Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática

Grado de Ingeniería Informática

▪ Proyecto Fin de Grado ▪

Ingeniería de Computadores

Estudio de arquitecturas Intel Xeon vs Intel Xeon Phi y comparativa de rendimiento.

Alumno

Javier Aldazabal Rego

Junio 2016

Director

Clemente Rodríguez Lafuente

informatika fakultatea
facultad de informática

Agradecimientos

Este proyecto se lo quiero agradecer a toda la gente que me ha apoyado a lo largo de mi vida. A mi padre Valentín, a mi hermano Oscar, y especialmente a mi madre Ana, cuya marcha no hizo más que aumentar su presencia en mí y darme fuerzas para seguir adelante.

En segundo lugar se lo quiero agradecer a mis amigos, que me han aguantado y animado durante todo este tiempo, sacándome una sonrisa cuando lo necesitaba.

Finalmente, me gustaría agradecer enormemente toda la ayuda que he recibido por parte de mi tutor, Clemente, por brindarme la oportunidad de hacer este proyecto y por el apoyo y paciencia proporcionados pese a las dificultades presentadas.

De igual manera gracias a Santiago Díez por su inmejorable disposición a solventar los problemas técnicos y al grupo de Alex Mendiburu por prestarnos sus recursos sin dudarlo y poder así completar el proyecto.

A todos ellos, ***gracias de todo corazón****.*

"La privacidad es uno de los mayores problemas en esta nueva era electrónica"

Andy Groove (1936 -2016)

Resumen

El objetivo principal del proyecto es analizar el rendimiento de un acelerador específico como es el coprocesador Xeon Phi de Intel. Para este propósito se ha estudiado el impacto que su utilización tiene en un conjunto de programas, junto a una plataforma que incorpora un Intel Xeon E5.

Hemos analizado las optimizaciones que ICC, *Intel C++ Compiler*, ofrece para aumentar la eficiencia de los programas, aprovechando factores como la arquitectura en la que se ejecutan o distintas técnicas sobre los algoritmos, pero dando especial relevancia a la vectorización y la paralelización.

El rendimiento del coprocesador Xeon Phi lo hemos evaluado en relación a las prestaciones del Xeon E5 utilizando para ello el mencionado compilador de Intel. Asimismo, hemos trabajado con diversas herramientas presentes en el desarrollo para este tipos de sistemas, como son las librerías MKL, *Math Kernel Library*, o extensiones de lenguaje como LEO, *Language Extensions for Offload,* para la computación heterogénea. También se han utilizado la notación array de Intel CILK para la vectorización o una herramienta como OpenMP, *Open Multi-Processing,* para trabajar el paralelismo.

Como conclusiones, hemos podido comprobar que tras estudiar el sistema y utilizar herramientas para obtener rendimiento del mismo, el tipo de aplicación ejecutada determina el aprovechamiento de un recurso como el coprocesador Xeon Phi.

Para el tipo de aplicaciones limitadas por memoria, la velocidad del bus hace que la transferencia de datos domine la ejecución. Por otra parte, hemos visto que las limitadas por cómputo son las adecuadas para el uso de los Xeon Phi. Para las aplicaciones mixtas como la desarrollada hemos comprobado que aumentamos el rendimiento necesitando algo más de consumo, por lo que en este tipo de escenarios se ha de evaluar la rentabilidad en cada caso.

Tabla de contenidos

Lista de Ilustraciones y Tablas

ILUSTRACIONES

TABLAS

1

Introducción

El modelo de fabricación de microprocesadores de Intel ha seguido siempre una metodología concreta llamada *tick-tock*. Ésta consiste en una **reducción en la fabricación** de sus chips que sería el ***tick*** (integración a nivel de nanómetro) seguido de la etapa de diseño de una **nueva microarquitectura** de sus procesadores conocida como ***tock***.

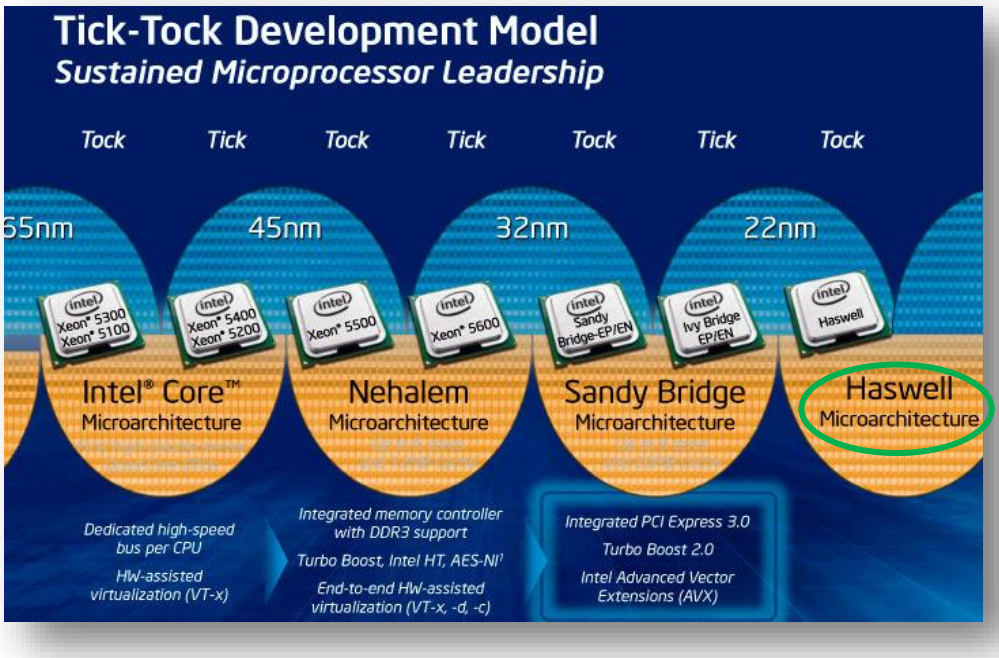


Ilustración 1.1 Modelo tick tock de Intel

En ilustración 1.1 podemos ver la evolución de este proceso (aunque en el futuro puede no continuar este modelo debido a la dificultad de disminuir la integración por debajo de los 14nm).

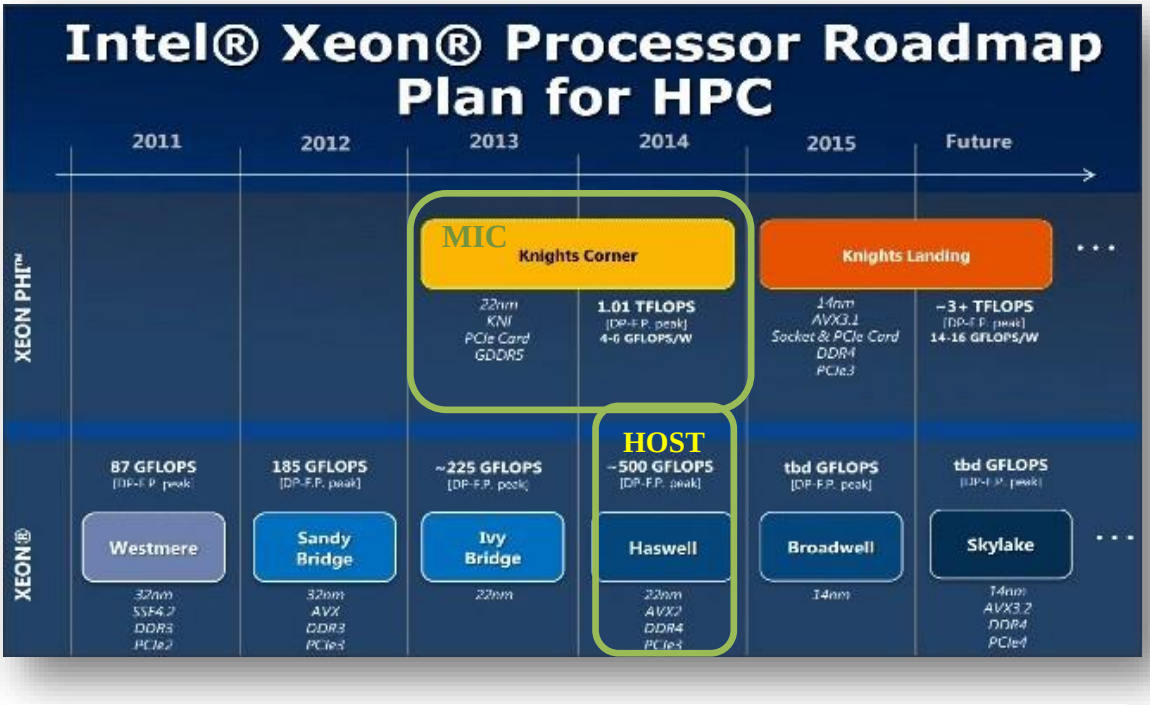


Ilustración 1.2 Evolución Xeon - Xeon Phi

En las últimas decadas, la demanda de velocidad de cómputo se resolvió con el aumento sucesivo de la frecuencia de trabajo de los procesadores. Las altas frecuencias de reloj empezaban a generar importantes consumos de energía y más disipación de calor, por lo que en los últimos años se optó por incorporar un nuevo diseño: la introducción de varias unidades de proceso en el mismo chip era una realidad, los llamados procesadores multicore.

El modelo de diseño multicore (dos o más núcleos de procesador en el mismo circuito integrado) ha sido llevado al extremo para fabricar procesadores de más de 50 núcleos a menor frecuencia y con un arquitectura más sencilla, llamados procesadores *manycore*. Un ejemplo es el coprocesador Intel Xeon Phi, que vamos a utilizar en este trabajo, concretamente el modelo 3120A.

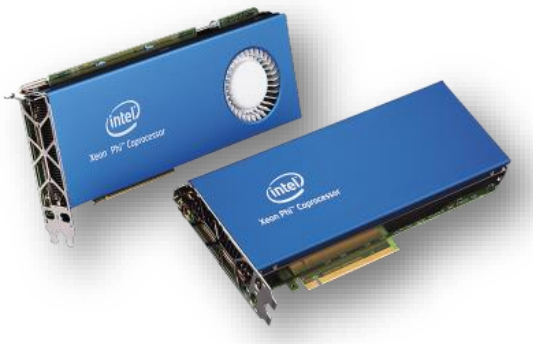


Ilustración 1.3 Intel Xeon Phi

Estos coprocesadores además de contar con múltiples núcleos para aprovechar el **paralelismo**, cuentan también con una unidad de procesamiento **vectorial** en cada uno de ellos, con el objetivo de usar más datos por operación. Con un uso adecuado de estos recursos disponibles se puede conseguir aumentar el rendimiento. Es esencial evaluar la eficiencia del sistema ponderando el rendimiento y el consumo eléctrico.

Es clave que antes de cambiar nuestros proyectos a otra plataforma para adecuarla a esa arquitectura, los optimicemos antes para el sistema anfitrión (paralelizable / vectorizable) y comprobar así la mejora, si existe, al utilizar el nuevo sistema Xeon Phi.

Para trabajar con sistemas diferentes en una misma aplicación se utiliza la **computación heterogénea**, cuyo código se ejecuta en diferentes procesadores para aprovechar todas las capacidades de su arquitectura.

Mostramos con una serie de gráficos la tendencia del mercado en cuanto al **uso de coprocesadores** por parte de supercomputadores de www.**top500**.org, proyecto que elabora una lista con los 500 de mayor rendimiento del mundo.

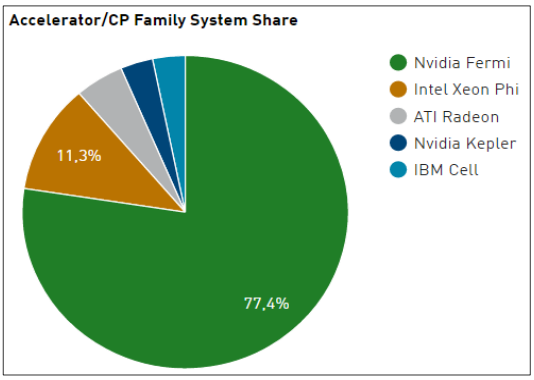


Ilustración 1.4 Aceleradores Noviembre 2012

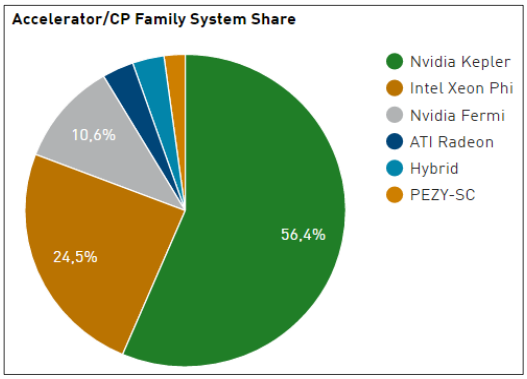


Ilustración 1.5 Aceleradores Junio 2016

Empezamos con las ilustraciones 1.4 y 1.5 para reflejar la cuota de este mercado desde el inicio hasta la actualidad para el uso de Intel Xeon Phi frente a otras familias, y posteriormente en la tabla 1.1 la fluctuación a lo largo de todo el periodo.

