for the creation of the numerical results graphs exposed in Chapters 2, 3 and 4.

1.4 Work summary

As it was mentioned in Section 1.1, numerical simulation is a very versatile tool whose application is well extended in industrial and scientific areas. Anyone with access to a desktop computer can perform a numerical simulation analysis of problems that do not require large computational resources. For large enough problems, Supercomputer Centers deliver large computational resources in order to run *ad-hoc* solutions. Moreover, they also have numerical simulation software installed in order to expand their offer to the industrial and scientific communities. At its core, any simulator uses a system of linear equations solver to calculate a solution for the problem at hand. As it was also mentioned in Section 1.1, most system solvers employ iterative algorithms due to their superior numerical robustness, inferior memory requirements, and easy parallel implementation features. An iterative solver is composed of a sequence of linear algebra operations. Commonly, the AXPY ($a\mathbf{x}+\mathbf{y}$), DOT ($\mathbf{x}\cdot\mathbf{x}$) and SpMV (Equation 1.1) operations can be found within an iterative solver algorithm. From these operations, the most time-consuming operation is the SpMV product. In fact, the SpMV kernels historically run at 10% or less of peak performance on cache-based superscalar architectures [1]. Thus, the SpMV becomes one the most difficult operations to optimize in modern computers, and the focus of this thesis. Since all modern computers are parallel now [2], it is essential to exploit their parallel capabilities in order to harness most of their processing power. Efficient parallel processing requires data to be arranged in regular patterns that generates regular execution paths and regular memory accesses. As a consequence, the performance of the SpMV product is strongly related to the sparse matrix storage format used to contain the matrix, and it is also related to the implementation of the computing kernel [12, 13]. Hence, the present summary intends to provide the roadmap followed towards the optimization of the SpMV product, and consequently of iterative solvers.

1.4.1 Iterative solvers in OpenCL for semiconductor simulation

The first step, in this thesis roadmap, was a performance comparison of a semiconductor device simulator running on three different computer equipment and using different versions of iterative solvers [4].

3D FEM Simulator

The simulator employed for this comparison, is a 3D Finite Element Method (FEM) simulation tool for nano scaled semiconductor devices [4]. This simulator was developed to satisfy the accuracy and efficiency required to simulate MOSFET transistors in the nanometer regime due to the aggressive scaling, and the use of new architectures and materials. The simulator includes the Drift-Diffusion (DD) model for the carrier transport and the Density-Gradient (DG) approach to include quantum corrections. The mathematical model of the physical phenomenon is described in the equations (1.4) - (1.8), where: equation (1.4) is Poisson's equation for the electrostatic potential, equations (1.5) - (1.6) are the density gradient equations and equations (1.7) - (1.8) are the current density equations for electrons and holes with the regeneration factor (R),

$$\begin{aligned}
\operatorname{div}(\varepsilon\nabla\phi) &= q(p-n+N_D^+ + N_A^-) && (1.4)\\
2b_n\frac{\nabla^2\sqrt{n}}{\sqrt{n}} &= \phi_n - \phi + \frac{k_B T}{q}\ln\left(\frac{n}{n_i}\right) && (1.5)\\
2b_p\frac{\nabla^2\sqrt{p}}{\sqrt{p}} &= \phi - \phi_p + \frac{k_B T}{q}\ln\left(\frac{p}{p_i}\right) && (1.6)\\
\operatorname{div}(J_n) &= qR && (1.7)\\
\operatorname{div}(J_p) &= -qR && (1.8)
\end{aligned}$$

where the unknown variables are: ϕ is the electric potential, ϕ_n and ϕ_p are the quasi-potentials, and n and p the concentrations.

The simulation process is described next:

1. The physical problem is described by a mathematical model.
2. The mathematical model is reformulated as a variational problem.
3. The variational problem is discretized using the Finite Element Method.
4. The generated systems of nonlinear equations, generated by the discretization, are linearized by the Newton method.
5. The systems of linear equations are solved using an iterative solver.
6. The calculated numerical results are ready for their analysis and visualization.

Once the system of linear equations is generated in step 5, then it is solved using three different versions of iterative solvers. In order to target many core processors, the custom implementations of the solvers were written in OpenCL. These custom solvers gave place to two modified versions of the simulator ready for heterogeneous computing.

OpenCL programming model

The OpenCL programming model was introduced in Section 1.3.3.

Tests and results

This section describes the testing framework for the modified versions of the simulator and shows the numerical results obtained. The simulator was run on three different architectures: the Intel Core i7-3770 multicore processor (Section 1.3.2), the Intel Xeon Phi 3120A coprocessor (Section 1.3.2) and the NVIDIA Tesla S2050 GPU (Section 1.3.2). Two iterative solvers were selected to be implemented in the 3D FEM simulator: the FGMRes (Algorithm 1) [6], and the Preconditioned BiCGStab (Algorithm 2) [7]. A tested version of the 3D FEM simulator using the FGMRes solver from the PSPARSLIB [6], was used as reference to compare the performance of the custom solvers coded in OpenCL. The reference version ran entirely on the Intel Core i7-3770 multicore processor. The custom versions of the simulator used the Intel coprocessor and the NVIDIA GPU to offload highly intensive computing tasks such as the calculation of the norm of a vector, the dot product between vectors or the sparse matrix vector product. Low intensive tasks such as solving the overdetermined system generated in step 17 of Algorithm 1 were done on the host. The preconditioner matrix required by both algorithms (step 6 in Algorithm 1 and steps 12 and 17 in Algorithm 2) was generated using the incomplete LU factorization.

In order to validate the numerical results, the following metrics were collected from the executions of all versions of the simulator:

- **ite**: the number of iterations needed to achieve convergence.
- **nrm_res**: the norm of the residual vector:
- **t_sol**: time spent during the execution of the solver.
- **t_preAlg**: time spent during the execution of tasks for setting the OpenCL framework, such as the context creation and command queues association with devices.

- **t_alg**: total time from start to end of the algorithm.
- **t_tra**: time spent on data transfers between host and devices.
- **t_dev**: time spent exclusively in the execution of the OpenCL kernels.

Algorithm 1: Flexible Generalized Minimal Residual

PSEUDOCODE

1 $\mathbf{w}_0 = \mathbf{A}\mathbf{x}_0$
2 $\mathbf{r}_0 = -\mathbf{w}_0 + \mathbf{b}$
3 $\beta = ||\mathbf{r}_0||_2$
4 $\mathbf{v}_1 = \frac{\mathbf{r}_0}{\beta}$
5 **for** $j = 1, ...,$ **to** m **do**
6 $\quad \mathbf{z}_j = \mathbf{M}^{-1}\mathbf{v}_j$
7 $\quad \mathbf{w} = \mathbf{A}\mathbf{z}_j$
8 $\quad$ **for** $j = 1, ...,$ **to** m **do**
9 $\quad\quad h_{i,j} = (\mathbf{w} \cdot \mathbf{v}_i)$
10 $\quad\quad \mathbf{w} = -h_{i,j}\mathbf{v}_i + \mathbf{w}$
11 $\quad$ **end**
12 $\quad h_{j+1,j} = ||\mathbf{w}||_2$
13 $\quad \mathbf{v}_{j+1} = \frac{\mathbf{w}}{h_{j+1,j}}$
14 $\quad$ define $\mathbf{Z}_m = \{\mathbf{z}_1, ..., \mathbf{z}_m\}$
15 $\quad$ define $\mathbf{H}_m = \{h_{i,j}\}, i \in [1, j], j \in [1, m]$
16 **end**
17 $\mathbf{y}_m = argmin_y ||\beta\mathbf{e}_1 - \mathbf{H}_m\mathbf{y}||_2$
18 $\mathbf{x}_m = \mathbf{x}_0 - \mathbf{Z}_m\mathbf{y}$ on HOST
19 **if** *satisfied* **then**
20 $\quad$ stop
21 **else**
22 $\quad \mathbf{x}_0 \leftarrow \mathbf{x}_m$ and GOTO 2
23 **end**

Algorithm 2: Preconditioned BiConjugate Gradient Stabilised

PSEUDOCODE

$\mathbf{w}_0 = \mathbf{A}\mathbf{x}_0$
$\mathbf{r}_0 = -\mathbf{w}_0 + \mathbf{b}$
$\delta_0 = ||\mathbf{r}_0||_2$
eps $= \varepsilon * \delta_0$
$\mathbf{v}_0 = \mathbf{p}_0 = \mathbf{0}$
$\hat{\mathbf{r}} = \mathbf{r}_0$
$\rho_0 = \alpha_0 = \omega_0 = 1$
for $i = 1, ...,$ **to** *maxI* **do**
    $\rho_{i-1} = (\hat{\mathbf{r}} \cdot \mathbf{r}_{i-1})$
    $\beta_{i-1} = \left(\frac{\rho_{i-1}}{\rho_{i-2}}\right)\left(\frac{\alpha_{i-1}}{\omega_{i-1}}\right)$
    $\mathbf{p}_i = \beta_{i-1}\mathbf{p}_{i-1} - \beta_{i-1}\omega_{i-1}\mathbf{v}_{i-1} + \mathbf{r}_{i-1}$
    $\hat{\mathbf{p}} = \mathbf{M}^{-1}\mathbf{p}_i$
    $\mathbf{v}_i = \mathbf{A}\hat{\mathbf{p}}$
    $\sigma_i = (\hat{\mathbf{r}} \cdot \mathbf{v}_i)$
    $\alpha_i = \frac{\rho_{i-1}}{\sigma_i}$
    $\mathbf{s} = -\alpha_i\mathbf{v}_i + \mathbf{r}_{i-1}$
    $\hat{\mathbf{s}} = \mathbf{M}^{-1}\mathbf{s}$
    $\mathbf{t} = \mathbf{A}\hat{\mathbf{s}}$
    $\omega_i = \frac{(\mathbf{t}\cdot\mathbf{s})}{(\mathbf{t}\cdot\mathbf{t})}$
    $\mathbf{x}_i = \alpha_i\hat{\mathbf{p}} + \omega_i\hat{\mathbf{s}} + \mathbf{x}_{i-1}$
    $\mathbf{r}_i = -\omega_i\mathbf{t} + \mathbf{s}$
    $\delta_i = ||\mathbf{r}_i||_2$
    if $\delta_{i+1} < eps$ **then**
        return $\mathbf{x}_{i+1}$
    end
    $\rho_i = \rho_{i+1}$
end

Additionally, the simulations used three different meshes: E001 with 11,931 nodes, E002 with 121,316 nodes and E003 with 279,255 nodes. The Table 1.1 shows a comparison between two simulations using the PSPARSLIB and the custom version of the FGMRes. Despite

using the same solver, the number of iterations is different which leads to a larger **t_sol**. The time increase is generated by the higher number of operations performed (2 more iterations to converge), the time spent in creating the OpenCL context and the time transferring data.

Metric	FGMRes PSPARSLIB	FGMRes OpenCL
ite	2	4
nrm_res	1.2371e-04	2.3601e-06
t_sol	1.0813	1.3195
t_preAlg	NA	0.0037
t_alg	NA	0.2422
t_tra	NA	0.0289
t_dev	NA	0.042

Table 1.1: Metrics comparison between simulations using the mesh E001 on the Intel Core i7-3770 (FGMRes PSPARSLIB) and the Intel Xeon Phi 3120A (FGMRes OpenCL). Time unit is second. NA stands for Not Applicable

Tables 1.2 and 1.3 show values of **t_sol** for all the combinations of solvers and meshes running on the different platforms. The numerical results show that the custom implementations were slower than the PSPARSLIB version. Clearly, the size of the meshes were not large enough to overcome the time overhead due to setting the OpenCL context and data transferring. However, two factors are worth to mention along these results. The first factor is that the OpenCL kernels were not optimized (e.g., the SpMV kernel assigned a row by column product to single threads, which leads to workload unbalances). The second factor is the size of the mesh. As the size of the mesh increases the time penalties due to the OpenCL framework are reduced. The second factor can be appreciated in the Figure 1.6. It shows how each OpenCL variant reduces the gap between their own execution time and the execution time of the PSPARSLIB solver.

Solver	Mesh		
	E001	E002	E003
FGMRes PSPRASLIB	363	6,309	32,607
FGMRes OpenCL	388	6,548	33,925
Preconditioned BiCGStab OpenCL	423	7,162	34,740

Table 1.2: Numerical results (**t_sol**) for the simulator running on the Intel Core i7-3770 and the NVIDIA Tesla GPU S2050. Time unit is second.

Solver	Mesh		
	E001	E002	E003
FGMRes PSPRASLIB	317	5,265	25,272
FGMRes OpenCL	368	5,571	26,222
Preconditioned BiCGStab OpenCL	384	6,004	28,205

Table 1.3: Numerical results (**t_sol**) for the simulator running on the Intel Core i7-3770 and the Intel Xeon Phi 3120A. Time unit is second.

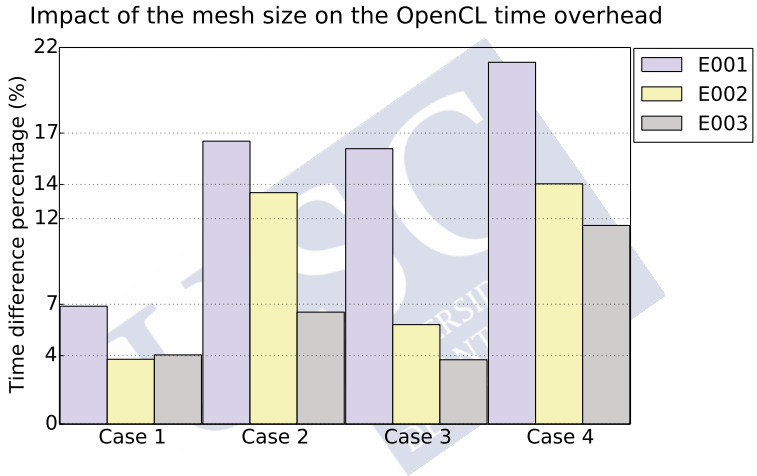

Figure 1.6: The figure shows the time difference percentage of the different cases analyzed. Case 1 compares FGMRes PSPARSLIB vs FGMRes OpenCL on the NVIDIA Tesla GPU. Case 2 compares FGMRes PSPARSLIB vs Preconditioned BiCGStab OpenCL on the NVIDIA Tesla GPU. Case 3 compares FGMRes PSPARSLIB vs. FGMRes OpenCL on the Intel Xeon Phi coprocessor. Case 4 compares FGMRes PSPARSLIB vs. Preconditioned BiCGStab OpenCL on the Intel Xeon Phi coprocessor. The Case 4, for instance, shows that the OpenCL variant of the preconditioned BiCGStab solver reduces the execution time gap with the PSPARSLIB variant of the FGMRes from 21.5% to 12% using a larger size mesh.

To summarize, the main purpose of this study was to familiarize the author of this thesis with the heterogenous programming paradigm, specifically the basic concepts and requirements of the OpenCL language. The study also showed the importance of the problem size to hide the overhead due to the OpenCL implementation. Additionally, the results also showed the need to optimize the OpenCL kernels because the naive implementation of the basic linear algebra operations could not improve the execution time of the original version. Lastly,

the study showed that the Intel Xeon Phi platform executed the simulations faster than the NVIDIA Tesla GPU (Tables 1.2 and 1.3).

1.4.2 Study of basic algebra operations on different accelerators

The first study exhibited two key factors to improve the overall performance of an iterative solver, and consequently of a numerical simulator: the size of the problem, and the optimization of the kernels for the algebra operations. The second step in this thesis roadmap was to focus on the performance of the basic algebra operation kernels on different platforms varying the size of the input arguments [10, 11]. An inspection on the previous Algorithms 1 and 2 shows their strong dependency on the operations: AXPY, DOT, and SpMV. Even the the norm of a vector can be considered as the square root of the dot product of a vector by itself. The following section shows the OpenCL kernel for the operation AXPY.

Function: AXPY

The function AXPY is the most basic of the operations. It is ideal for parallel processing because there is no dependency between the elements of the vectors. The Figure 1.7 shows the independency of the elements graphically.

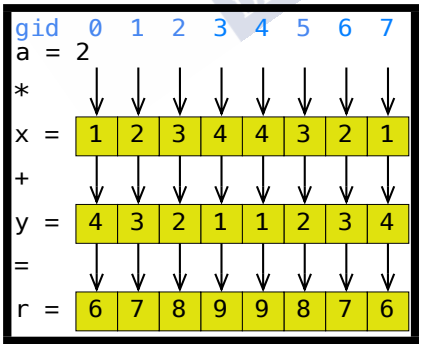

Figure 1.7: AXPY operation. The figure shows no dependency between the elements of the vectors.

The CPU implementation uses a loop to sweep all elements in the arrays to calculate the result (Listing 1.1). The OpenCL implementation is also straight forward. Each work-item is assigned to perform the operation on the same position of the two vectors (Listing 1.2).

```
1 typedef unsigned int UIN;
2 typedef float FPT;
3 void axpy_cpu( FPT a, FPT * x, FPT * y, FPT * r )
4 {
5      UIN i;
6      for ( i = 0; i < LENGTH; i++ )
7           r[i] = a * x[i] + y[i];
8 }
```

Listing 1.1: AXPY routine for CPU. The macro `LENGHT` indicates the size of the vectors.

```
1 typedef unsigned int UIN;
2 typedef float FPT;
3 __kernel void axpy_ocl( FPT a,__global FPT * x,__global FPT * y, __global FPT * r )
4 {
5      UIN gid = get_global_id(0);
6      r[gid] = a * x[gid] + y[gid];
7 }
```

Listing 1.2: AXPY routine in OpenCL.

The `axpy_ocl` kernel was tested by averaging the execution time of ten iterations for different vector sizes. The Figure 1.8 shows that the NVIDIA GPU performs faster for vectors with less than 1M elements. Once the vectors exceed 1M elements the Intel Xeon Phi overcome the GPU achieving a speedup factor of x2 for vector with 50M elements.

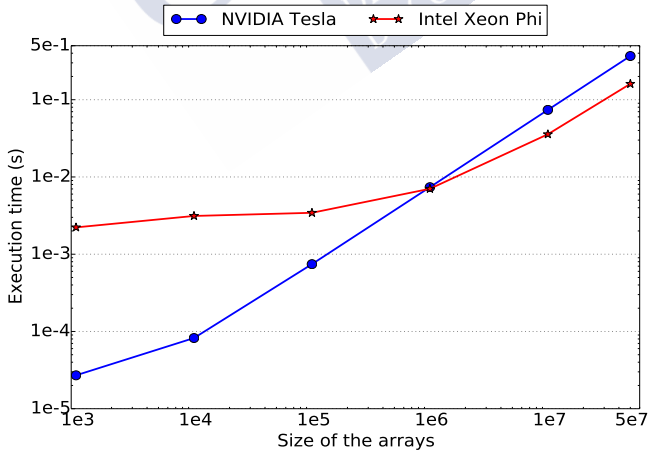

Figure 1.8: The figure shows the execution time behaviour for the AXPY kernels on the NVIDIA Tesla S2050 GPU and the Intel Xeon Phi 3120A.

Function: DOT

The DOT function is a very important operation because its algorithm is required in multiple numerical codes. Additionally, its algorithm is also the building core of the SpMV product. Because the SpMV product is basically a collection of dot products between the rows of a sparse matrix and another vector. At this still early stage, this operation was targeted, by the author of this thesis, for optimization.

The naive implementation of the DOT operation for the CPU is shown in Listing 1.3.

```
1 typedef unsigned int UIN;
2 typedef float FPT;
3 void dot_cpu( FPT a, FPT * x, FPT * y, FPT * r )
4 {
5      UIN i;
6      FPT r = 0.0;
7      for ( i = 0; i < LENGTH; i++ )
8           r = r + x[i] * y[i];
9 }
```

Listing 1.3: Naive DOT implementation for CPU. The macro `LENGHT` indicates the size of the vectors.

The first OpenCL kernel for the DOT operation (Listing 1.4) is basically a translation from CUDA to OpenCL of the work done by Harris in [49]. This kernel basically uses a work-item to load two values from each vector, calculates their product and perform a first two-element reduction and store this result in a block in the local memory of the accelerator device. Once all the positions within the block are filled, the `for` loop, in step 11, performs a reduction based on sequential addressing that uses the work-items local ID as index to avoid memory bank conflicts. Within each iteration of the loop, only the work-items whose local ID is inferior to the lower half of the initial range will be performing a further reduction over the elements of the initial range. This can be graphically appreciated in the Figure 1.9.

The second OpenCL kernel (Listing 1.5) requires two kernels to perform the dot product between two vectors of an arbitrary size. The first kernel assigns each work-item to perform the product between their corresponding elements within the vectors, then each work-item is assigned a chunk (`CHK`) of elements to perform a partial accumulation. The resulting vector will have in the positions multiple of the work-group size the partial reduction of `CHK` elements. The Figure 1.10 shows graphically the first kernel functioning. The complementary OpenCL kernel for the `dot2_ocl` function is shown in Listing 1.6. This kernel uses the work-item whose global ID is 0 to perform a final reduction by sweeping all the positions multiple of `CHK` to calculate the final result.

```
1  typedef unsigned int UIN;
2  typedef float FPT;
3  __kernel void dot1_ocl( __global FPT * v1,__global FPT * v2, __global FPT * vrp )
4  {
5       UIN gid = get_global_id(0);
6       UIN lid = get_local_id(0);
7       UIN i;
8       __local FPT LB[CHK];
9       lb[lid] = v1[gid] * v2[gid] + v1[gid*CHK] * v2[gid+CHK];
10      barrier(CLK_LOCAL_MEM_FENCE);
11      for ( i = (CHK/2); i > 0; i = i >> 1 )
12      {
13           if ( lid < i ) lb[lid] = lb[lid] + lb[lid+i];
14           barrier(CLK_LOCAL_MEM_FENCE);
15      }
16      if ( lid == 0 ) vrp[gid] = lb[lid];
17 }
```

Listing 1.4: Function `dot1_ocl` in OpenCL. The macro `CHK` indicates the size of the work-group.