Un buen indicador de la potencia de cálculo de este coprocesador es su utilización por parte del supercomputador que figura como el más potente en la lista top500, el **Tianhe-2**, pues emplea unidades **Xeon Phi 31S1P.**

Lista	NOV'12	JUN'13	NOV'13	JUN'14	NOV'14	JUN'15	NOV'15	JUN'16
% Cuota	11,3	20,4	22,6	25	26,7	33,7	28,2	24,5

Tabla 1.1 Presencia de coprocesadores Xeon Phi desde su creación

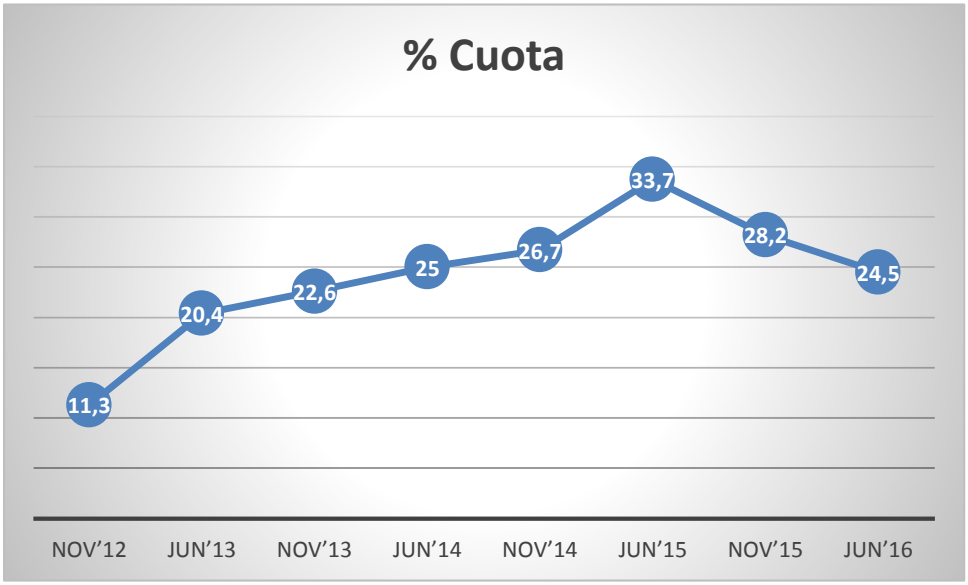


Ilustración 1.6 Evolución presencia Xeon Phi en top500

Destacamos también el gran dominio del Intel **Xeon E5** en el sector de los procesadores, que hemos utilizado como host. La ilustración 1.7 es un reflejo en la actualidad de este uso mayoritario en la citada lista con el **85,2%** de cuota entre las distintas arquitecturas (Haswell, IvyBridge, SandyBridge, Broadwell).

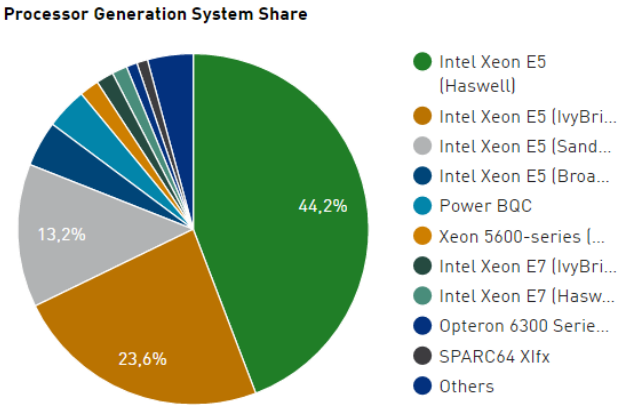


Ilustración 1.7 Familia Xeon E5 Junio 2016

1.1. Objetivos del proyecto

El objetivo principal será **verificar la utilidad del coprocesador** Intel Xeon Phi con procesadores actuales en diferentes contextos.

Las líneas generales del presente proyecto son:

a) Una descripción de los sistemas utilizados, profundizando en su arquitectura y manejo.

   1. Expondremos una **visión general** del sistema utilizado, proporcionando una guía para conectarse al mismo y para manejarlo en entornos Windows y Linux.
   2. Mostraremos también **cómo compilar** para este tipo de plataformas además de una guía para la conexión al coprocesador y cómo monitorizarlo.
   3. Realizaremos un estudio de las plataformas **Xeon y Xeon Phi**, describiendo características como la memoria, el set de instrucciones, el procesador o el consumo.

b) El análisis del uso intensivo del compilador de Intel, mostrando opciones útiles para el rendimiento, e introduciendo herramientas habituales en el desarrollo para un sistema como el utilizado :

   1. Analizaremos el potencial de ICC, tanto en **uso de optimizaciones** como en la generación de código **vectorial y paralelo**. Haremos especial referencia a la producción de **reportes** que nos ofrecen para verificar estas optimizaciones.
   2. Resumiremos cómo ejecutar aplicaciones de manera **nativa** en el Xeon Phi.
   3. Trabajaremos con la **programación heterogénea**, usando herramientas que permiten la ejecución de código en el host y en el coprocesador, tanto secuencialmente como simultáneamente.
   4. Utilizaremos la notación para **arrays CILK** de Intel con el fin de ayudar al compilador a vectorizar funciones.
   5. Explicaremos el uso de **OpenMP** para trabajar el paralelismo y ver cual es la configuración adecuada.
   6. Utilizaremos algunas de las librerías más optimizadas como las librerías matemáticas **Intel MKL**.

c) La categorización por tipo de aplicaciones según su limitación, destacando un programa representativo de cada categoría.

    1. Describiremos los programas **acotados por memoria** (*memory bound*), seleccionando la función Saxpy de la librería BLAS1, *Basic Linear Algebra Subprograms,* como referencia.
    2. Explicaremos los programas **acotados por cómputo** (*cpu bound*). La referencia aquí será la multiplicación de matrices genérica de categoría BLAS3.
    3. Propondremos programas que son una **mezcla de los anteriores**. Este caso consistirá en un programa que utiliza un servidor de peticiones y que trabaja con una matriz de datos.

En resumen, explicando el sistema en profundidad, aprovechando las posibilidades del compilador ICC y teniendo en cuenta cuál sería el cuello de botella por tipo de aplicación, trataremos de concluir la utilidad del uso de un sistema heterogéneo con un coprocesador Xeon Phi.

1.2. Estructura de la memoria

En este apartado se explica el contenido de este documento, detallando cada uno de los capítulos que encontramos:

1. **Introducción y objetivos:** En este capítulo se muestra la evolución del sistema utilizado, así como los objetivos del proyecto y la descripción de la estructura de la memoria.

2. **Sistema:** Este capítulo muestra una descripción detallada de las arquitecturas de las plataformas utilizadas, así como su conexión y manejo.

3. **Compilación:** En este capítulo se describe el uso del compilador ICC junto a diversas optimizaciones, y también aparece el modo de ejecutar de forma nativa en el coprocesador. Se explica la computación heterogénea y algunas herramientas para este tipo de modelos.

4. **Categorías de aplicaciones:** Este apartado muestra inicialmente una configuración ideal del coprocesador, seguida de los tipos de aplicaciones según la limitación de recursos, que van acompañados de ejemplos referentes. Se muestran, para éstos, resultados de ejecución, consumo energético y conclusiones extraídas.

5. **Planificación:** Este capítulo contiene el diagrama de realización del proyecto, junto a la lista de reuniones concertadas con el director del proyecto.

6. **Conclusiones y líneas futuros:** En este apartado se reflejan las conclusiones generales que se desprenden de la realización del proyecto, y también posibles áreas de investigación adicional relacionadas con el proyecto.

7. **Bibliografía:** En este capítulo se ilustra todo el material consultado a la hora de realizar la totalidad del proyecto.

11. **Anexo A**: Se lista el código fuente de los últimos programas realizados en el capítulo 4.

2

El sistema

2.1. Visión general

La plataforma Xeon E5 que hemos utilizado en este trabajo, tiene instalado el coprocesador Intel Xeon PHI en una ranura PCIexpress v2, *peripheral Component Interconnect Express*. La ilustración 2.1 muestra una **visión general de la arquitectura** del sistema:

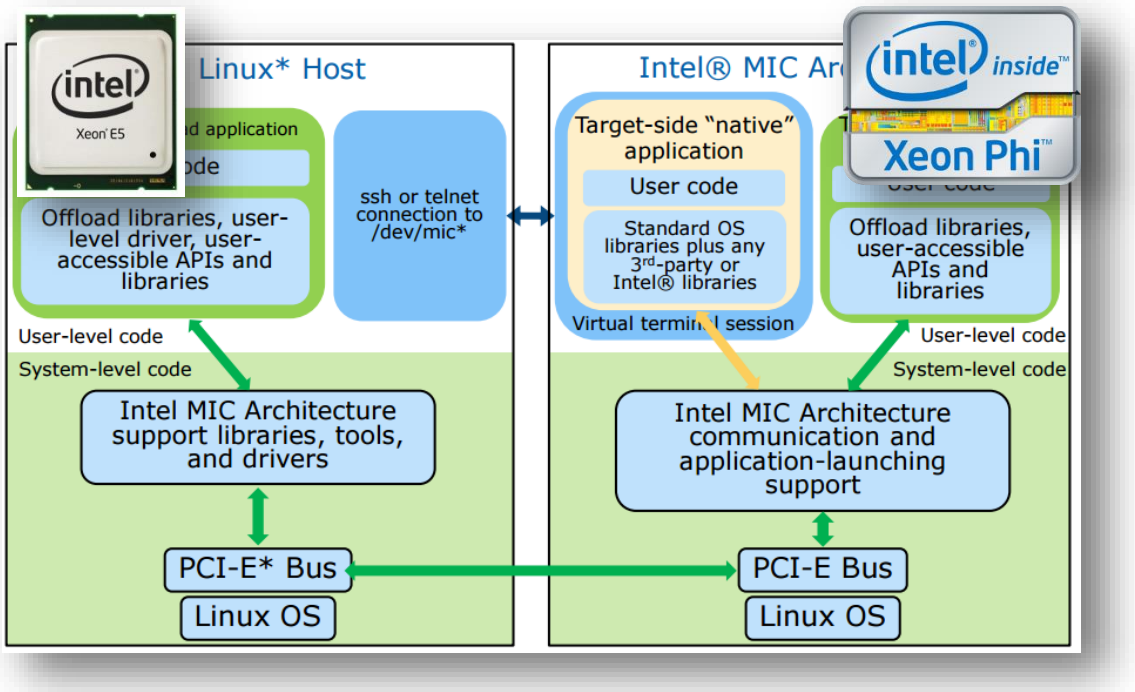


Ilustración 2.1 Visión general del sistema

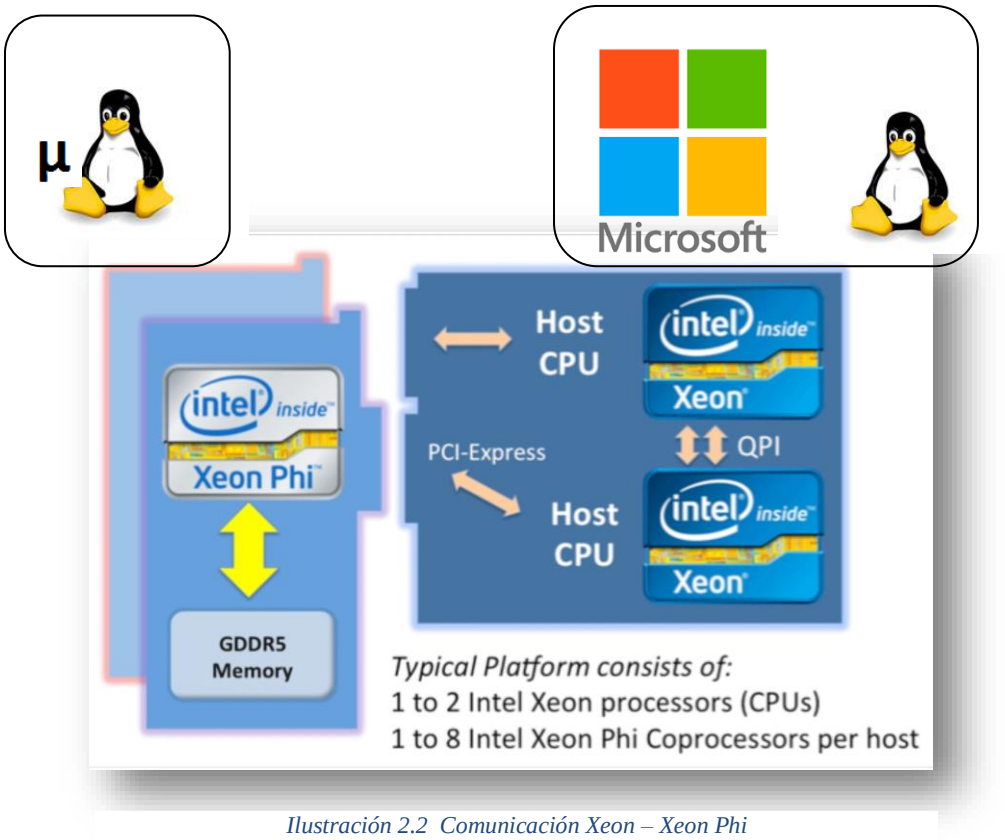


Ilustración 2.2 Comunicación Xeon – Xeon Phi

El sistema tiene como **host** el procesador Xeon E5, cuyo manejo hemos podido comprobar en **Windows** (Server 2012) y en **Linux** (CentOS7). Por otro lado tenemos el coprocesador Xeon Phi que ejecuta de sistema operativo una **microversión de Linux**, como vemos en la ilustración 2.2.

En cuanto a la **comunicación**, puesto que el coprocesador está conectado a una ranura **PCI-Express v2**, el ancho de banda entre los sistemas es el siguiente:

500 MB/s por carril → 16 carriles = **8GB/s**

La conversión de las unidades GT/s a GB/s se obtiene realizando las operaciones:

5GT/s * (8/10) codificación * 16 carriles / 8 (bit_a_Byte) = 8GB/s

2.1.1.- Guía de conexión

Para conectarse al host desde el exterior y poder manejarlo, deberemos seguir los siguientes pasos:

1. Conectarse a la red de la facultad de informática mediante una aplicación de gestión de redes privadas virtuales o VPN, *virtual private network*.

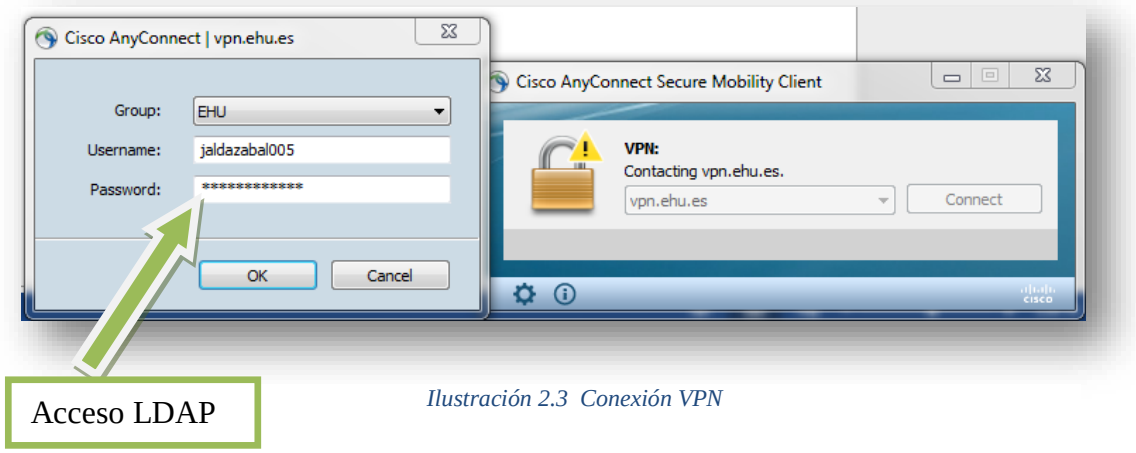


Ilustración 2.3 Conexión VPN

A. Si el host está ejecutando Windows, podremos entonces realizar una conexión a escritorio remoto del host, tal como indica la ilustración 2.4.