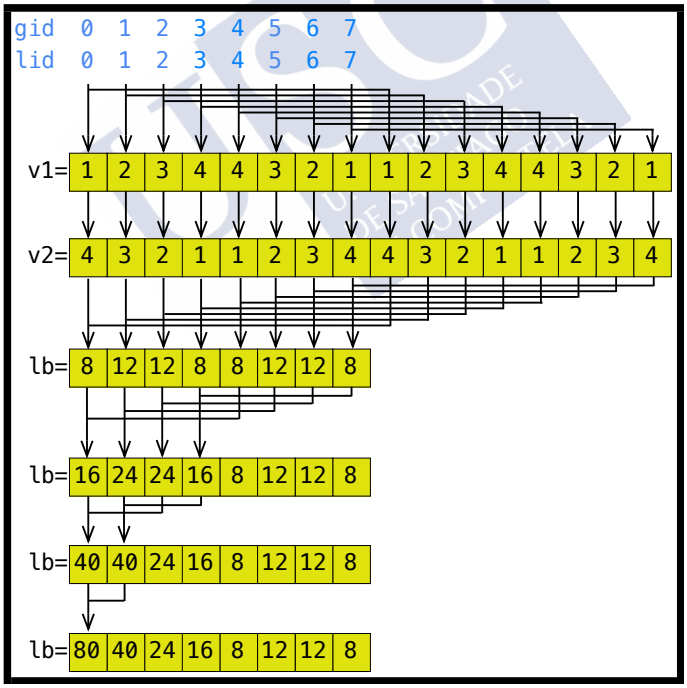

Figure 1.9: Parallel reduction with sequential addressing.

```
1  typedef unsigned int UIN;
2  typedef float FPT;
3  __kernel void dot2_ocl( __global FPT * v1,__global FPT * v2, __global FPT * vrp )
4  {
5       UIN gid = get_global_id(0);
6       UIN lid = get_local_id(0);
7       UIN i;
8       FPT aux = 0.0;
9       vrp[gid] = v1[gid] * v2[gid];
10      if ( lid == 0 )
11      {
12           aux = 0.0;
13           for ( i = 0; i < CHK; i++ )
14                aux = aux + vrp[gid+i];
15           vrp[gid] = aux;
16      }
17 }
```

Listing 1.5: Function `dot2_ocl` in OpenCL.The macro `CHK` indicates the number of elements assigned to each wrok-item.

```
1  typedef unsigned int UIN;
2  typedef float FPT;
3  __kernel void red2_ocl( __global FPT * vrp, __global FPT * vr )
4  {
5       UIN gid = get_global_id(0);
6       UIN ngp = get_num_grps(0);
7       UIN i;
8       FPT aux = 0.0;
9       if ( gid == 0 )
10      {
11           aux = 0.0;
12           for ( i = 0; i < ngp; i++ )
13                aux = aux + vrp[i*CHK];
14           vrp[gid] = aux;
15      }
16 }
```

Listing 1.6: Function `red2_ocl` in OpenCL.The macro `CHK` indicates the number of elements assigned to each work-item.

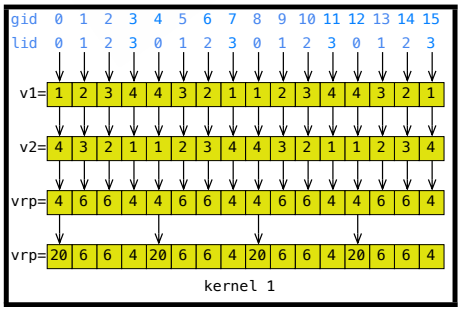

Figure 1.10: DOT operation. The figure shows a two-step reduction. In the first step, each work-item processes a chunk of `CHK` elements, and saves their partial reductions in the positions multiple of `CHK` in the `vrp[]` array.

The two OpenCL kernels for the DOT operation were tested using a different approach.

Two vectors of fixed size of 2,097,152 elements were selected as input arguments for the kernels, instead of varying the size of the input arguments. By doing so, the impact of the work-group size on the performance of the kernels was inspected. The minimum and maximum sizes for the work-groups were selected according the technical specifications for each device. The Figure 1.11 shows that the kernel `dot1_ocl` was faster than the kernel `dot2_ocl` for a work-group size of 256. Clearly, the use of local memory benefits this kernel performance on the NVIDIA GPU [49]. On the other hand, the kernel `dot2_ocl` was faster than the kernel `dot1_ocl` on the Intel Xeon Phi coprocessor (Figure 1.12). The `dot2_ocl` avoids using local memory on the Intel coprocessor since it is not on-chip as it is the case of the NVIDIA GPU. The `dot2_ocl` reaches its faster performance with a work-group size of 128 work-items. Lastly, a comparison between the Figures 1.11 and 1.12 shows that the NVIDIA GPU is faster than its competitor for performing the dot product of two vectors of 2M elements.

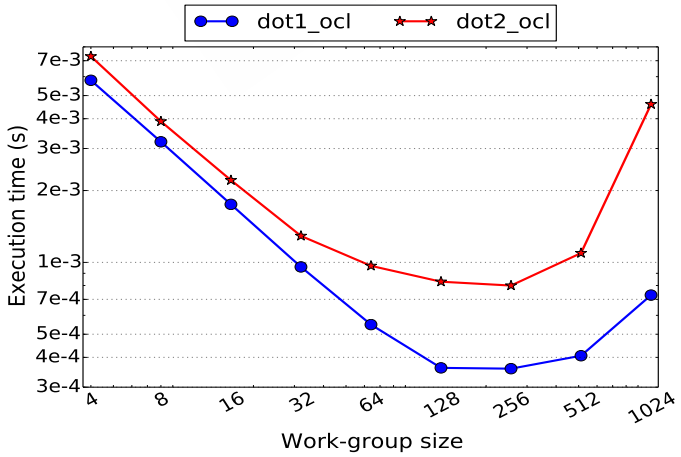

Figure 1.11: Behaviour of the execution time of the dot kernels on the NVIDIA Tesla S2050 GPU.

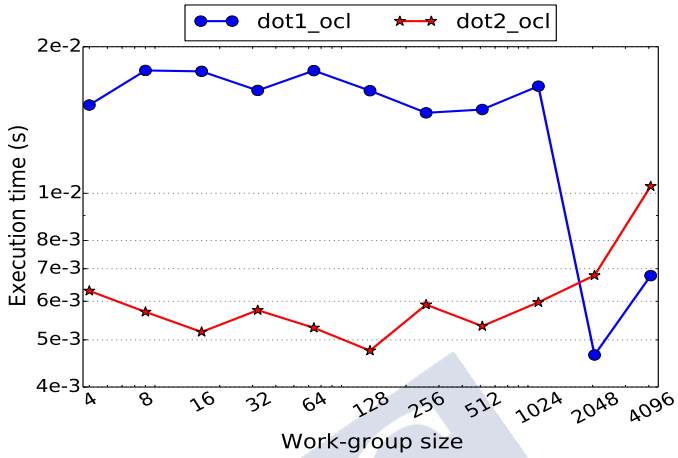

Figure 1.12: Behaviour of the execution time of the dot kernels on the Intel Xeon Phi 3120A coprocessor.

Function: SpMV

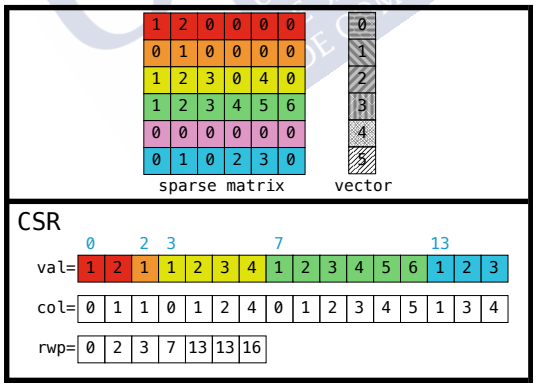

Figure 1.13: Example of a sparse matrix stored using the CSR format.

The SpMV product is the most complex of the basic algebraic operations to optimize. This product not only needs a well-designed computing kernel, but it also needs a storage scheme, for the sparse matrix, that facilitates its parallel implementation. At this stage, only the Compressed Sparse Row (CSR) format was used to study the SpMV product. The CSR format

uses three arrays to store the sparse matrix: the `val[]` array contains the entries of the matrix, the `col[]` array has their corresponding column indices, and the `row[]` array indicates the starting position of each row on the previous two arrays. An example of a sparse matrix contained using the CSR format is shown in the Figure 1.13.

The naive SpMV implementation using the CSR format for the CPU is shown in the Listing 1.7.

```
1  typedef unsigned int UIN;
2  typedef float FPT;
3  void spmv_cpu( FPT * val, UIN * col, UIN * row, FPT * vec, FPT * res )
4  {
5       UIN i,j;
6       for ( i = 0; i < ROWS; i++ )
7       {
8            res[i] = 0.0;
9            for ( j = row[i]; j < row[i+1]; j++ )
10                res[i] = res[i] + val[j] * vec[col[j]];
11      }
12 }
```

Listing 1.7: Naive SpMV implementation for CPU. The macro ROWS indicates the number of rows of the sparse matrix.

Two OpenCL kernels were implemented to perform the DOT product. The first kernel shown in Listing 1.8 is known as the *vector* kernel [12, 13]. This kernel assigns 32 work-items to perform the dot product between each row of the sparse matrix and the right-hand vector. First each work-item, within the work-group, loads an entry from the matrix and a value from the vector and performs their product, then stores its result in an array in local memory (Figure 1.14), and finally, a reduction is performed on the elements of this array using sequential addressing (Figure 1.9). The performance of this kernel depends highly on the number of elements within the rows of the sparse matrix. Clearly, matrices whose rows are highly populated will be better suited for this kernel than matrices with few elements per row.

```
1  typedef unsigned int UIN;
2  typedef float FPT;
3  __kernel void spmv1_ocl( __global FPT * val,__global UIN * col, __global UIN * row, __global FPT * vec,
       __global FPT * res )
4  {
5       UIN lid = get_local_id(0);
6       UIN grpid = get_group_id(0);
7       UIN ind1, ind2, i;
8       __local FPT lb[32];
9       if (grpid < ROWS)
10      {
11           ind1 = row[grpid];
12           ind2 = row[grpid+1];
13           __local lb[lid] = 0.0;
14           for ( i = (ind1+lid); i < ind2; i = i + 32 )
15                lb[lid] = lb[lid] + val[i] * vec[col[i]];
16           barrier(CLK_LOCAL_MEM_FENCE);
17           if ( lid < 16 ) lb[lid] = lb[lid] + lb[lid+16];
18           if ( lid <  8 ) lb[lid] = lb[lid] + lb[lid+ 8];
19           if ( lid <  4 ) lb[lid] = lb[lid] + lb[lid+ 4];
20           if ( lid <  2 ) lb[lid] = lb[lid] + lb[lid+ 2];
21           if ( lid <  1 ) lb[lid] = lb[lid] + lb[lid+ 1];
22           barrier(CLK_LOCAL_MEM_FENCE);
23           if ( lid == 0 ) vrp[grpid] = lb[lid];
24      }
25 }
```

Listing 1.8: Function `spmv1_ocl` in OpenCL.

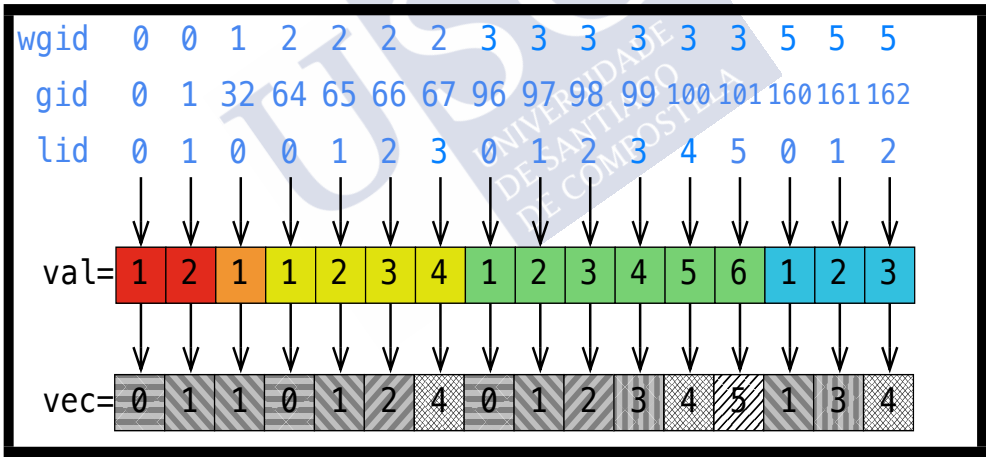

Figure 1.14: Memory accesses by work-items from the execution of kernel `spmv1_ocl`.

The second kernel is shown in Listing 1.9. This kernel is known as *scalar* kernel [12, 13]. It assigns one work-item to perform the dot product between one row of the sparse matrix and the right-hand vector. The performance of this kernel is highly affected by matrices whose rows population is highly variable, because this will generate workload unbalances among threads. The Figure 1.15 shows graphically the memory accesses done by the work-items generated by the execution of the kernel `spmv2_ocl`.

```
1  typedef unsigned int UIN;
2  typedef float FPT;
3  __kernel void spmv2_ocl( __global FPT * val,__global UIN * col, __global UIN * row, __global FPT * vec,
       __global FPT * res )
4  {
5      UIN lid = get_global_id(0);
6      UIN ind1 = row[gid];
7      UIN ind2 = row[gid+1];
8      UIN i;
9      FPT aux = 0.0;
10     for ( i = ind1; i < ind2; i++ )
11       aux = aux + val[i] * vec[col[i]];
12     res[gid] = aux;
13 }
```

Listing 1.9: Function `spmv2_ocl` in OpenCL.

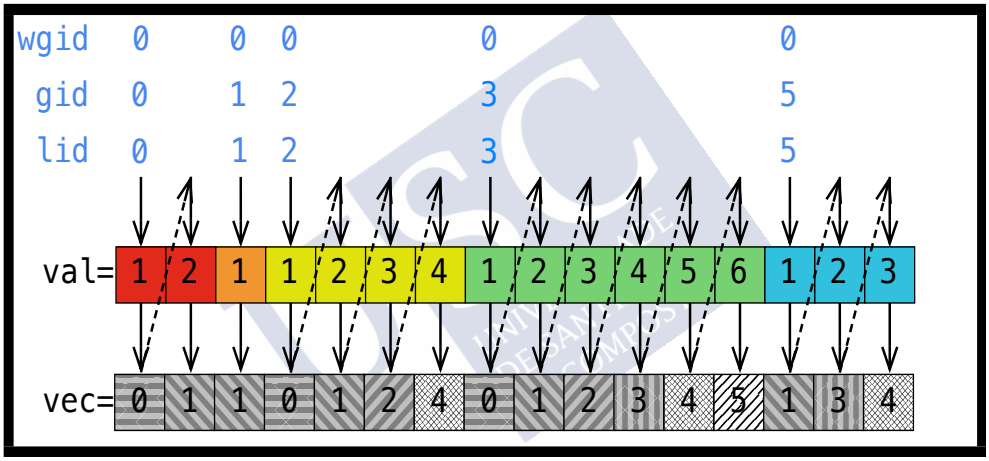

Figure 1.15: Memory accesses by work-items from the execution of kernel `spmv2_ocl`.

The OpenCL kernel `spmv1_ocl` was tested by using a sparse matrix of fixed size and varying the size of the work-group, according to the technical specifications of each device. The sparse matrix used has 16,384 rows with 128 elements per row, for a total of 2,097,152 nonzero elements. The Figure 1.16 shows that this kernel performs better on the NVIDIA GPU because it uses local memory to perform the reduction of the 32 elements of the local array. The optimum work-group size was 64 and 1024 for the NVIDIA GPU and the Intel Xeon Phi coprocessor respectively.

The kernel `spmv2_ocl` was tested by varying the size of the input matrix (Figure 1.17). The second kernel performs faster for matrices with less than 10M elements, achieving an impressive speedup of x30 for the smallest matrix. For matrices with more than 10M elements the Intel Xeon Phi coprocessor closes the performance gap, and it manages to overperform

the NVIDIA GPU achieving a speedup factor of x1.8.

Clearly, the NVIDIA GPU executed faster the operations tested for most vectors and matrices in this study. The Intel Xeon Phi improves its performance when processing high-element vectors and matrices. This study also shows that the SpMV vector is the most time-expensive and difficult operation to optimize. Therefore, it redirected this thesis focus towards the study and analysis of novel sparse matrix formats and their SpMV performance on modern computer architectures.

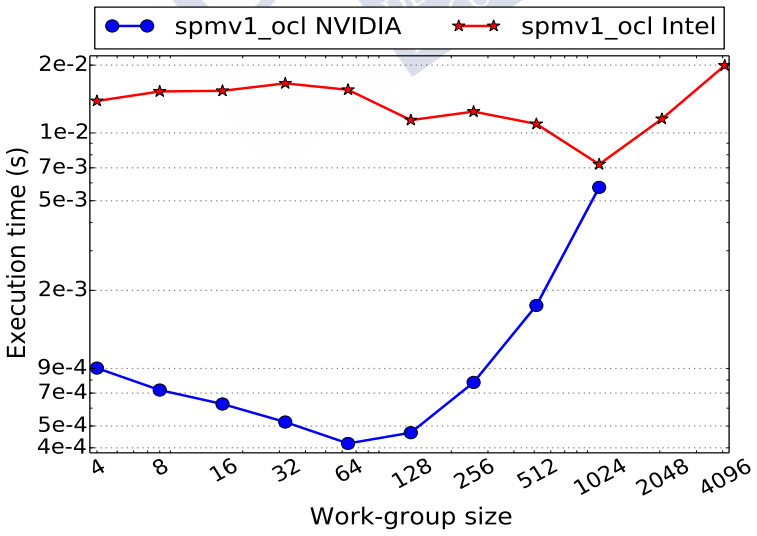

Figure 1.16: Execution of the `spmv1_ocl` kernel on the NVIDIA Tesla S2050 GPU and the Intel Xeon Phi 3120A coprocessor.

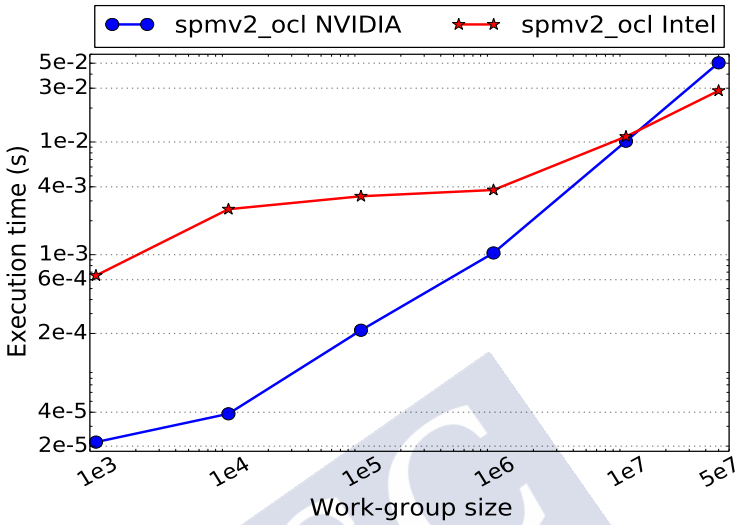

Figure 1.17: Execution of the `spmv2_ocl` kernel on the NVIDIA Tesla S2050 GPU and the Intel Xeon Phi 3120A coprocessor.

1.4.3 New AXC format for the Intel Xeon Phi coprocessor in OpenCL

The previous study exhibited the SpMV product as the main bottleneck in most iterative solvers. It also shows a strong dependency of this product on the storage format and its kernel implementation [12, 13].

The next step in this thesis was to propose the new AXC format for the Intel Xeon Phi 7120P coprocessor [31], based on the analysis of the state-of-the-art solutions proposed to improve the performance of iterative solvers. Specially, the focus was on solutions based on the design of new storage formats and their kernel implementations. A deep look into this field revealed the following formats.

Related work

The CSR format [7] is very popular due to its low memory footprint and its simplicity. It has several variants, one of them is its block version (BCRS) [50]. Other interesting formats are the Diagonal (DIA) format, the Packet (PKT) format and the Hybrid (HYB) format introduced in a study of the SpMV product in CUDA [12, 13].

The ELLPACK format [51] was the first vector-friendly format designed for this kind of machines. This format has served as base for important variants, such as: the ELL-R and ELLR-T formats [14, 20] and [17] respectively, the Sliced ELL (SELL) format [21], the Sliced ELL-C-α (SELL-C-α) [23] and the ELL-WARP (a.k.a K1) format [19].

The Jagged Diagonal Storage (JDS) format [52, 32] is a complex scheme that introduced the use of row sorting to rearrange the rows by population descending order to favor vector processing. One of its variants is the padded JDS (pJDS) format [22].

Competitor formats

Three formats were selected to compare the performance of the AXC format: the CSR, the ELLR-T and the K1 formats.

The CSR format uses three arrays (`val[]`, `col[]` and `rowp[]`) to store the values, the column's indices and the row's starting positions of the matrix respectively.

The ELLR-T format splits each row of the matrix in segments of length $RMAX/T$, where $RMAX$ is the number of elements of the most populated row in the matrix, and T is the number of work-items assigned to process each row. This format uses three arrays (`val[]`, `col[]` and `rl[]`) to store the values, the column's indices and the row's length of the matrix respectively.