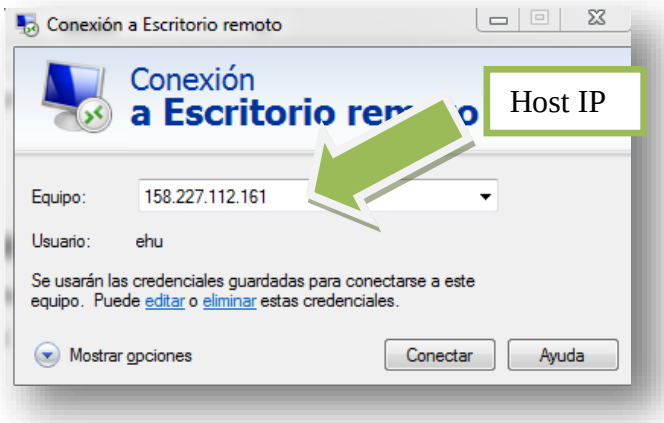


Ilustración 2.4 Conexión a host Windows

B. Si el host está ejecutando Linux, utilizaremos Putty para realizar una conexión SSH, *Secure Shell,* como en la ilustración 2.5.

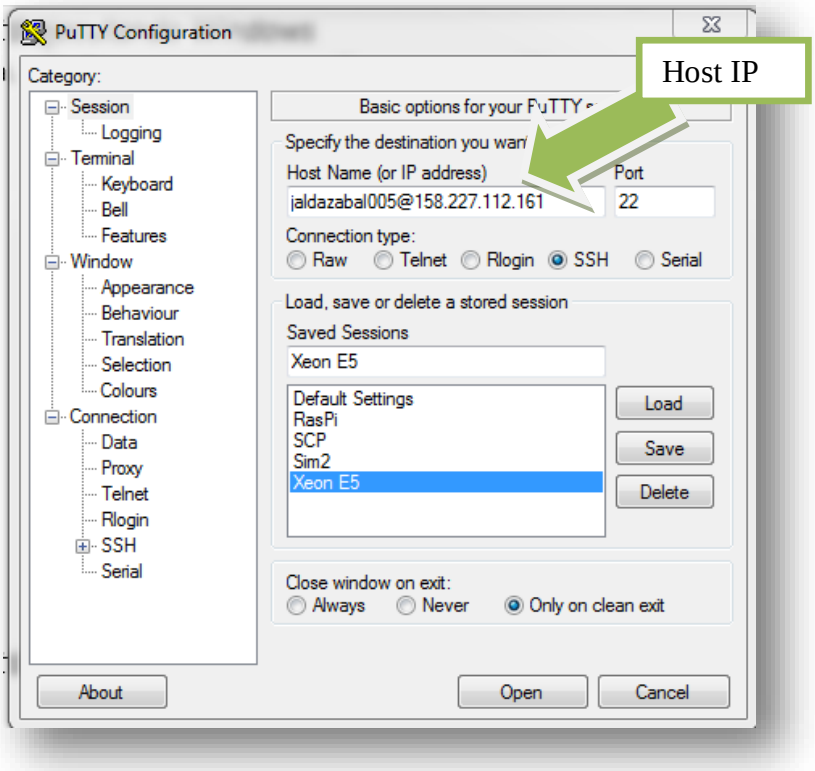


Ilustración 2.5 Conexión a host Linux

- En cualquiera de los dos casos anteriores hay que establecer las **variables de entorno** antes de utilizar nuestro sistema ejecutando la siguiente instrucción, teniendo en cuenta nuestra ruta de instalación.

```
"C:\Program Files (x86)\Intel\Parallel Studio XE 2015\psxevars.bat" intel64
```

- Una vez dentro del host podemos conectarnos al coprocesador por medio de una **sesión SSH**, ya sea con Putty en Windows (ilustración 2.6) o con la instrucción '*ssh root@mic0*' en Linux.

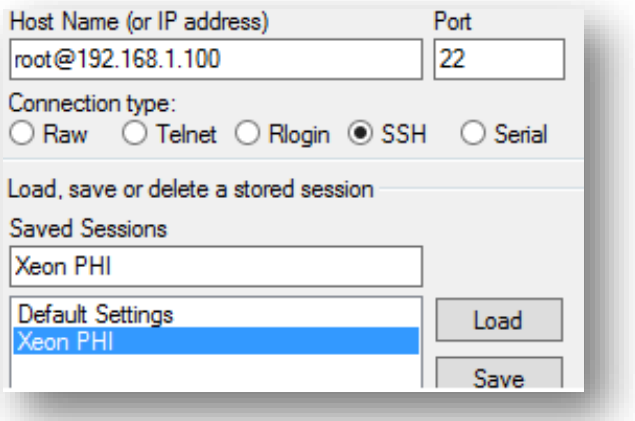


Ilustración 2.6 Conexión a Xeon Phi

- Para utilizar la **ejecución nativa** compilar los ejecutables con el flag /Qmic o –mmic:
    - Transferencia del ejecutable y las librerías necesarias (situadas en "C:\Program Files (x86)\Intel\Composer XE 2015\compiler\lib\mic" en Windows, como la de openmp) al coprocesador mediante WinSCP o el comando SCP, *Secure Copy*.
    - Colocarse en el directorio y establecer la ruta de las librerías (export LD_LIBRARY_PATH=/root/user), tal como señala la ilustración 2.7.
    - Establecer las variables de entorno que deseemos, como podría ser controlar el número de threads con OMP_NUM_THREADS=112.

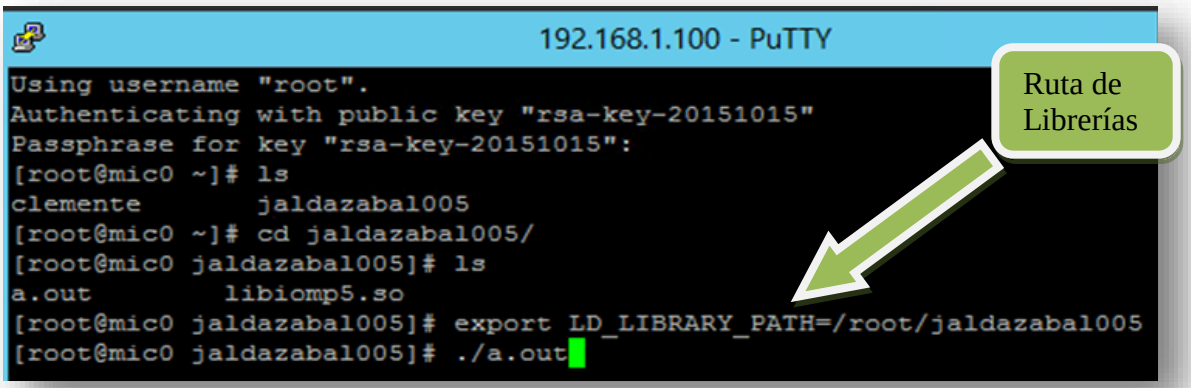


Ilustración 2.7 Ejecución Xeon Phi

2.1.2.- Monitorización

Para utilizar la herramienta de monitorización del coprocesador SMC, *System Management and Configuration,* (temperatura, uso de CPU y memoria…) en el terminal escribir el comando "**micsmc**" . En Windows arrancara una interfaz gráfica o GUI, *graphical user interface,* como la mostrada en la ilustración 2.8.

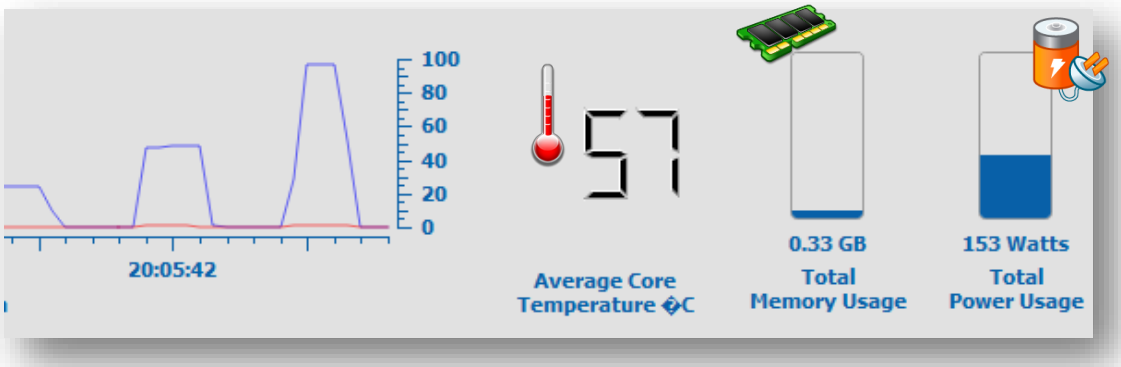


Ilustración 2.8 Monitorización Xeon Phi GUI

En Linux para obtener información acerca del estado del Xeon Phi usaremos el comando micmsc junto con parámetros como podemos apreciar en la ilustración 2.9.

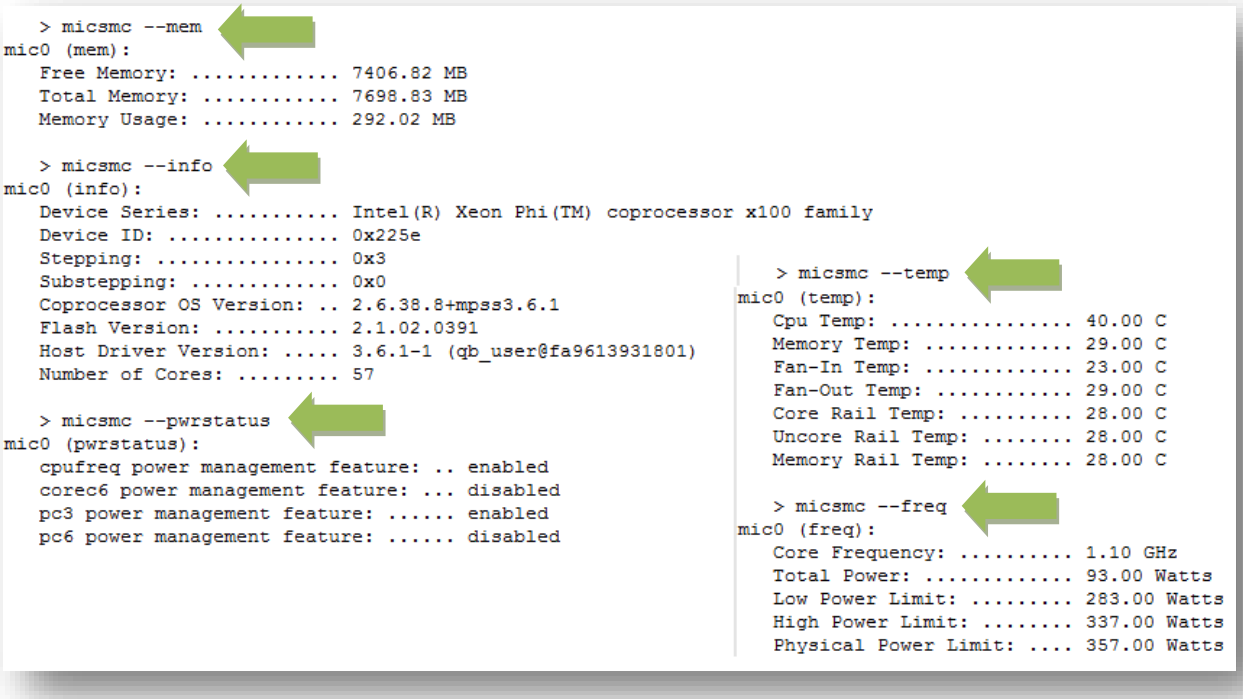


Ilustración 2.9 Monitorización Xeon Phi texto

También podemos obtener un resumen de las características de los coprocesadores instalados con la instrucción "micinfo", tales como la descripción general (ilustración 2.10) o la memoria y cores (ilustración 2.11).

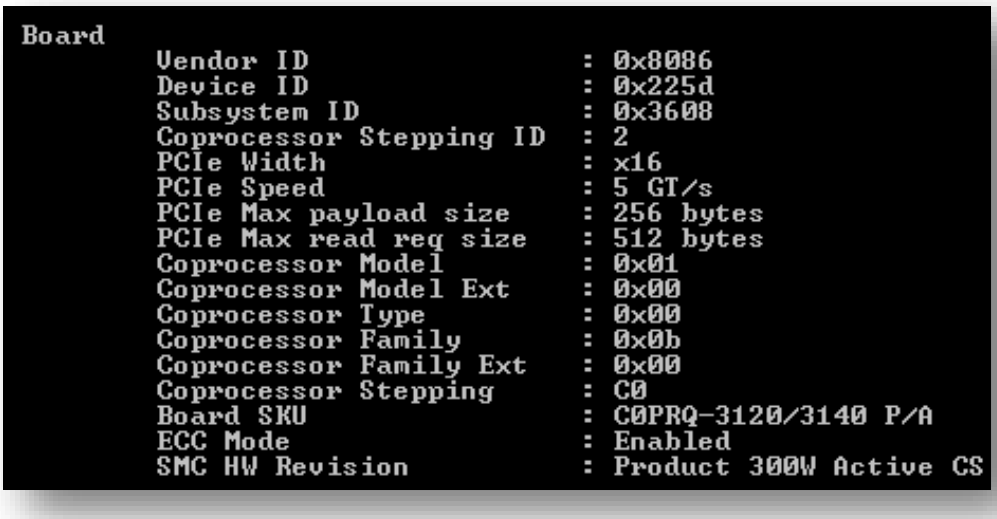


Ilustración 2.10 Información chipset Xeon Phi

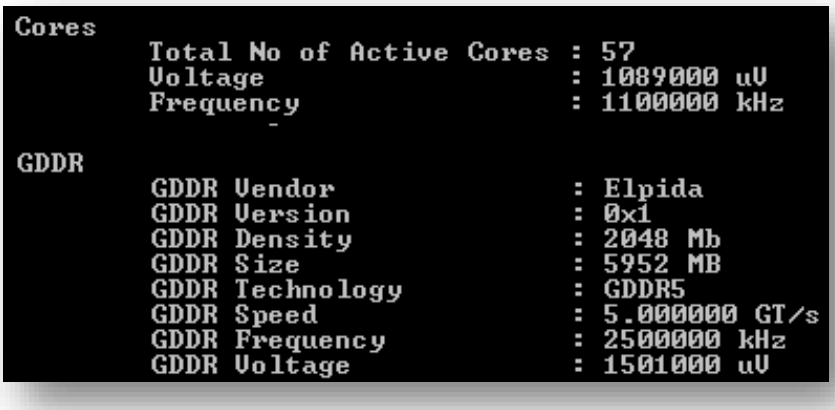


Ilustración 2.11 Información memoria Xeon Phi

2.1.3.- Compilación y desarrollo

En Windows utilizaremos el terminal que vemos en la ilustración 2.12 para usar el compilador de Intel en los programas desarrollados, para construir aplicaciones que se ejecutan en arquitecturas Intel 64 o Intel MIC, *Many Integrated Core*.

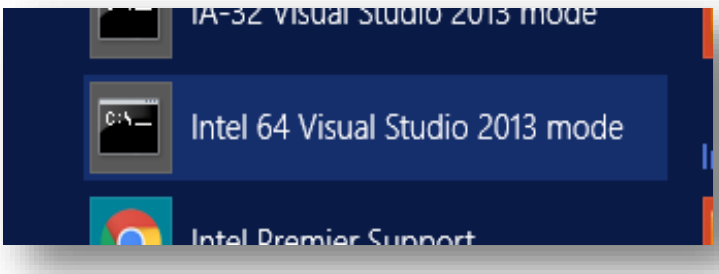


Ilustración 2.12 Compilador ICC para Intel 64

Una buena manera de hacer más eficiente la compilación es crear archivos **.bat,** *batch file*, y escribir ahí las instrucciones que deseamos ejecutar.

En Linux, por otra parte, en la misma terminal de CentOS podremos construir las aplicaciones, y de la misma manera crear **scripts** que compilen y lancen las aplicaciones, lo que nos facilitará el trabajo.

Para el **desarrollo** del código se ha utilizado principalmente **Visual Studio** Ultimate 2013 y también Notepad++.

2.1.4.- Herramientas

El diagrama de la ilustración 2.13 muestra las utilidades disponibles tanto para paralelismo como para vectorización, ordenándolas en función de **facilidad de uso** versus **control.**

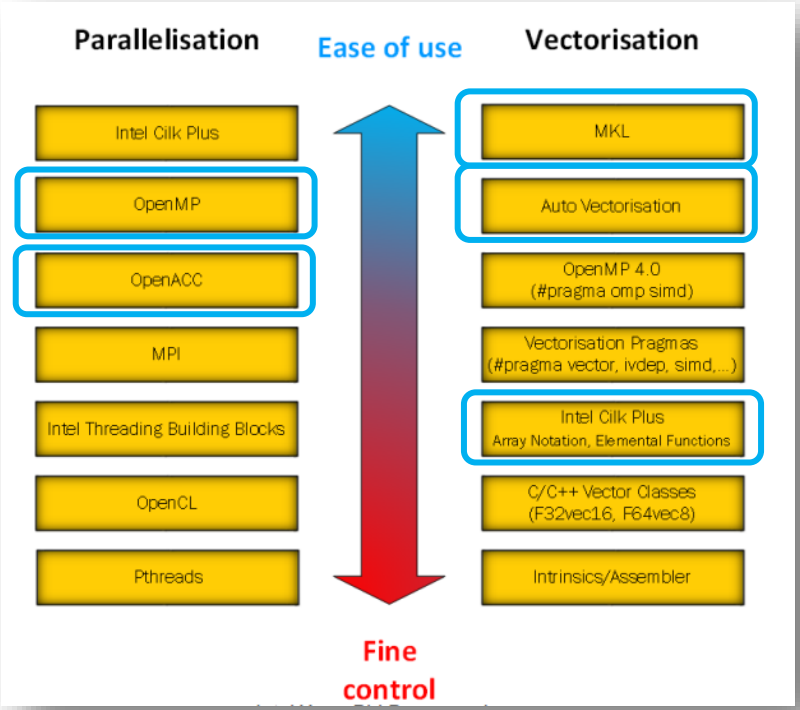


Ilustración 2.13 Herramientas utilizadas

En nuestro caso hemos utilizado:

- **OpenMP** para la paralelización, puesto que es una herramienta estándar y de fácil uso, a la vez que flexible.
- Para la vectorización, hemos usado la librería **MKL** (simplemente llamar a una función externa), la **auto-vectorización** (que se activa a partir de la optimización O2) y también **Intel Cilk Plus Array Notation** para ver el potencial y la fácil descripción de operaciones vectoriales que ofrece.

Estas herramientas nos han servido muy bien durante el desarrollo y son bastante simples de manejar, y esta facilidad de uso es la que buscamos precisamente en el uso de un sistema heterogéneo.

2.2. Intel Xeon

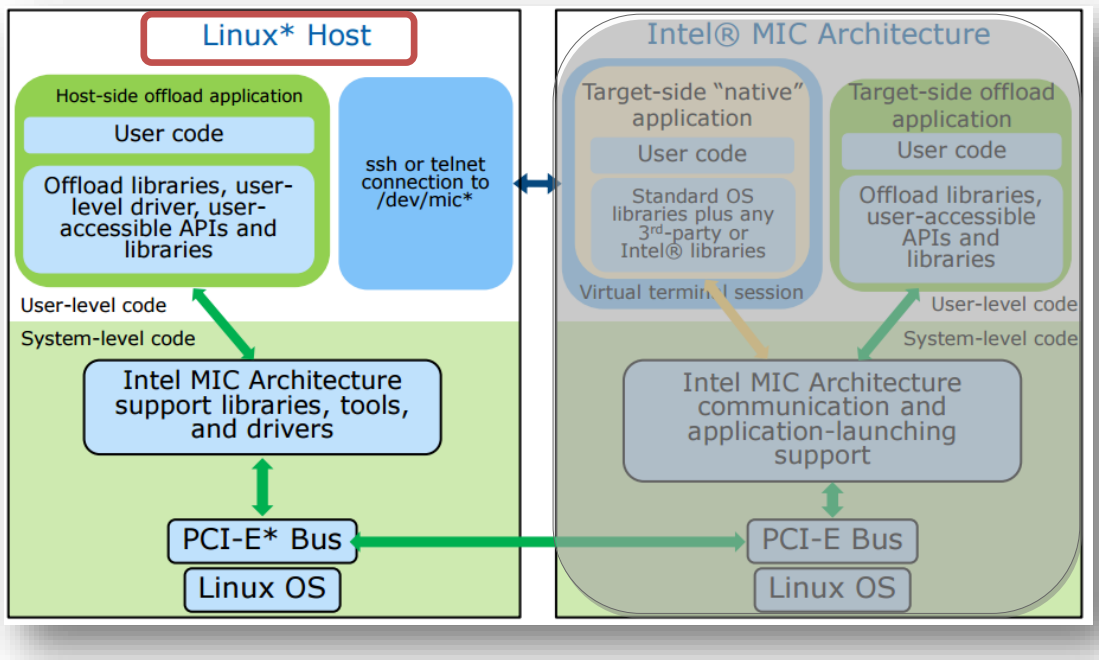


Ilustración 2.14 Introducción Xeon E5

2.2.1.- Arquitectura

MEMORIA

El Intel Xeon E5-1607 cuenta con un tipo de **memoria DDR4 SDRAM,** *Double Data rate* Synchronous Dynamic Random Access Memory. Cuenta con un menor voltaje (1,02 a 1,05), mayor velocidad (2133Mhz) y densidad que su predecesora.
Utiliza la funcionalidad ECC, *Error-Correcting Code*, que verifica la integridad de los datos.

Disponemos en nuestra máquina de 4 módulos como los de la ilustración 2.15, con un ancho de banda máximo de 59GB/s.

Ilustración 2.15 Memoria DDR4

CACHE

La **cache** sigue el esquema de organización de la ilustración 2.16.

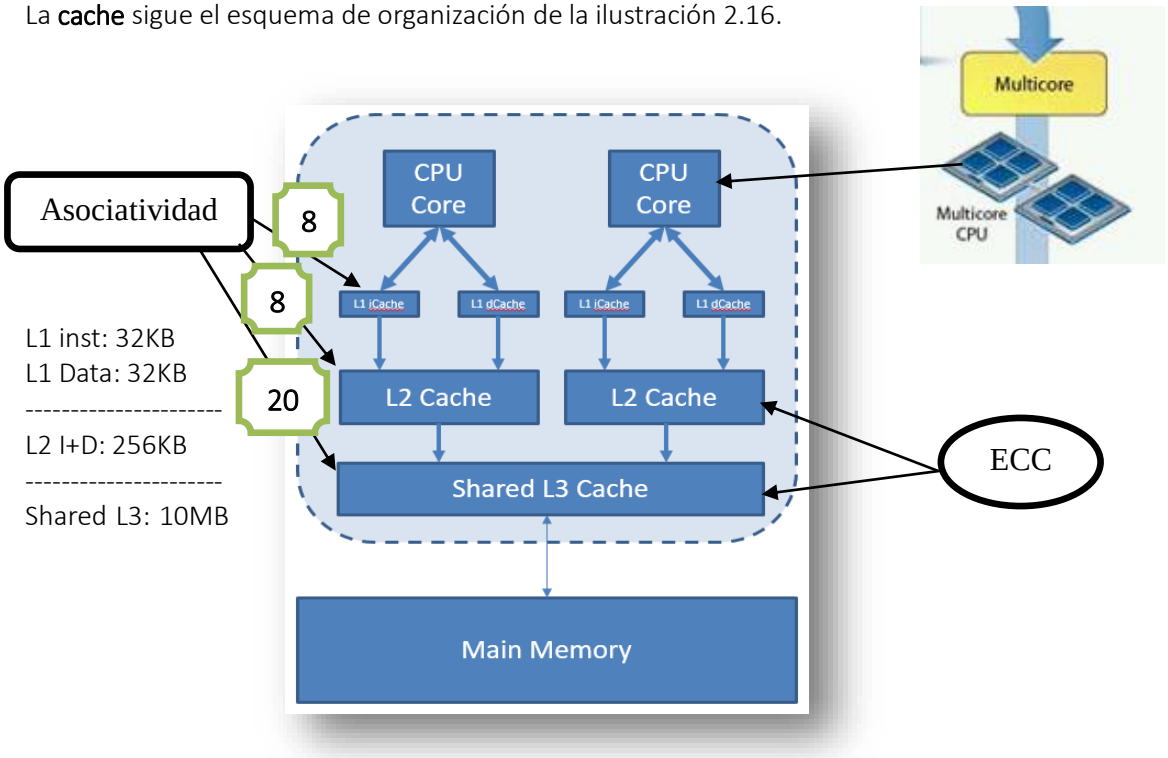


Ilustración 2.16 Descripción cache host

SET DE INSTRUCCIONES

Tiene soporte para el conjunto de instrucciones **AVX2,** *Advanced Vector eXtension*, lo que permite operar con registros SIMD, *Single Instruction Multiple Data*, de **256 bits** de ancho. Estos se llaman registros YMM y hay un total de 16. Es capaz también de realizar operaciones **FMA,** *Fused Add-Multiply*, que realizan multiplicación y adición de elementos coma flotante en el mismo paso.

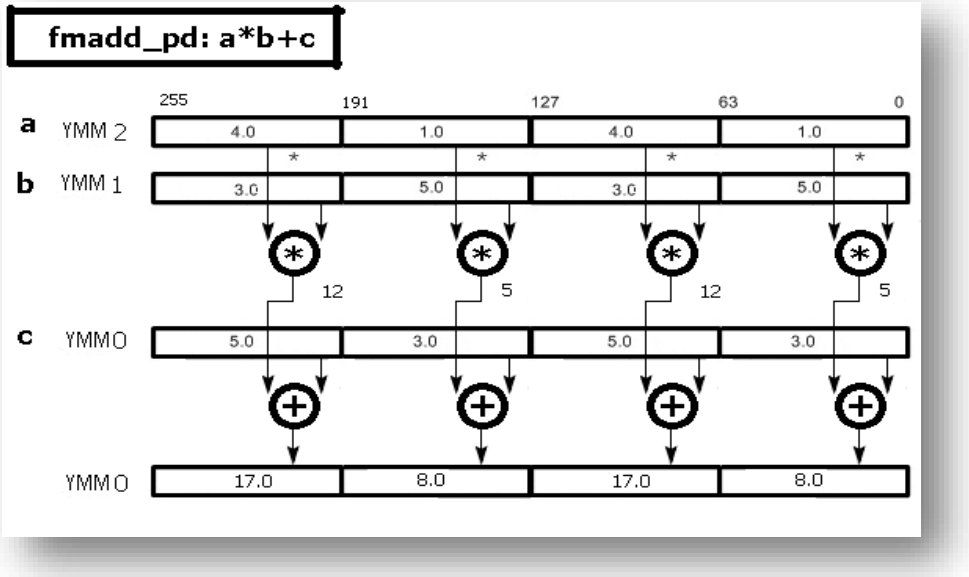


Ilustración 2.17 Set de instrucciones AVX2 con FMA

PROCESADOR

El procesador Xeon E5-1607 v3 tiene **4 núcleos**, con 1 thread de ejecución cada uno, y la tecnología de fabricación es de 22 nanómetros.

El **rango de voltaje** o VID, *Voltage Identification*, del procesador va desde 0.65V a los 1.30V, y es un indicador de los valores de voltaje mínimo y máximo a los que el procesador está diseñado para funcionar. Su **TDP**, *thermal design power,* que es la máxima cantidad de potencia permitida por el sistema es de 140W.