The K1 format performs a previous ordering of the matrix in descending order of the rows' population. This ordering is done to group rows with similar number of elements into segments called *blocks*. The size of a block is controlled by the parameter `BLS`, that is, the numbers of rows contained by it. The purpose of this ordering is to balance the workload of the work-groups by creating the most homogeneous blocks as possible. This format uses five arrays (`val[]`, `col[]`, `nmc[]`, `blp[]`, and `permi[]`) to store the values, the column's indices, the number of maximum columns per block, the starting position of each block, and the inverse permutation map of the matrix respectively.

These formats are shown in the Figure **??**.

Metrics

Two metrics were required to test the AXC format. The storage occupancy was used to measure the storage efficiency of the format, and it is also needed to forecast the performance,

it is defined as:

$$\beta = \frac{NNZ}{SE}$$

where NNZ is the number of non-zero elements, and SE is the number of stored elements by the format to reproduce the matrix. And the performance is a measure of the processing speed of the kernels, is defined by:

$$P = \frac{2 \cdot NNZ}{\text{execution time}},$$

where the execution time is measured in seconds.

AXC in OpenCL

The first novelty introduced by this thesis was the AXC format. This format was designed to exploit the wide registers and vectorization capabilities of the Intel Xeon Phi 7120P co-processor. This device was targeted because its superior programmability. It offers two programming approaches: the *native* and the *offload* approaches. The offload approach has the main program executed in the host and only some sections are offloaded to the coprocessor, which make it to behave as an accelerator device. The native approach allows the coprocessor to behave as an Intel Xeon processor running programs natively without necessity of special instructions. This work used OpenCL and the offload approach to code and test the AXC format.

The AXC format was designed to avoid memory indirections caused by accessing no contiguous positions of the right-hand vector, and to exploit the memory cache utilization by grouping data together in segments that fits the cache line. The AXC format stores the matrix entries and vector values contiguously in the `ax[]` array. This array is split in segments called *bricks* of size 2∗`HBRS`. The parameter `HBRS` half the number of elements per brick. The format uses the `brp[]` array to pinpoint the starting position of each row. This format is shown in Figure **??**.

The Listing **??** shows the OpenCL kernel for the AXC format. It uses the explicit vectorization approach. That is, every multiplication and adding operation inside the `for` loop are being vectorised as it explicitly uses the `double8` vector data type. Thus, the vector data types are mapped directly to the hardware vector registers.

The Roofline model [53] was deduced for the kernel in Listing **??**. The kernel needs the transference between host and device of three arrays: `ax[]`, `brp[]` and `y[]`. The effect

of the `brp[]` array is neglected because it does not contribute directly to the floating point operations. Analyzing the contribution of the `ax[]` and `y[]` arrays it arrives to the following formula:

$$P_{AXC} = \frac{b_{max}}{\frac{8}{\beta} + 1.1} \text{ FLOPS.}$$

where b_{max} is the maximum achievable bandwidth.

Numerical results

The AXC format was tested in two stages. The first stage tested exclusively its performance while executing the SpMV product using a suite of 12 matrices (Table **??**). The execution time for each kernel was obtained as the average time of 500 iterations. The bandwidth used for the performance model was measured by averaging the execution time of 1000 iterations of the code shown in Listing **??**, whose value was 123.074 GBytes/s. The performance and the expected performance (Roofline model) for the formats tested (Listings **??** to **??**) are shown in the Figure **??**.

The first set of results showed that the AXC format outperformed its competitors for 7 out of 12 matrices. The K1 format was the best performer for the remaining 5 matrices. The CSR format was the third best performer, and the ELLR-T was the worst performer. Most of these formats performed below the roofline model prediction. The AXC format achieved the nearest performance to the roofline model forecast.

The CSR performance was affected by matrices with few elements per row and poor spatial locality. Its performance for matrix M12 shows this, as it has an average of 3.11 elements per row and has a random sparsity pattern.

The ELLR-T performance is especially susceptible to the number of elements per row of a matrix. If the quantity is low the work-items will process very small workloads because each row is split in T work-items (e.g., matrix M07). On the contrary, if the quantity is high, this could lead to a memory overhead because the format pads all rows with zeros to match the length of the longest row of the matrix (e.g., matrix M12).

The K1 format works specially well for banded matrices (e.g., matrices M01, M02 and M09). But its performance is greatly affected by poor spatial locality (e.g., matrices M06, M11 and M12).

The AXC format was affected by matrices whose rows population is inferior to `HBRS=8`, because it can not effectively exploit the 512-bits registers (e.g., matrices M06 and M07). However, the AXC performance remained unaffected by indirect memory accesses as it outperformed its competitors for matrices with arrow-head sparsity pattern (e.g., matrices M03, M04 and M05).

The second stage tested these formats in a real application, such as the CG solver. The Algorithm **??** shows all the operations, their type, and place of execution needed to implement this solver. The Figure **??** shows a graph for each type of operation (conversion, memory and floatin point) and the total execution time of the algorithm.

The results showed that AXC format has the fastest conversion time due to its simplicity. This is an advantage because it represents the execution of one or two iterations before the solvers based in other formats start their algorithm. The floating point operation times confirms the results from the previous stage, the AXC format was the best performer for 7 out of 12 matrices. The memory operations exposed a major drawback of the AXC format, load a new vector on the `ax[]` array before executing the SpMV product (Step 15 of Algorithm **??**). The time overhead due to memory operations caused that the CG solver based on the AXC format was not the fastest implementation.

In summary, the offload approach leads to a high volume of data transfers that increase the time expended in memory operations, such as array copies from one device to another. The lack of built-in functions in OpenCL to perform vector operations force the use of explicit vectorization on basic operations. The explicit vectorization lacks the optimization of built-in native vector functions as the AVX instructions. The AXC format has the fastest conversion time that leads to an early start of iterative solvers. Its SpMV performance was affected by low populated rows, but it remained unaffected by poor spatial locality.

These observations motivated the use of OpenMP and the Intel AVX-512 instructions to implement and test the AXC format on the Intel Xeon Phi coprocessor.

1.4.4 The AXC format for the Intel Xeon Phi coprocessor with AVX-512

The poor performance showed using the offload approach and OpenCL led this thesis to test again the AXC format using the native approach and OpenMP in conjunction with the AVX-512 instructions on the Intel Xeon Phi 7120P coprocessor [29]. Most of the metrics and methodology employed in the previous study remained the same for this new one [29]. The differences will be pronounced in the following sections.

Competitor formats

The formats used to compare the AXC format were the CSR format and the SELL-C-α format. The optimized Intel MKL function `mkl_cspblas_dcsrgemv()` was used in this work to evaluate the CSR format. The SELL-C-α format is equivalent to the K1 format. However, it was proposed independently in [23]. This variant was selected because its SpMV kernel (Listing **??**) was implemented using the Intel AVX-512 instructions for the Intel Xeon Phi coprocessor. The ELLR-T format was excluded from this study because it could not handle matrices with highly populated rows.

Metrics

This work used the same metrics than the previous study, those are: storage efficiency and SpMV performance. However, because the native approach requires less memory operations, the total execution time describes better the real computing throughput. For this reason, the performance for the two real applications implemented in this analysis were calculated. The first application is the CG solver (Algorithm **??**) whose performance was calculated by:

$$P_{CG} = \frac{2EE + 5N + NIT\,(2EE + 12N)}{ET},$$

where EE is the number of stored elements by the format, N is the order of the matrix, NIT is the number of complete iteration of the main loop and ET is the execution time. The second application is the BiCGStab solver (Algorithm **??**), and its performance was calculated according to:

$$P_{BICGSTAB} = \frac{2EE + 3N + NIT\,(22N + 4EE)}{ET}.$$

AXC in OpenMP and AVX-512

In this work, the AXC format experienced modifications only in its SpMV kernels. Two SpMV kernels were implemented using OpenMP in conjunction with the AVX-512 instructions for the AXC format. The kernel `kaxc1` (Listing **??**) is equivalent to the OpenCL kernel shown in the previous study as Listing **??**. This kernel assigns one row per thread and relies on the OpenMP dynamic policy to balance the workload among threads. The kernel `kaxc2` (Listing **??**) assigns a chunk of rows to each thread, in order to balance manually the workload among threads.

Numerical results

The testing platform was the Intel Xeon Phi 7120P coprocessor using its maximum capacity, that is, 240 threads. The code was compiled and executed using OpenMP and the native approach to avoid data transferring between host and device. A new suite of 25 matrices (Table **??**) was considered for evaluation. Tests were performed in two phases: first phase explored the performance of the SpMV product only, and the second phase focused on the overall performance of the solvers where the SpMV kernels were integrated. A total of 250 runs was performed for the SpMV product, and for the solvers.

The Figure **??** shows the performance of the SpMV product. The AXC format outperformed its competitors in 21 out of 25 matrices. The average performance was: 15.9 GFLOPS for the AXC format, 5.5 GFLOPS for the SELL-C-α format and 8.4 GFLOPS for the CSR-based MKL function. The maximum performance (23.96 GFLOPS) was achieved by the CSR format for matrix M10 which is a banded matrix. The maximum speedup factor (x6.8) was achieved by the AXC format for matrix M07, which has poor spatial locality due to its random sparsity pattern. The maximum AXC format performance was 20.79 GFLOPS for matrix M22, which has an arrow-head sparsity pattern. These results verified the AXC resilience to indirect memory accesses observed in the previous OpenCL results.

The code for all solvers is basically the same. The main differences between one version or another, lies in the extra memory operations required by the AXC and SELL-C-α formats (highlighted in blue and red fonts respectively in Algorithms **??** and **??**), and in the SpMV kernel used for each version. The Figure **??** shows the performance of the CG solver. Despite the AXC format requires the extra step 8, its CG solver outperforms its competitors in 20 out of 25 matrices. The CG solver based on the AXC format achieves a maximum speedup factor of x1.8 over the MKL function for the CSR format for matrix M04.

The Figure **??** shows the performance for the BiCGStab solver. For this solver, the AXC-based version outperforms its competitors in 17 out of 25 matrices. The reduction in the number of positive cases, compared with the CG solver, is explained by the fact that the BICGStab algorithm requires two extra memory operations pointed by steps 12 and 18 in Algorithm **??**. The AXC-based solver achieved a speedup factor of x1.9 over it closes competitor for matrix M07. Both solvers performances showed that the permutation operation required by the SELL-C-α format is more expensive than updating the `ax[]` array. A worthy remark is that the AXC performance was optimal for matrices with arrow-head structure (M00, M03, M09, M16 and M22), and matrices with highly random sparsity pattern (M07).

These results validated the AXC format as a legit option to the CSR format to store a sparse matrix and perform the SpMV product efficiently on the Intel Xeon Phi coprocessor. However, up to this point, the AXC format targeted only the Intel platform disregarding other popular accelerator devices, such as the NVIDIA GPUs. Therefore, the next objective was broadening the range of application of the AXC format to other platforms.