La **frecuencia base** es de 3.10 Ghz (Velocidad del bus frontal 99.76Mhz * Multiplicador CPU 31 = 3092,56 Mhz).

TECNOLOGÍAS INTEL

- No dispone de tecnología **Turbo Boost** (tecnología que hace que el procesador sea capaz de aumentar su frecuencia de funcionamiento, de forma automática en determinadas circunstancias).

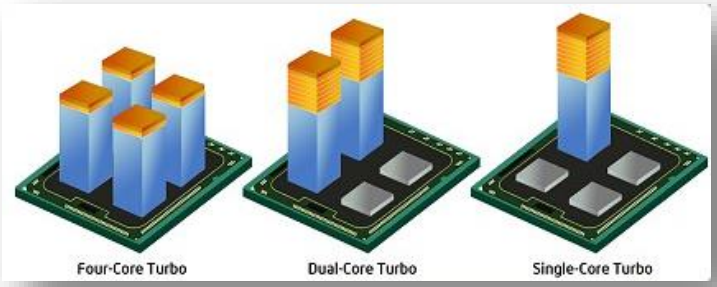


Ilustración 2.18 Intel Turbo-Boost

- No dispone de **Hyper-Threading** (simulación de procesadores lógicos en uno físico para mejor aprovechamiento de recursos y por tanto aumento de rendimiento).

RESUMEN

Utilizando la aplicación CPU-Z podemos obtener una visión general sus características técnicas:

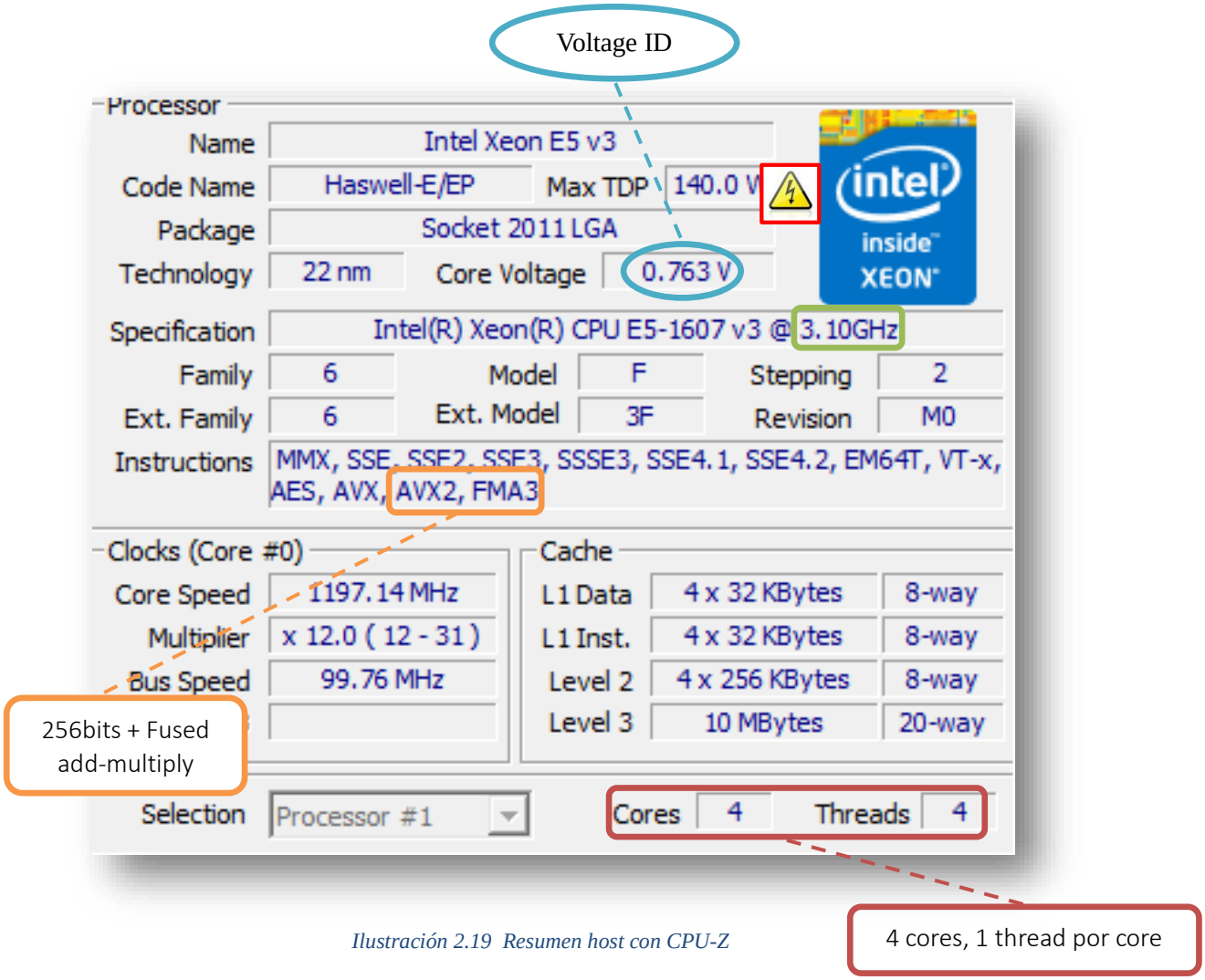


Ilustración 2.19 Resumen host con CPU-Z

2.2.2 Consumo Idle

Los C-Estados determinan el estado de reposo de los núcleos, donde C0 es el operacional mientras que C1 y C6 son estados idle. Desde C0 las CPUs van deteniéndose hasta que tras quedar todas *clock-gated* (técnica usada para reducir el consumo dinámicamente) pasan a C1, como vemos en la ilustración 2.20.

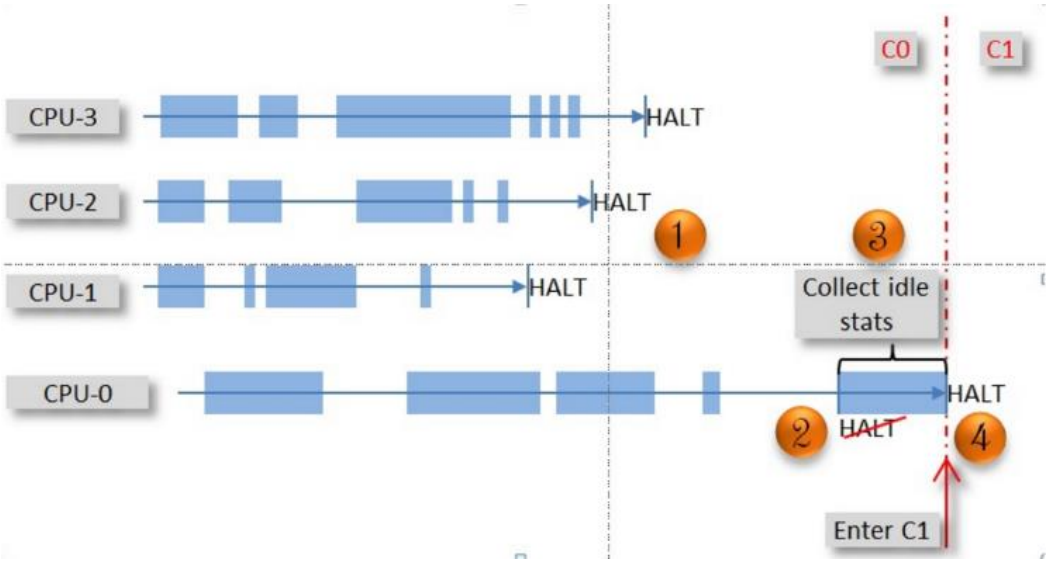
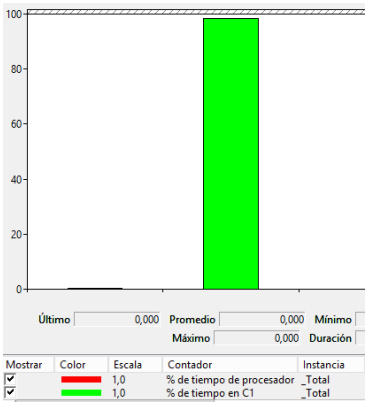


Ilustración 2.20 Transición C0 a C1

Estas transiciones de estados son técnicas de ahorro energético puesto que, si no se está utilizando algo, para qué mantenerlo encendido? El procedimiento es parecido para transiciones a estados superiores y que realizan *clock-gate* en más recursos. Sin embargo, esta técnica puede resultar ineficiente si las interrupciones vuelven enseguida, por lo que existe una rutina de gestión de consumo que trata de predecirlas y actuar en consecuencia.

En los gráficos 2.21 y 2.22 se puede observar al Host manteniendo un estado de baja energía C1 cuando no ejecuta código, y cómo pasa a operacional/C0 cuando se lanza un proceso que ocupa todas las CPUs.



Ilustracion 2.21 Estado C1

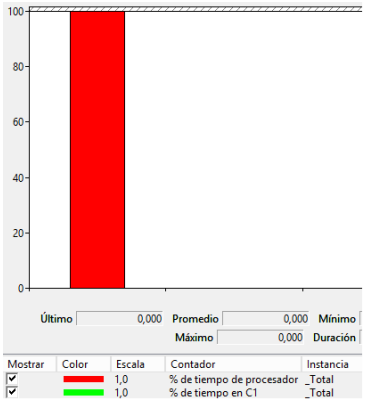


Ilustración 2.22 Estado C0

Utilizando la herramienta Intel Power Gadget 3.0, se muestra **medición media** del host en cuanto a consumo del procesador y la memoria a lo largo 30 segundos en la ilustración 2.23. Se puede ver también con otra herramienta (AIDA64) cómo las CPUs no están siendo utilizadas (Idle) en 2.24.

```
154  Total Elapsed Time (sec) = 30.540570
155  Measured RDTSC Frequency (GHz) = 3.093
156
157  Average Processor Power_0 (Watt) = 13.717078
158
159  Average DRAM Power_0 (Watt) = 5.489974
160
```

Ilustración 2.23 Medición consumo Intel Power Gadget 3.0

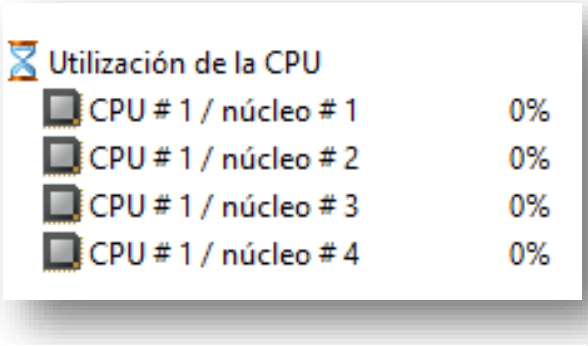


Ilustración 2.24 Uso de cores AIDA64

2.3. Intel Xeon Phi

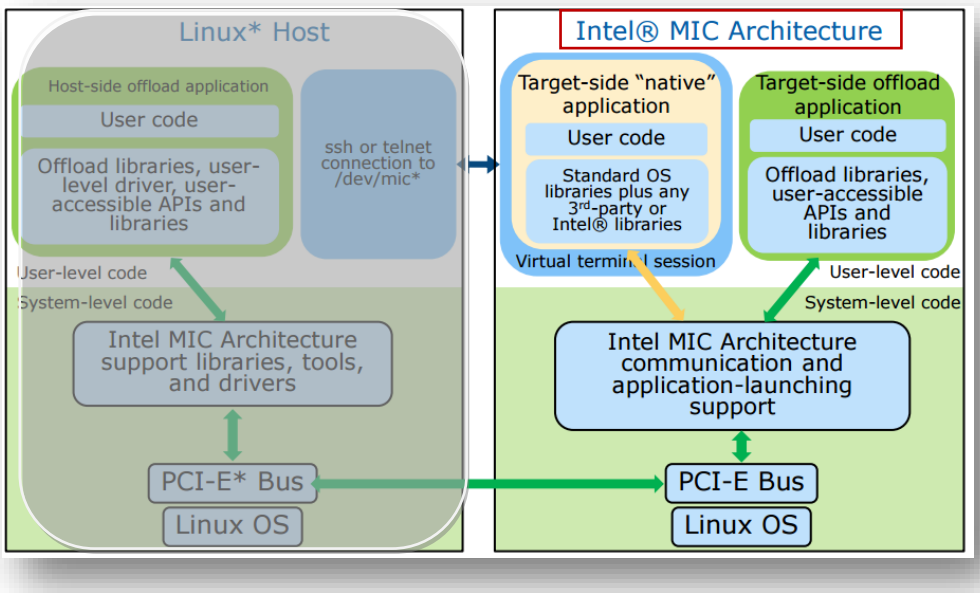


Ilustración 2.25 Introducción Xeon Phi

2.3.1. Arquitectura

MEMORIA

La memoria principal del coprocesador Intel Xeon Phi, es de tipo **GDDR5** y tiene una capacidad de **6GB** (3 módulos de 2GB) tal y como muestra la instrucción "micinfo" en la ilustración 2.26.

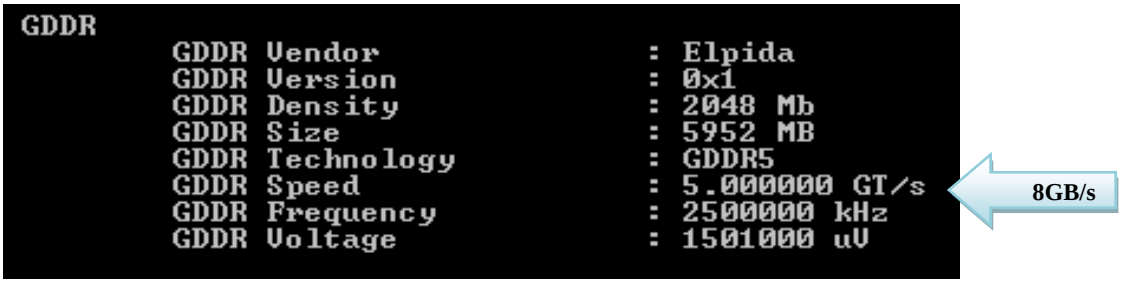


Ilustración 2.26 Memoria principal Xeon Phi

Funcionando a un voltaje de **1,5V**, dispone de **12 canales** para conseguir un ancho de banda máximo de memoria de **240GB/s** a. Cada acceso supone aproximadamente unos 300ns, y dispone de ECC (código de corrección de errores).

CACHE

La organización de la memoria se muestra en la ilustración 2.27.

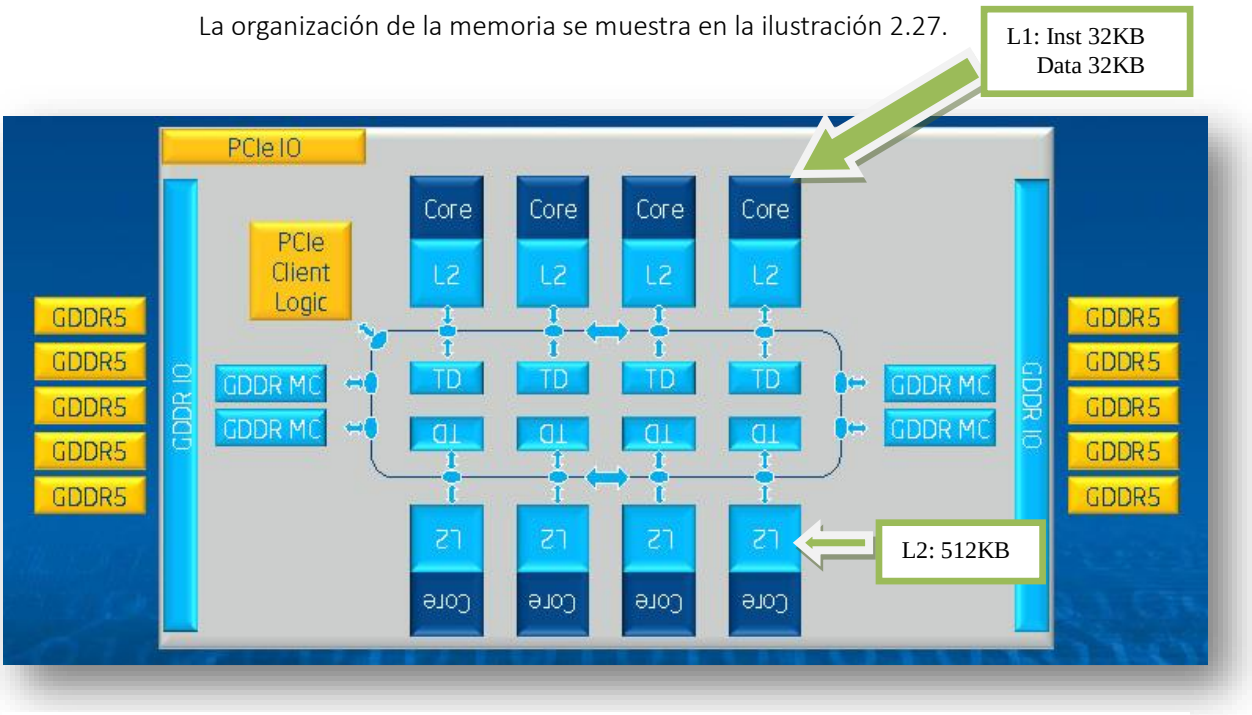


Ilustración 2.27 Descripción cache Xeon Phi

Cada núcleo tiene 32KB de datos y 32KB de instrucciones de nivel 1 exclusivamente. Dispone también 512KB de caché de nivel 2 y todas están interconectadas mediante una topología de bus de anillo bidireccional (hasta 28.5MB con 57 cores), a la que la interfaz PCIe y la memoria principal están también conectadas.

El tiempo de acceso de la caché L1 es de aproximadamente 3 ciclos y el de la caché L2 es en de unos 14 ciclos en el mejor de los casos. Ambas son asociativas de 8 vías y totalmente coherentes, para lo que utilizan el protocolo de coherencia MESI. En el nivel 2 de caché para ello existe un directorio de etiquetado distribuido global, el *Tag-Directory*.

SET DE INSTRUCCIONES

Cada núcleo tiene una Unidad de Procesamiento Vectorial o VPU, *vector processing unit*. Cada unidad vectorial soporta un nuevo juego de instrucciones o ISA, *instruction set arquitecture,* de tipo SIMD conocido como AVX-512. Hay **32 registros vectoriales ZMM de 512 bits** en cada core (pudiendo ejecutar en un ciclo 8 operaciones de coma flotante de doble precisión).

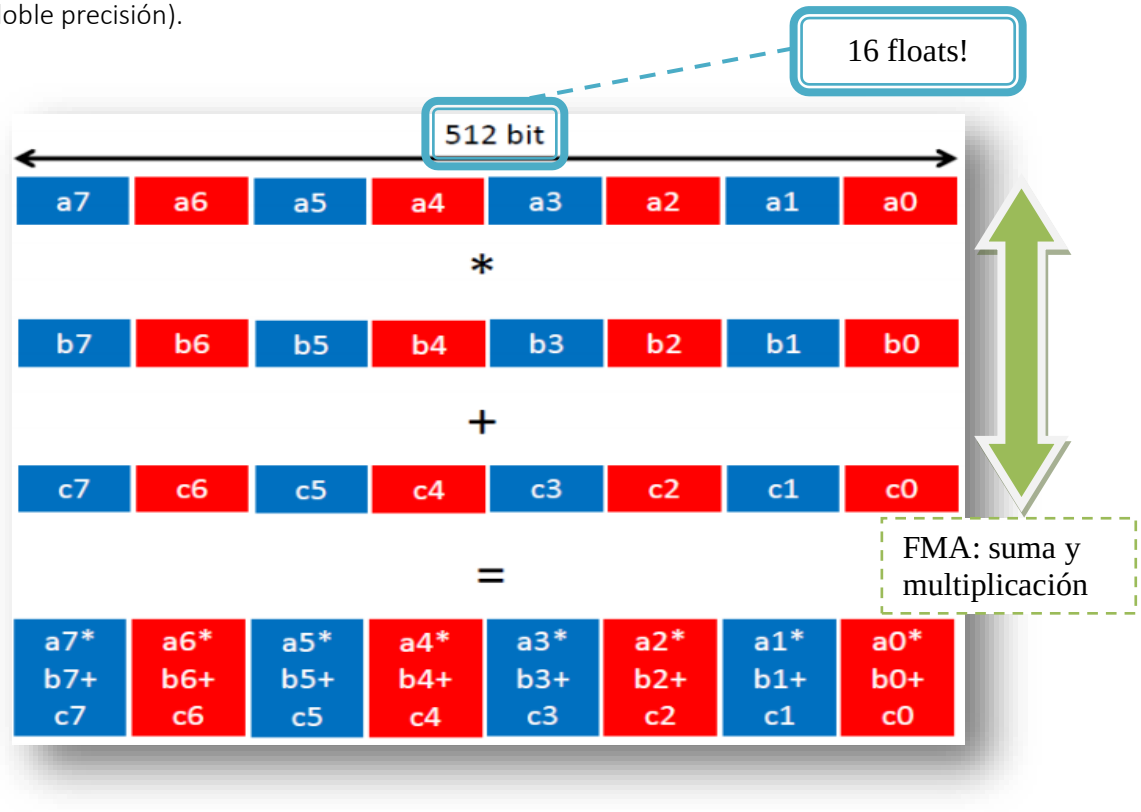


Ilustración 2.28 Set de instrucciones AVX-512 con FMA

La arquitectura soporta **x87**, que es un subconjunto del juego de instrucciones x86. Aprovechar la VPU al máximo es esencial para el mejor rendimiento del coprocesador Intel Xeon Phi. También soporta el estándar **FMA**, que como vemos en la ilustración 2.28 permite ejecutar en el mismo paso una multiplicación junto a una suma.

Podemos apreciar la evolución en cuanto a longitud que han experimentado los juegos de instrucciones a lo largo de los años en la ilustración 2.29.

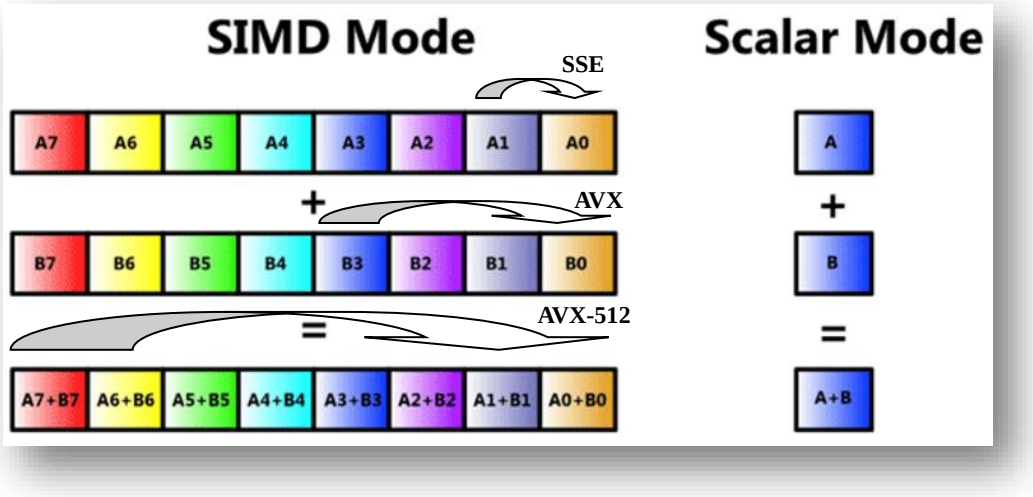


Ilustración 2.29 Evolución tamaño de registros vectoriales

PROCESADOR

El modelo Intel Xeon Phi 3120 es un coprocesador que contiene **57 núcleos** (el sistema operativo corre en uno de ellos), fabricados con una tecnología de 22 nanómetros. Cada uno de ellos tiene una segmentación **in-order** y tiene por hardware hasta **4 threads**, lo que permite ocultar latencias (en total 228). En la ilustración 2.30 vemos el aspecto físico del coprocesador Xeon Phi.

Ilustración 2.30 Aspecto físico Xeon Phi

El **voltaje** de cada procesador es de 1.09V, y su **TDP**, es de 300W. La **frecuencia** de cada uno de ellos es de 1.100Ghz.

Se considera prácticamente un **superescalar de grado 2** debido a los dos canales a donde van las instrucciones decodificadas. La arquitectura de cada procesador está basada en la de un **Pentium II modificada** con soporte de hyperthreading y algunas nuevas instrucciones x86 creadas para aprovechar la unidad vectorial.

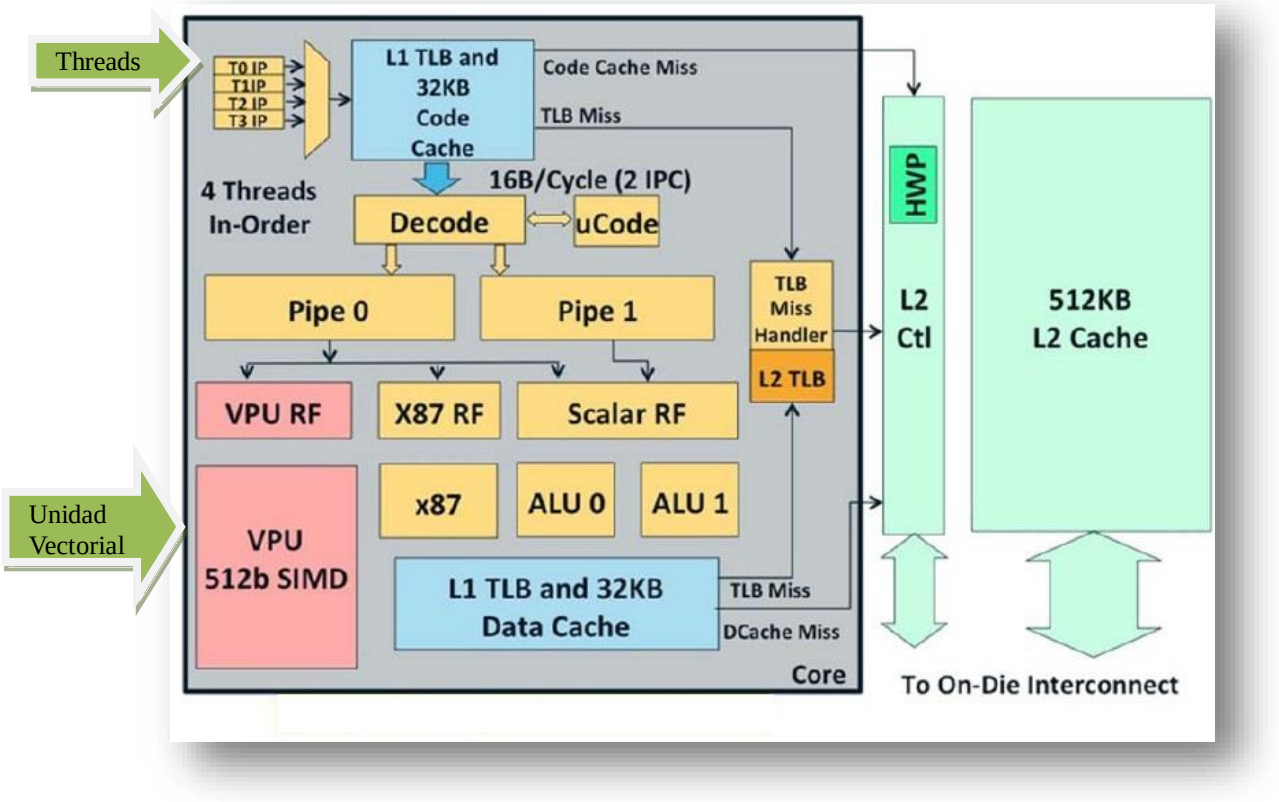


Ilustración 2.31 Descripción interna Xeon Phi

RESUMEN

Es importante señalar que la familia de coprocesadores Xeon Phi, denominada *Knights Corner*, no puede ejecutar por sí mismo los mismos ficheros binarios que otras plataformas Intel, y necesita que el **código se compile de manera explícita** para esta arquitectura.