1.4.5 New AXT format using AVX-512 instructions and CUDA

The main objective of the last work of this thesis was to extend the range of application of the AXC format. For this purpose, the new AXT format was proposed [30]. The new AXT format depends on three parameters to adapt itself to different architectures without affecting the scope for general sparse matrices. The AXT format can spawn four different data patterns depending on the values selected for these parameters. This format was tested on two devices: the Intel Xeon Gold 6148 processor with Skylake architecture (Section 1.3.2) and the NVIDIA Tesla V100 GPU with Volta architecture (Section 1.3.2). The Intel AVX-512 instructions were used, in combination with OpenMP, to code the SpMV computational kernels for the Intel processor and CUDA is used for the computational kernels used on the NVIDIA GPU.

Competitor formats

This work used the CSR and the AXC formats to compare the performance of the new AXT format. The current state-of-the-art CSR-based functions were selected from the platforms optimized libraries to obtain the maximum performance for the CSR format. For the Intel Xeon Gold processor, the Intel MKL function `mkl_sparse_d_mv` was used. And for the NVIDIA Tesla V100 GPU, the cuSPARSE function `cusparseCsrmvEx` was used. The AXC format on Intel Xeon Gold processor used the kernel introduced in the previous work, Listing **??**. The AXC format on the NVIDIA GPU used the kernel shown in Listing **??**.

Metrics

This work only focused on the performance of the SpMV product, and the storage efficiency of the format. Hence, only the metrics introduced in Section 1.4.3 were needed: the performance for the SpMV product and the storage occupancy (β).

AXT

The new AXT format uses only two arrays to store the sparse matrix: the `ax[]` array where the matrix's entries and vector values are stored contiguously, and a secondary array. The new format incorporates three tuning parameters in its design to reduce its memory footprint, adapt itself to different architectures and improve the workload balance. The tuning parameters are the tile's half width (`THW`), the tile's height (`TH`) and the mode (`MODE`).

The tile's half width (`THW`) is a machine dependent parameter set to the size of the SIMD execution unit of the targeted device. In this work, `THW=8` for the Intel Xeon Gold processor and `THW=32` for the NVIDIA Tesla V100 GPU, for double precision arithmetic.

The mode (`MODE`) parameter is key to the AXT format because it activates, or deactivates, the usage of zero padding between different row elements in the `ax[]` array. There values for this parameter are uncompacted (`UNC`) and compacted (`COM`). The uncompacted mode indicates that the elements in a row are going to be fitted to as many tiles, or tile's columns, as needed without mixing elements from a different row through the use of zero padding. It also sets the auxiliary array to store row indices (`rwp[]`). The compacted version joins the non-zero elements of a matrix and their corresponding vector values together without any zero padding, which enables a tile, or tile's column, to contain elements from different rows. The compacted version changes the function of the auxiliary array (`hdr[]`), which contains a combination of row indices and offsets in this mode.

The tile's height (`TH`) depends mainly on the matrix's structure. The value of `TH` should be large enough to amortize the cost of the algorithms at a thread-level and yet small enough to avoid a performance reduction through an excess of null operations due to the zero padding in the uncompacted variants or a shared memory access penalization in the compacted variants. This parameter was obtained empirically for each matrix.

The AXT format can generate four different data arrangements according to the values selected for the parameters `TH` and `MODE`. These variants are enumerated in the following list:

- the *AXTUH1* variant is the AXT format with `MODE=UNC` and `TH=1`,
- the *AXTUH* variant is the AXT format with `MODE=UNC` and `TH>1`,
- the *AXTCH1* variant is the AXT format with `MODE=COM` and `TH=1`,
- the *AXTCH* variant is the AXT format with `MODE=COM` and `TH>1`;

and shown in the Figure **??**.

The AXTUH1 variant

The uncompacted mode of the AXT format with `TH=1` stores `THW` matrix entries with the corresponding `THW` vector values contiguously in segments named tiles in the `ax[]` array in row major order. The auxiliary `rwp[]` array assigns the corresponding row index of its elements to each tile. This way race conditions can appear, as different threads processing different elements in the same row could try to write their partial results at the same time and position in the resulting array. This is fixed though the use of atomic functions. The atomic functions reduced the performance of this variant on the Intel processor, but they make the tiles' computation independent which favored its overall performance on the NVIDIA GPUs. The Intel version of the SpMV kernel used an AVX-512 instruction for the reduction step of the partial results (Listing **??**). Its CUDA counterpart (Listing **??**) implemented a warp-level parallel reduction algorithm presented in [54].

The AXTUH variant

This is the 2D variant of the previous scheme, thus here the same arrays are used in the same way as its 1D predecessor. This scheme stores the non zero elements of a matrix together with their corresponding vector values in a column major order separated by `THW` positions. Because this variant's mode is uncompacted, if a row has `NNZR` non zero elements this scheme will store its elements in `(NNZR + TH - 1)/TH` consecutive columns. If there are unused positions in the last column they will be padded with zeros. Unused columns will have a zero on their corresponding position on the `rwp[]` array. Storing the elements in column major order will guarantee coalesced memory access from SIMD lanes and enforces a one-step accumulation of the partial result from each column. Listing **??** and Listing **??** show the kernels for the Intel processor and the NVIDIA GPU respectively.

The AXTCH1 variant

This variant stores the matrix entries and vector values contiguously mixing elements from different rows in the same tile in the `ax[]` array. Having elements from different rows in the same tile, requires an algorithm that can perform more than one reduction within the same tile, that is, a *segmented scan*. A segmented scan is an operation for performing separate parallel scans simultaneously on arbitrary contiguous partitions ("segments") on a given array of numbers. Segmented scans require a segment descriptor that encodes how a sequence is

divided into segments besides the sequence of values. In order to perform a segmented scan, this work used the algorithm presented in [55] known as *fast segmented sum*. The segmented sum requires an input sequence of values (`is[]`) and an array that indicates the distance between the first and last element of each partition (`off[]`). Then, the first step is to store the elements of `is[]` in the `tmp[]`. The second step is to perform a scan on the `is[]` array. And finally, the third step is to apply the following formula:

$$os[i] = is[i + off[i]] - is[i] + tmp[i].$$

The right section of the Figure **??** shows an example of the fast segmented sum algorithm.

Because the fast segmented sum requires to perform a full scan operation on the input sequence. The *block scan* algorithm presented in [56] was used on kernels for this variant. The block scan algorithm requires to use and additional parameter, the block's length. Hence, this variant of the AXT format requires the block's size (`BS`), to indicate the length of the sequence to be scanned. The block scan algorithm follows five steps:

1. scan the input sequence at the SIMD unit level,
2. collect the SIMD unit level results,
3. scan the SIMD unit level results,
4. accumulate the SIMD unit level results,
5. write and return the final results.

Two algorithms were employed in order to perform the first step from the previous list. The first algorithm employed for the CUDA kernel is the warp scan [56] that uses the threads in a warp for 32 elements scans (Figure **??** a)). The second algorithm implemented using the AVX-512 instructions (Listing **??**) is the work-efficient scan [57, 2] that uses the 512-bit registers for 8 double precision elements scans (Figure **??** b)).

Additionally, the auxiliary array for this variant has a different function from those in the previous variants. The auxiliary array for this variant is referred as the `hdr[]` array, and has `THW` elements for every tile contained in the `ax[]` array. The `hdr[]` array needs to provide the distance between the first and last element of each partition (referred to as the offset) and the row's index to which this partition belongs to. Therefore, the first bits of an element belonging to the `hdr[]` array (counting from the least significant bit (LSB)) are dedicated to

specifying the offset of the partition. The last bits of an element from the `hdr[]` array are dedicated to indicate the row's index of the partition. For example, for a block of `BS=1024` elements, the maximum offset within a block is 1023, and it would require 10 bits to cover this value as $1023_{10} = 1111111111_2$, then the remaining 22 bits would indicate the row index, enough to cover matrices whose number of rows is lower than $2^{22} = 4,194,304$. This work covers matrices with higher number of rows than $4,194,304$ by using a maximum block's size of 512 for this variant.

Listings **??** and **??** show the kernels for the Intel processor and the NVIDIA GPU respectively.

The AXTCH variant

The 2D compacted variant of the AXT format stores the matrix's entries and the corresponding vector values separated by `THW` elements in column-major order in the `ax[]` array. This variant allows elements from more than one row to be mixed in the same column to reduce its memory footprint. This variant uses the fast segmented sum algorithm on the tile's columns. Therefore, the `hdr[]` array adopts the same ideas of the previous variant, but its elements are stored in column-major order producing a 2D array. This way, the offset extracted from the `hdr[]` elements indicates the "vertical" distance between the first and last element of a partition. Because the fast segmented sum is applied in column order, the maximum offset is set by the parameter `TH`, hence this variant does not require the parameter `BS`. The compacted mode allows the value for the parameter `TH` to be selected as large as desired without the storing penalty the uncompacted version could generate. However, having a large `TH` could lead to more partial sums that would produce a performance reduction due to more memory writes. The Intel kernel is shown in Listing **??**, and the CUDA kernel is shown in Listing **??**. Notice that the Intel kernels requires to use two static arrays (`blk1[]` and `blk2[]`), while its CUDA counterpart can do the algorithm using only local variables that are stored in the thread registers.

Numerical results

The new AXT format and its competitors were tested on the Intel Xeon Gold 6148 processor and the NVIDIA Tesla V100 PCIe GPU from the GPU nodes of the Cirrus facility [58] at the EPCC, of the University of Edinburgh. The code was compiled using OpenMP and Intel

AVX-512 instructions on the Intel processor and CUDA version 9.1 on the NVIDIA GPU. A suite of 26 matrices were used for testing (Table **??**). The memory requirement, storage occupancy and optimal value selection for all formats and matrices are registered in Tables **??** and **??** for the Intel and NVIDIA platforms respectively. The execution time was the average of 100 executions of each kernel.

The numerical results for the Intel Xeon Gold processor (Figure **??**) show that the *AXTUH* variant was the best performer on this platform using `TH=4` or `8` for most cases. It achieved a performance improvement of 18.1% and it managed to reduce the memory footprint a 5.1% over the AXC format. Compared to the CSR format, the *AXTUH* variant achieved a performance improvement of 44.3% and needed 34.6% more memory. There is no clear preference of this variant for a specific type of matrix because it achieved outstanding speed up factors over different type of matrices like on: the M03 matrix with an arrowhead sparsity pattern (x2.41), the M24 matrix with an irregular sparsity pattern and its abnormally large row (x7.33) and on the M25 matrix with a band sparsity pattern (x2.73).

The performance on the NVIDIA Tesla V100 GPU (Figure **??**) shows that the *AXTUH* and *AXTCH1* variants achieved the best overall performance on this platform using `TH=4` or `TH=8` and `BS=512` or `BS=1024` respectively for most cases. Their performance improvement is lower than 10% while their memory requirements represent an increment of approximately 35% compared to the CSR format. The AXT format excelled processing matrices with abnormally large rows within their ranks (e.g. M07, M20, M21 and M24). These variants reached speedup factors from x2.68 up to x378.50 for this type of matrices. An important remark is that the AXT format overperformed the AXC format on the NVIDIA GPU, widening successfully the range of application of the AXC format.