Un esquema físico del coprocesador se puede ver en la ilustración 2.32:

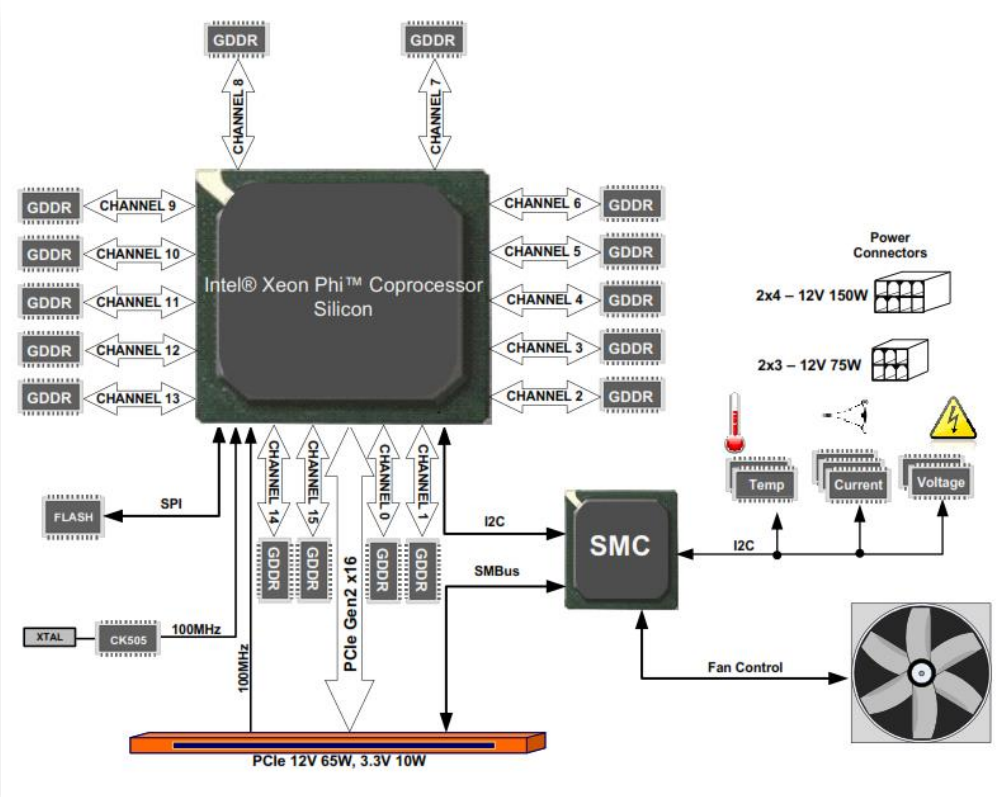


Ilustración 2.32 Esquema Xeon Phi

2.3.2 Consumo Idle

Se puede observar al coprocesador en un **estado de baja energía**, o ***IDLE*** *state*, cuando no se encuentra ejecutando instrucciones, mostrando un menor consumo en comparación a estados más activos.

Hemos de destacar que en el uso de las mediciones de consumo, se interpretan como interrupciones por el sistema operativo y debe despertarse para atenderlas, y por tanto cambiando a un estado de consumo superior (deep-PC3 -> *wake up* -> C0).

Esto hace que aquí las muestras no sean precisas, porque mientras recibimos datos de consumo de 93 vatios, desde Intel y documentación oficial la establecen en **40 vatios**.

```
micsmc --freq

mic0 (freq):
  Core Frequency: ..... 1.10 GHz
  Total Power: ..... 93.00 Watts
  Low Power Limit: ..... 315.00 Watts
  High Power Limit: ..... 375.00 Watts
  Physical Power Limit: .... 395.00 Watts
```

Please note that the power levels displayed cannot be used to determine idle power consumption.

3

Compilación con ICC

El compilador de Intel, también conocido como ICC o ICL, es un conjunto de compiladores para los lenguajes C y C++ desarrollado por Intel. Los compiladores están disponibles para los Sistemas Operativos Linux, Microsoft Windows y MAC OS X.

Estos compiladores pueden funcionar sobre procesadores IA-32, Intel64, Itanium2, y otros procesadores ajenos a la marca, pero compatibles, como los de AMD. El Intel C++ Compiler para IA-32 e Intel 64 dispone de una vectorización automática que puede generar instrucciones SIMD.

El Intel C++ Compiler soporta tanto OpenMP 3.0 como paralelización automática para el multiprocesamiento simétrico. Con el complemento Cluster OpenMP, el compilador también puede generar automáticamente llamadas de Interfaz de Paso de Mensajes O MPI, *message passing interface,* para el multiprocesamiento de la memoria distribuida desde las directivas de OpenMP.

Ilustración 3.1 Versión Intel C++ Compiler

En este capítulo:

- Describiremos el uso intensivo realizado con ICC explicando las optimizaciones.
- Explicaremos el uso de Xeon Phi de manera nativa.
- La programación heterogénea utilizada la desarrollaremos en este apartado.
- Explicaremos el uso de herramientas como la notación array Cilk, el uso de OpenMP o las potentes librerías MKL de Intel.

3.1. Optimizaciones del compilador de Intel

A continuación se detallan las opciones de optimización de programas disponibles en el compilador de Intel C[++], para procesadores IA-32, Intel 64 y otros no Intel que sean compatibles.

Aclaraciones:

- En las compilaciones incluimos siempre el flag -openmp porque hemos utilizado la función de medición de tiempos de esta librería.
- A continuación los flags de compilación se citan para Windows, y en el caso de Linux es sustituir "/Q" por "-" .

Es aconsejable empezar con la opción **/O0** para verificar la validez del programa, y progresivamente aumentar el nivel de optimización con las optimizaciones **/O1, /O2 y /O3**, como vemos en la ilustración 3.2.

Debemos señalar que no se ha escogido –O0 como referencia debido a que el compilador sin especificar nada **establece /O2 por defecto**.

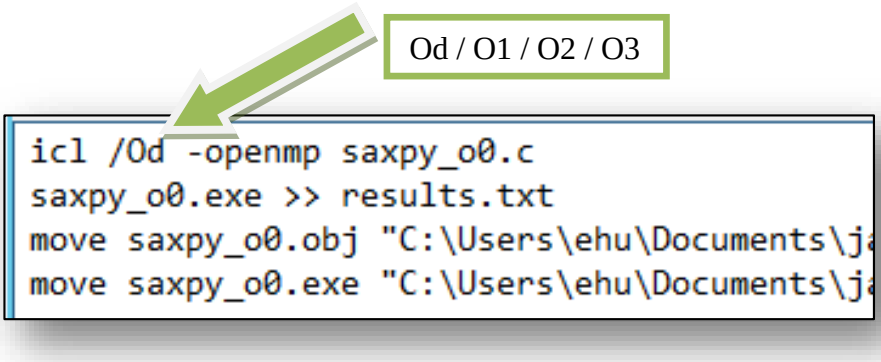


Ilustración 3.2 Compilar 0-1-2-3

En cuanto a la arquitectura, hay una serie de optimizaciones que **adaptan el juego de instrucciones** del programa a la que indiquemos, o **donde se encuentre ejecutando**, por ejemplo en un procesador Xeon E5 o un Xeon Phi en nuestro caso (comprobando soporte para los registros **SSE_X, AVX(2) o AVX-512**).

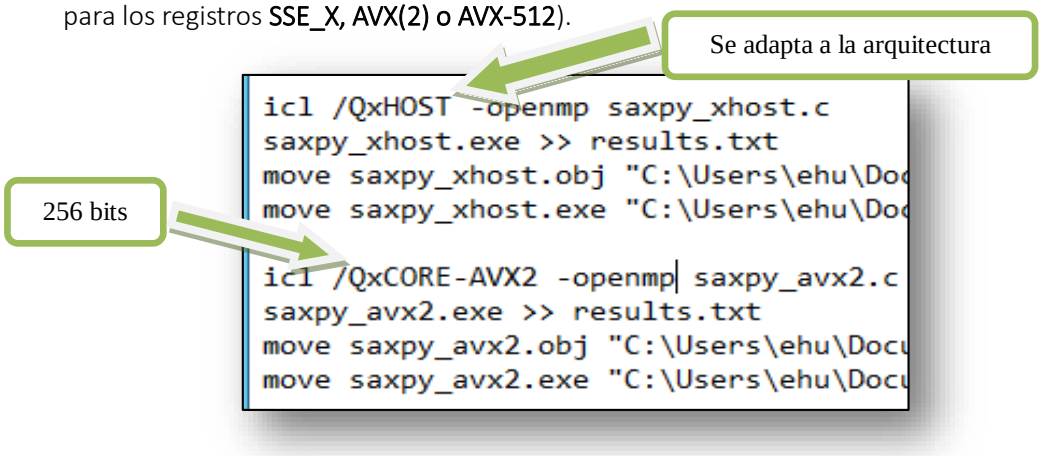


Ilustración 3.3 Compilar xhost / avx2

Existen además **optimizaciones de rendimiento paralelo**, que se utilizan en arquitectura *multi/many core* y que permiten aumentar el rendimiento del programa sacando provecho de este tipo de hardware.

El flag **/Qopenmp** permite interpretar las sentencias '#pragma omp', para utilizar en este caso los cores del procesador que tengamos. También podemos utilizar el flag **/Qopt-threads-per-core:n**, como vemos en la ilustración 3.4, para establecer cuantos threads lanzar en cada core.

```
icl /Qopenmp /Qopt-threads-per-core:1 saxpy_omp.c
saxpy_omp.exe >> results.txt
move saxpy_omp.obj "C:\Users\ehu\Documents\jaldaza
move saxpy_omp.exe "C:\Users\ehu\Documents\jaldaza
```

Ilustración 3.4 Compilar openmp

También es posible utilizar las **librerías matemáticas** optimizadas que ofrece Intel, llamadas MKL y que, explorando distintos modelos de ejecución eligen el más adecuado. Podemos parametrizar este uso a ejecución secuencial o paralela, como refleja la ilustración 3.5.

Hemos de utilizar el flag **/Qmkl**, que puede ser parametrizado de forma secuencial (:sequential) o paralela (:parallel). En caso de querer usar esta librería en una plataforma distinta como Xeon Phi, es obligatorio que al ser una llamada a una función externa en región offload la etiquetemos con **/Qoffload-attribute-target=mic**, y generará así el código correspondiente para esta plataforma.

```
icl /Qopenmp /Qmkl:sequential saxpy_mkl_seq.c
saxpy_mkl_seq.exe >> results.txt
move saxpy_mkl_seq.obj "C:\Users\ehu\Documents\jaldazabal005\saxpy\entero\obj
move saxpy_mkl_seq.exe "C:\Users\ehu\Documents\jaldazabal005\saxpy\entero\exe

icl /Qopenmp /Qmkl:parallel saxpy_mkl_par.c
saxpy_mkl_par.exe >> results.txt
move saxpy_mkl_par.obj "C:\Users\ehu\Documents\jaldazabal005\saxpy\entero\obj
move saxpy_mkl_par.exe "C:\Users\ehu\Documents\jaldazabal005\saxpy\entero\exe

::icl /Qopenmp /Qoffload-attribute-target=mic /Qmkl:parallel saxpy_mkl_off.c
::set OFFLOAD_REPORT=2
::saxpy_mkl_off.exe >> results.
::move saxpy_mkl_off.obj "C:\Users\ehu\Documents\jaldazabal005\saxpy\entero\
```

Activar monitorización Offload

Ilustración 3.5 Compilar mkl

Siempre es interesante obtener información de estas optimizaciones para comprobar en qué podemos mejorar, ya sea en cuanto a vectorización, paralelización… y por eso de cada compilación se pueden **generar reportes** que indiquen esto con distintos niveles de detalle.

A continuación veremos ejemplos de reportes de diferentes fases:

1. Reporte de **vectorización**: utilizar el flag /Qvec-report:n [niveles 1 a 5 (3)], que nos muestra en la ilustración 3.6 si el bucle ha sido vectorizado, que accesos a memoria se realizan y si están alineados, y una estimación del potencial speedup entre otras.

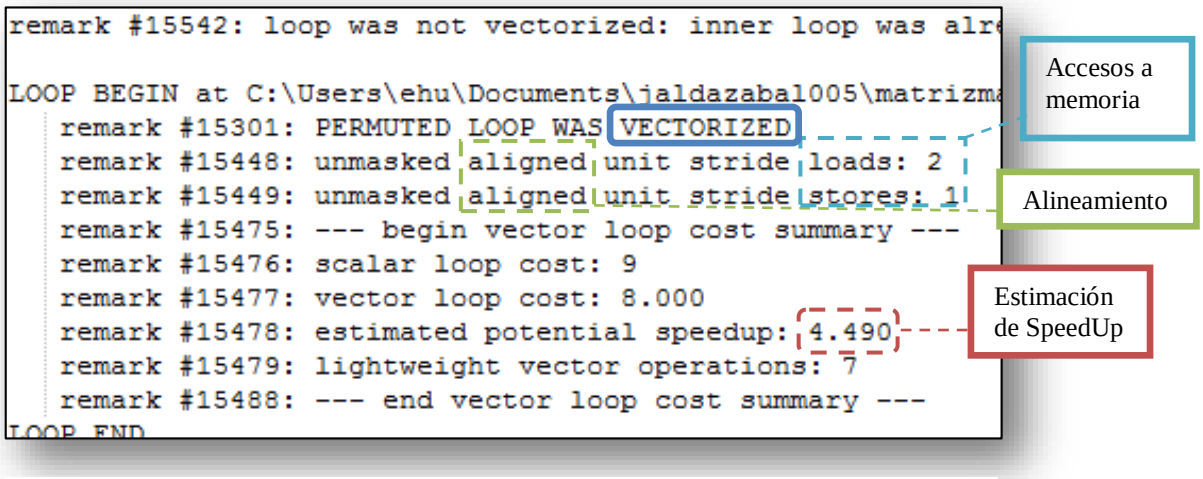


Ilustración 3.6 Reporte de vectorización

2. Reporte de **paralelización**: utilizar el flag /Qopt-report-phase=openmp, y refleja si la región que hemos definido como 'omp parallel' ha sido efectivamente paralelizada.