Finally, the last step in this thesis roadmap was the proposal of the AXT format, a highly adaptable format through the use of three parameters: the tile's half width (`THW`), the mode (`MODE`), and the tile's height (`TH`). This proposal outperformed several highly optimized formats on different modern computer platforms using novel and complex parallel algorithms. Making it a legit option to be used on current numerical codes because it opmitizes the performance of a key operation such as the SpMV product.

1.5 Outline

Chapters 2, 3 and 4 are reproductions of articles that compose the main body of this thesis. The author of this thesis is the main contributor in all of them. These articles have been published in JCR journals: Concurrency and Computation: Practice and Experience, The Journal of Supercomputing and Advances in Engineering Software. These articles deepen the objectives exposed in Section 1.2 and unfold the work undertaken in the development of this thesis. Section 1.6 provides a full compendium of the journal publications and conferences attended related to this thesis.

Chapter 2 proposes the new AXC format to improve the performance of the SpMV product using the OpenCL standard on the Intel Xeon Phi coprocessor. In this chapter, the Roofline Model is used to obtain a performance model for the AXC format. Also, a comparison of the SpMV performance is made using the new proposed format against three other efficient schemes: the Compressed Sparse Row (CSR), the ELLPACKR-T (ELLR-T) and the ELL-WARP (K1) formats. The matrix suite used for testing is composed by 12 matrices with different characteristics. Numerical results show that the AXC format is more robust to spatial indirections proper of sparse matrices. Also, the several CG variants are implemented to expose the strengths and weaknesses of the formats compared in a real application.

Chapter 3 verifies the AXC format as a valid option to improve the iterative solvers performance on the Intel Xeon Phi coprocessor. This chapter presents a two phase comparison of the AXC. The first phase compares the SpMV product performance of the AXC, Sliced ELLPACK-C-α (SELL-C-α) and CSR formats using the OpenMP standard in conjunction with the Intel intrinsic instructions, and the Intel MKL library. The second phase of the evaluation compares the performance of the CG and BiCGStab solvers using the three formats. A suite of 25 matrices is used for testing. Numerical results show that the AXC format achieves the higher average performance of 15.9 GFLOPS. The CSR function of the Intel MKL library achieves 8.4 GFLOPS. And SELL-C-α achieves 5.5 GFLOPS. Results also show that the AXC solver variants achieve a 1.5x and 1.9x performance improvement over the CSR variants of CG and BiCGStab solvers respectively.

Chapter 4 proposes the new AXT format that improves the performance of the AXC format. The new AXT format also widens the application of the AXC format to GPUs. The new proposal can be adapted to different platforms through the adjustment of 4 parameter values that are hardware or matrix dependent. The new format is compared with the AXC and CSR formats using a suite of 26 matrices on an Intel Xeon Gold 6148 processor and an NVIDIA

Tesla V100 GPU. On the Intel Xeon processor the AXT format reaches speedup factors of up to x7 over its competitors. And, on the NVIDIA GPU the AXT format reaches speedup factors of x378.

Lastly, the conclusions of this work, the articles reproduced in the following chapters, and the intended future work are presented in Chapter 5.

1.6 List of publications

The list of publications written by the author throughout the development of this thesis is listed hereunder.

1.6.1 International Journals

- [31] E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, AXC: A new format to perform the SpMV oriented to Intel Xeon Phi architecture in OpenCL, *Concurrency and Computation: Practice and Experience*, 31, 2018.
  Category: Computer Science, Theory & Methods. Rank 59/105.
  Impact Factor (JCR 2018): 1.167. Q3.

- [29] E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Improving Performance of Iterative Solvers with the AXC Format Using the Intel Xeon Phi. *The Journal of Supercomputing*, 74(6): 2823–2840, June 2018.
  Category: Computer Science, Theory & Methods. Rank 35/105.
  Impact Factor (JCR 2018): 2.157. Q2.

- [30] E. Coronado-Barrientos, M. Antonioletti and A.J. García-Loureiro, A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA, *Advances in Engineering Software*, 156:102997, 2021.
  Category: Computer Science, Software Engineering. Rank 13/108.
  Impact Factor (JCR 2019): 3.884. Q1.

1.6.2 National Journals

- [11] E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Study of basic vector operations on Intel Xeon Phi and NVIDIA Tesla using OpenCL, *Annals of Multicore and GPU Programming*, 2(1):66–80, 2015.

1.6.3 National Conferences

- [4] E. Coronado-Barrientos, A.J. García-Loureiro, G. Indalecio and N. Seoane, Implementation of numerical methods for nanoscaled semiconductor device simulation using OpenCL, *In Proceedings of the 2015 Spanish Conference on Electron Devices*, CDE 2015, IEEE, 2015.

- [10] E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Implementation and performance analysis of the AXPY, DOT, and SpMV functions on Intel Xeon Phi and NVIDIA Tesla using OpenCL, *In Second Congress on Multicore and GPU Programming*, PPMG 2015, pages 9–17, University of Extremadura, 2015.

Chapter 2

AXC: A new format to perform the SpMV oriented to Intel Xeon Phi architecture in OpenCL

E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, AXC: A new format to perform the SpMV oriented to Intel Xeon Phi architecture in OpenCL, *Concurrency and Computation: Practice and Experience*, 31, 2018.

DOI: https://doi.org/10.1002/cpe.4864

Copyright

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Chapter 3

Improving Performance of Iterative Solvers with the AXC Format Using the Intel Xeon Phi

E. Coronado-Barrientos, G. Indalecio and A.J. García-Loureiro, Improving Performance of Iterative Solvers with the AXC Format Using the Intel Xeon Phi, *The Journal of Supercomputing*, 74(6): 2823–2840, June 2018.

DOI: https://doi.org/10.1007/s11227-018-2325-6

Copyright

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Chapter 4

A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA

E. Coronado-Barrientos, M. Antonioletti and A.J. Garcia-Loureiro, A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA, *Advances in Engineering Software*, 156:102997, 2021.

DOI: https://doi.org/10.1016/j.advengsoft.2021.102997

Copyright

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

CHAPTER 5

CONCLUSION

The main objective of this thesis, as stated in Section 1.2, was to develop an efficient sparse matrix storage format capable of improving the performance of the SpMV product, an essential operation for a myriad of numerical codes. This was to be accomplished by identifying bottleneck points, successful strategies and useful tools employed in the design of modern efficient formats in state-of-the-art solutions for modern computer architectures.

The following outcomes were observed during the pursue of the secondary objectives that composed the primary goal:

- **objective1**: implement custom iterative solvers using OpenCL to test the performance of a simulator [4].
  outcome1: the OpenCL versions of the FGMRes and the preconditioned BiCGStab solvers were slower than the PSPARSLIB version of the FGMRes solver because naive OpenCL kernels were implemented for the basic linear algebraic operations required by the solvers.
  outcome2: problems solved using OpenCL should be large enough to overcome the time penalties due to OpenCL framework setup, data preprocessing and data transferences. For the particular case of the simulator tested, it needs meshes larger than 279,255 nodes.

- **objective2**: study the performance of the basic linear algebraic operations that are commonly present in iterative solver algorithms on different accelerator devices [10, 11].

outcome3: the NVIDIA Tesla S2050 GPU processed faster than the Intel Xeon Phi 3120A coprocessor the AXPY and DOT operations for arrays whose number of elements are inferior to 50K.
outcome4: the use of local memory favors the performance of the NVIDIA GPU, while its use penalizes the performance of the Intel Xeon Phi coprocessor.
outcome5: the performance of the SpMV product is the hardest to improve and its performance strongly depends on the sparse matrix storage format selected and its kernel implementation.

- **objective3**: Develop a simple sparse matrix format using OpenCL, designed to exploit the Intel Xeon Phi coprocessor capabilities. And test it on a real application [31].
  outcome6: the recommendations on [3] lead to the design of the AXC format. This is a simple format that only uses two arrays to store the sparse matrix. The matrix entries and their corresponding vector values are stored in contiguous positions in the `ax[]` array, which is partitioned in segments of size equal to cache memory lane size.
  outcome7: the data arrangement in the `ax[]` array favors the exploitation of the cache memory, the major performance obstacle of the Intel Xeon Phi coprocessor according to [28].
  outcome8: the inclusion of vector values in the `ax[]` array makes the format robust against indirect memory accesses.
  outcome9: the AXC format outperformed its competitors in 7 out of 12 matrices on the SpMV product. Most of these matrices presented random and arrow-head sparsity patterns.
  outcome10: the simplicity of the AXC format led to a fast conversion time from the CSR format.
  outcome11: the offload programing approach in combination with OpenCL were not optimal for the Intel Xeon Phi coprocessor as they lead to an increased time in memory operations. This fact negated the performance improvement achieved for the SpMV product alone.

- **objective4**: test the SpMV performance of the AXC format on the Intel Xeon Phi coprocessor using OpenMP and Intel AVX-512 vectorized instructions. Also, test the AXC format on iterative solvers. This objective seeks to test the native approach for

workload assignment, and test the OpenMP and Intel AVX-512 instructions combination as an optimization option for the Intel Xeon Phi coprocessor [29].
outcome12: the AXC format outperformed its competitors in 21 out of 25 matrices.
outcome13: the average performances of the SPMV product were 15.9 GFLOPS, 5.5 GFLOPS and 8.4 GFLOPS for the AXC, SELL-C-σ and CSR formats respectively.
outcome14: the maximum speedup factor (x6.8) was achieved by the AXC format for matrix M07, which has poor spatial locality due to its random sparsity pattern.
outcome15: the maximum AXC format performance was 20.79 GFLOPS for matrix M22, which has an arrow-head sparsity pattern.
outcome16: the CG AXC-based solver outperformed its competitors in 20 out of 25 matrices. It also achieved a maximum speedup factor of x1.8 over the MKL function for the CSR format for matrix M04 (with poor spatial locality).
outcome17: The BiCGStab AXC-based solver achieved a speedup factor of x1.9 over its closest competitor for matrix M07 (with poor spatial locality).
outcome18: the native approach led to fewer memory operations than the offload approach. This fact makes the performance of the solvers approximate to the results observed for the SpMV kernels alone.

- **objective5**: develop a new sparse matrix format that requires the minimum of arrays necessary to contain a sparse matrix in order to optimize the SpMV performance on the Intel Xeon and NVIDIA platforms using OpenMP, Intel AVX-512 instructions and CUDA respectively [30].
  outcome19: the new AXT format used two arrays to store the sparse matrix. Additionally, it uses three parameters: the tile's half width (`THW`), the mode (`MODE`), and the tile's height (`TH`), to adapt itself to any accelerator device and reduce its memory footprint if possible.
  outcome20: depending on the values for each parameter the AXT format can spawn four different data arrangements:

  1. the *AXTUH1* variant is the AXT format with `MODE=UNC` and `TH=1`,
  2. the *AXTUH* variant is the AXT format with `MODE=UNC` and `TH>1`,
  3. the *AXTCH1* variant is the AXT format with `MODE=COM` and `TH=1`,
  4. the *AXTCH* variant is the AXT format with `MODE=COM` and `TH>1`.

outcome21: the *AXTUH* variant was the best performer on the Intel Xeon platform using `TH=4` or `TH=8` for most cases. It achieved a performance improvement of 18.1% and it managed to reduce the memory footprint a 5.1% over the AXC format. Compared to the CSR format, the *AXTUH* variant achieved a performance improvement of 44.3% and needed 34.6% more memory.
outcome22: this variant showed no preference for a specific type of matrix because it achieved outstanding speedup factors over different type of matrices like on: the M03 matrix with an arrowhead sparsity pattern (x2.41), the M24 matrix with an irregular sparsity pattern and its abnormally large row (x7.33) and on the M25 matrix with a band sparsity pattern (x2.73).
outcome23: the *AXTUH* and *AXTCH1* variants achieved the best performance on the NVIDIA platform using `TH=4` or `TH=8` and `BS=512` or `BS=1024` respectively for most cases. Their performance improvement is lower than 10% while their memory requirements represent an increment of approximately 35% compared to the CSR format.
outcome24: the AXT format excelled processing matrices with abnormally large rows within their ranks (e.g., M07, M20, M21 and M24). These variants reached speedup factors from x2.68 up to x378.50 for this type of matrices.
outcome25: the increased performance of the AXT format over the AXC format on the NVIDIA platform supports the successful widening of the range of application of the new AXT format.

5.1 Future work

The following list presents intended objectives to accomplish that surge naturally from this thesis:

- The work presented in [30] provided the basis for the AXT_SPL library. This library, registered under the GPL licence, intends to optimize a plethora of numerical applications whose algorithms require sparse linear algebra operations. Naturally, part of the intended future work is to complete this library with the remaining operations not optimised up to this point.

- Test the performance of a real application using the AXT_SPL library.

- Several sparse matrix storage formats were implemented and tested during the development of this thesis, this generates other interesting objective: the utilization of machine learning techniques for the development of a metaheuristic to select the best format option for different types of matrices.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Bibliography

[1] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, 2003.

[2] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[3] Intel Corporation. OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor. URL https://software.intel.com/content/www/us/en/develop/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coproces html.

[4] E. Coronado-Barrientos, G. Indalecio, N. Seoane, and A.J. García-Loureiro. Implementation of numerical methods for nanoscaled semiconductor device simulation using OpenCL. In *Proceedings of the 2015 Spanish Conference on Electron Devices*, CDE 2015. IEEE, 2015. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=7087476&queryText=indalecio&newsearch=true&searchField=Search_All.

[5] Khronos Group. OpenCL. URL https://www.khronos.org/opencl/.

[6] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Philadelphia: Society for Industrial and Applied Mathematics, second edition, 2003.

[7] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, second edition, 1994.

[8] Ask geeks. Intel Xeon Phi 3120A coprocessor. URL https://askgeek.io/es/cpus/Intel/Xeon-Phi-3120A.

[9] Tech Power Up. NVIDIA Tesla S2050 GPU. URL https://www.techpowerup.com/gpu-specs/tesla-s2050.c1538#:~:text=NVIDIA%20has%20paired%2012%20GB,rated%20at%20900%20W%20maximum.

[10] E. Coronado-Barrientos, G. Indalecio, and A.J. García-Loureiro. Implementation and performance analysis of the AXPY, DOT, and SpMV functions on Intel Xeon Phi and NVIDIA Tesla using OpenCL. In *Second Congress on Multicore and GPU Programming*, PPMG 2015, pages 9–17. University of Extremadura, 2015.

[11] E. Coronado-Barrientos, G. Indalecio, and García-Loureiro. Study of basic vector operations on Intel Xeon Phi and NVIDIA Tesla using OpenCL. *Annals of Multicore and GPU Programming*, 2(1):66–80, 2015. URL https://revistaseug.ugr.es/index.php/amgp/article/view/3056.

[12] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA, 2008.

[13] N. Bell and M. Garland. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM. URL http://doi.acm.org/10.1145/1654059.1654078.

[14] F. Vázquez, E. M. Garzón, A. Martínez, and J. J. Fernández. The sparse matrix vector product on GPUs. *Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering*, 2:1081–1092, 2009. URL http://hpca.ual.es/~fvazquez/fp-content/attachs/NVIDIA_TECHREPORT09.pdf.

[15] Intel Corporation. Developer Reference for Intel Math Kernel Library - C BLAS and Sparse BLAS Routines. URL https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines.html.

[16] NVIDIA Corporation. Developer Zone - The API reference guide for cuSPARSE. URL https://docs.nvidia.com/cuda/cusparse/index.html.

[17] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón. Improving the performance of the sparse matrix vector with GPUs. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1146–1151, Washington, DC, USA, 2010. IEEE Computer Society. URL http://dx.doi.org/10.1109/CIT.2010.208.

[18] R. G. Grimes, D. R. Kincaid, and D. M. Young. *Itpack 2.0 User's Guide*. Center for Numerical Analysis, The University of Texas at Austin, 1979. URL https://books.google.es/books?id=EseZwgEACAAJ.

[19] J. Wong, E. Kuhl, and E. Darve. A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. *International Journal for Numerical Methods in Engineering*, 102(12):1784–1814, 2015. URL http://dx.doi.org/10.1002/nme.4865.

[20] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, June 2011. URL http://dx.doi.org/10.1002/cpe.1658.

[21] A. Monakov, A. Lokhmotov, and A. Avetisyan. *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings*, chapter Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, pages 111–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-11515-8_10.

[22] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop. Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1696–1702, Washington, DC, USA, 2012. IEEE Computer Society. URL http://dx.doi.org/10.1109/IPDPSW.2012.211.

[23] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 36:C401–C423, 2014. URL https://epubs.siam.org/doi/abs/10.1137/130930352?journalCode=sjoce3.

[24] NVIDIA Developer Zone. CUDA Toolkit documentation. URL https://docs.nvidia.com/cuda/index.html.

[25] Intel Corporation. Intel Intrinsics Guide. URL https://software.intel.com/sites/landingpage/IntrinsicsGuide/#.

[26] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming. Knights Landing Edition*. Morgan Kaufmann, 2016.

[27] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, first edition, 2011.

[28] E. Saule, K. Kaya, and Ü. V. Çatalyürek. *Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi*, pages 559–570. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. URL http://dx.doi.org/10.1007/978-3-642-55224-3_52.

[29] E. Coronado-Barrientos, G. Indalecio, and A. J. García-Loureiro. Improving Performance of Iterative Solvers with the AXC Format Using the Intel Xeon Phi. *J. Supercomput.*, 74(6):2823–2840, june 2018. URL https://doi.org/10.1007/s11227-018-2325-6.

[30] E. Coronado-Barrientos, M. Antonioletti, and A.J. Garcia-Loureiro. A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA. *Advances in Engineering Software*, 156:102997, 2021. URL https://doi.org/10.1016/j.advengsoft.2021.102997.

[31] E. Coronado-Barrientos, G. Indalecio Fernández, and A. J. García-Loureiro. AXC: A new format to perform the SpMV oriented to Intel Xeon Phi architecture in OpenCL. *Concurrency and Computation: Practice and Experience*, 31, 2018. URL https://doi.org/10.1002/cpe.4864.

[32] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientifics and Engineers*. CRC Press, 2010. URL https://dl.acm.org/doi/10.5555/1855048.

[33] J. N. Reddy. *An Introduction to Nonlinear Finite Element Analysis*. Oxford University Press, second edition, 2015. URL https://oxford.universitypressscholarship.com/view/10.1093/acprof:oso/9780198525295.001.0001/acprof-9780198525295.

[34] T. A. Beu. *Introduction to Numerical Programming A Practical Guide for Scientists and Engineers Using Python and C/C++*. CRC Press, 2015. URL https://www.routledge.com/Introduction-to-Numerical-Programming-A-Practical-Guide-for-Scienti Beu/p/book/9781466569676.

[35] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. TOP500. URL https://www.top500.org/lists/top500/.

[36] OpenMP organization. OpenMP. URL https://www.openmp.org/.

[37] Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, second edition, 2011.

[38] Ask geeks. Intel Xeon Phi 7120P coprocessor. URL https://askgeek.io/en/cpus/Intel/Xeon-Phi-7120P.

[39] Intel Corporation. Intel Core i7-3770. URL https://ark.intel.com/content/www/us/en/ark/products/65719/intel-core-i7-3770-processor-8m-cache-up-to-3-90-ghz.html.

[40] Ask geeks. Intel Xeon Gold 6148 processor. URL https://askgeek.io/en/cpus/Intel/Xeon-Gold-6148.

[41] NVIDIA. NVIDIA Tesla V100 PCIe GPU. URL https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184.

[42] R. Chandra and L. Dagum and D. Kohr and D. Maydan and J. McDonald and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, second edition, 2001.

[43] M. Scarpino. *OpenCL in Action: How to accelerate graphics and computations*. Manning Publications Co., first edition, November 2011.

[44] J. Cheng, M. Grossmann, and T. McKercher. *Profesional CUDA C Programming*. John Wiley & Sons, Inc., 2014.

[45] D. Storti and M. Yurtoglu. *CUDA for Engineers An introduction for High-Performance Parallel Computing*. Addison-Wesley, 2016.

[46] Texas A&M University. SuiteSparse Matrix Collection. URL https://sparse.tamu.edu/.

[47] Python organization. The Python languaje. URL https://www.python.org/.

[48] Matplotlib organization. matplotlib. URL https://matplotlib.org/.

[49] M. Harris. Optimizing Parallel Reduction in CUDA. URL https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[50] R. W. Vuduc and H.-J. Moon. Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure. In *Proceedings of the First International Conference on High Performance Computing and Communications*, HPCC'05, pages 807–816, Berlin, Heidelberg, 2005. Springer-Verlag. URL http://dx.doi.org/10.1007/11557654_91.

[51] D. R. Kincaid, T. C. Oppe, and D. M. Young. ITPACKV 2D User's Guide, May 1989. URL https://web.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[52] S. Hossain. On efficient data structures for sparse matrix storage, july 2006. URL http://www.mathematik.hu-berlin.de/~gaggle/EVENTS/2006/BRENT60/presentations/Shahadat%20Hossain%20-%20On%20efficient%20data%20structures%20for%20sparse%20matrix%20storage.pdf.

[53] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, 2009. URL http://doi.acm.org/10.1145/1498765.1498785.

[54] Y. Lin and V. Grover. Using CUDA Warp-Level Primitives. URL https://devblogs.nvidia.com/using-cuda-warp-level-primitives/.

[55] W. Liu and B. Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication, 2015.

[56] S. Sengupta and M. Harris and M. Garland. Efficient Parallel Scan Algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA Corporation, December 2008.

[57] S. Sengupta, M. Harris, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 39–69. Addison Wesley Professional, 2007.

[58] Edinburgh Parallel Computing Centre (EPCC). Cirrus. URL http://www.cirrus.ac.uk/.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

List of Algorithms

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

List of Figures

List of Listings

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

List of Tables