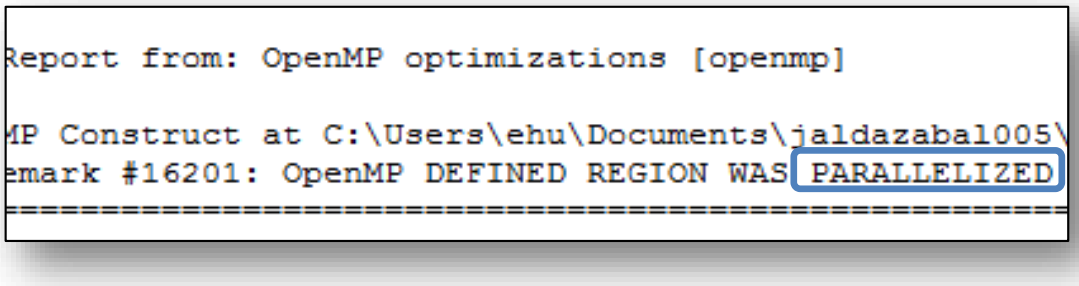


Ilustración 3.7 Reporte de paralelismo

En el fragmento de código de la ilustración 3.8, mostramos un ejemplo donde hemos aplicado la generación de reportes de las ilustraciones 3.6 y 3.7, señalando donde se producen estas optimizaciones.

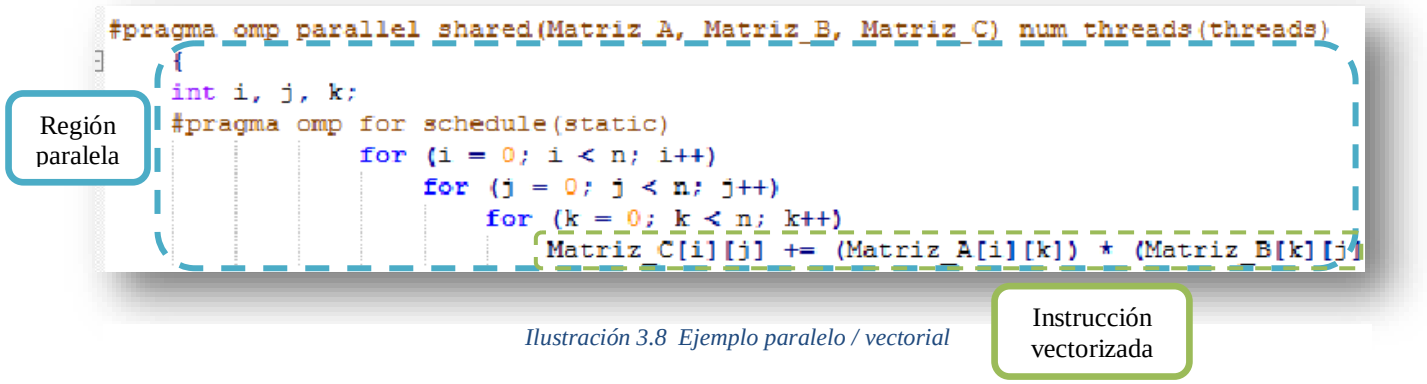


Ilustración 3.8 Ejemplo paralelo / vectorial

A continuación resumimos las principales optimizaciones del compilador de Intel, y utilizadas en el presente proyecto.

Opciones Generales de Optimización

Windows	Descripción
/Od	**Sin Optimización**. Usado en primeras fases de desarrollo del programa y depuración. Usar un ajuste mayor cuando el programa funciona correctamente.
/O1	**Optimizar para tamaño**. Omite las optimizaciones que incrementan el tamaño del objeto, y crea código optimizado de menor tamaño en la mayoría de los casos.
/O2	**Maximiza la velocidad**. Ajuste por defecto. Activa muchas optimizaciones, incluyendo vectorización. Crea un código más rápido que /O1 en la mayoría de los casos.
/O3	Activa las optimizaciones de /O2, y **añade optimizaciones de bucle** más agresivas y de acceso a memoria, como sustitución escalar, desenrollado de bucles, réplica de código para eliminar saltos, bloque de bucles para permitir un uso más eficiente de la caché y captación previa de datos adicional (data prefetch).
/Qopt-report[n]	Genera un informe de optimización , en un fichero con extensión .optrpt. **n** especifica el nivel de detalle, desde 0 (sin informe) a 5 (máximo). Por defecto 2.
/Qopt-report-phase=[arg1, arg2…]	Genera un informe específico de optimización, indicándole uno o varios aspectos determinados (all, loop, vec, par, openmp, ipo, pgo, offload).

Tabla 3.1 Optimizaciones generales

Optimizaciones de rendimiento paralelo

Windows	Descripción
/Qopenmp	Genera **código multihilo** cuando aparecen directivas OpenMP.
/Qparallel	**Detección automática de bucles** simples estructurados que se pueden ejecutar de forma segura en paralelo, incluyendo la notación array *Intel Cilk Plus*, y genera código multihilo automáticamente para estos bucles.
/Qguide[=n]	Hace que **el compilador muestre consejos** para ayudar a vectorizar o autoparalelizar los bucles (no genera objetos o ejecutables).
/Q[no-]opt-matmul	Activa o desactiva la llamada a una **librería de multiplicación de matrices**, identificando los bucles anidados correspondientes y utilizando una función para ello de rendimiento mejorado. Aparece con O3+parallel.
/Qmkl: arg	Enlaza la librería **Intel MKL**, cuyos parámetros son: **parallel**: enlace a la parte multihilo de Intel MKL (por defecto). **sequential**: enlace a la parte monohilo de Intel MKL. **cluster**: enlaza las partes grupal y secuencial de Intel MKL.

Tabla 3.2 Optimizaciones parallel

Optimizaciones basadas en la arquitectura

Windows	Descripción
/Qxarg	Genera **código específico** para procesadores de Intel que soporte el conjunto de instrucciones indicado en 'arg', que puede tomar los valores: CORE-AVX512, MIC-AVX512, CORE-AVX2, AVX, SSE4.2, ATOM_SSE4.2, SSE4.1, ATOM_SSSE3, SSSE3, SSE3, SSE2.
/Qxhost	Genera **conjuntos de instrucciones adaptados** al mayor que pueda soportar el compilador anfitrión.
/Qoffload-attribute-target:mic	**Etiqueta las funciones** externas u objetos de datos con el atributo offload para posibilitar su ejecución en el coprocesador.
/Qmic	Construye un programa que **ejecuta nativamente** en coprocesadores Intel Xeon Phi.
/Qopt-threads-per-core[=n]	Indica al compilador **optimizar para n hilos** por procesador físico. n=[1-4].

Tabla 3.3 Optimizaciones arquitectura

3.2. Compilación y ejecución nativa en Xeon Phi

Uno de los modos de ejecución es la ejecución independiente de programas en el coprocesador, y son necesarios ciertos pasos que detallaremos a continuación.

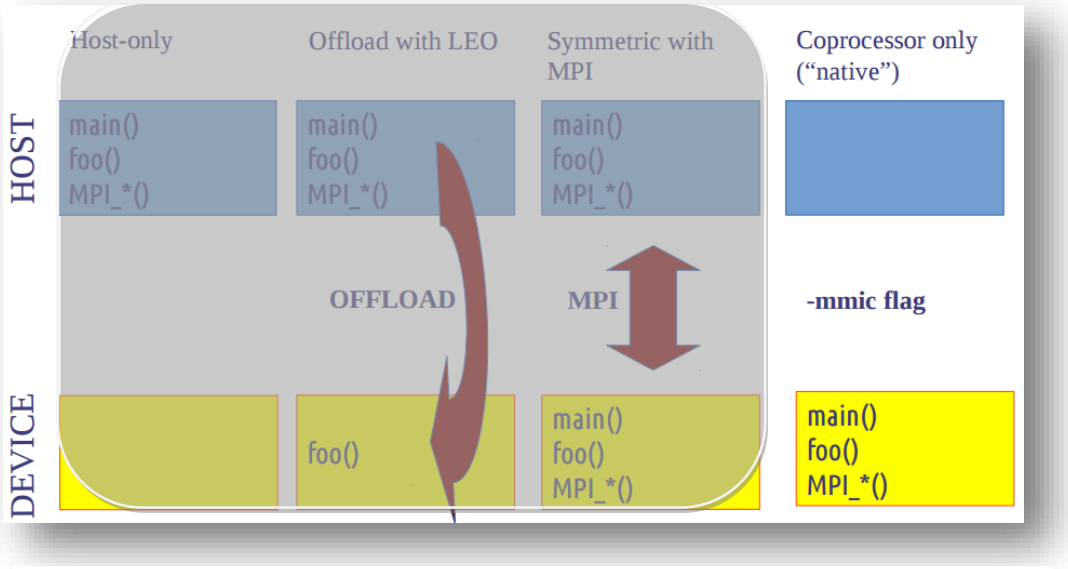


Ilustración 3.9 Introducción modo nativo

El modo de ejecución nativo implica iniciar directamente los programas en sí mismo sin necesidad de comunicación con el Host, pues el main() se inicia en el Xeon Phi.

- Utilización de compilador cruzado, con el flag /Qmic o -mmic, para generar código para la arquitectura MIC, como en la ilustración 3.10.
- Librerías en "~\Intel\Composer XE 2015\compiler\lib\mic".
- Con SCP/WinSCP enviamos los ejecutables y librerías, mientras que con SSH entramos en el sistema operativo del Xeon Phi, y se requieren credenciales.
- Establecer la ruta de las librerías (export LD_LIBRARY_PATH=/root/user).

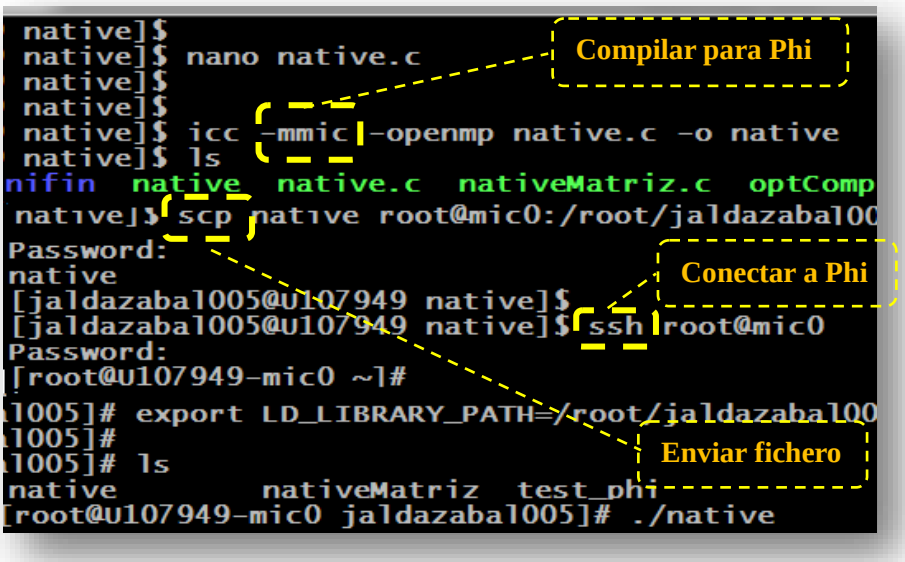


Ilustración 3.10 Modo ejecución nativa

3.3. Programación heterogénea

La computación heterogénea implica la ejecución de procesos en sistemas que utilizan más de un tipo de procesador, es decir, diferentes arquitecturas de conjuntos de instrucciones.

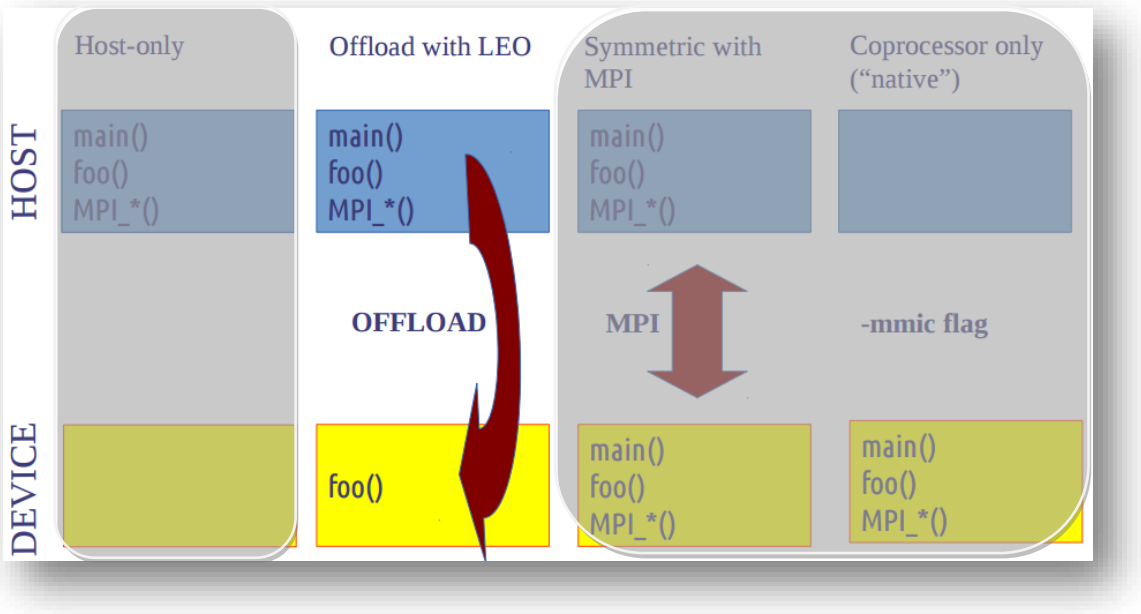


Ilustración 3.11 Introducción modo heterogéneo

Es entonces necesario que el compilador pueda crear código ejecutable para ambas plataformas, y para ello hay herramientas que permiten definir las regiones en el código.

El flujo del programa **se inicia en el host** y cuando se encuentra con estas regiones inicia la ejecución en la otra plataforma, como muestra la ilustración 3.11. Una vez terminada la ejecución en el dispositivo, se devuelve todo el control al anfitrión. El procedimiento es similar al de una subrutina (veremos que también es posible la ejecución simultánea).

3.3.1 Offload

Las **Extensiones de Lenguaje para Offload**, son un conjunto de directivas con cláusulas que permiten establecer un bloque de instrucciones que se ejecutarán en el coprocesador indicado. Trabajaremos con el modelo explícito, donde el programador controla el movimiento de los datos.

Existen 3 modelos de transferencia entre el host y el coprocesador, como describe la tabla 3.4.

Pragma	Sintaxis	Significado
Offload pragma	*#pragma offload <cláusulas> <sentencias>*	Permite a la siguiente sentencia ejecutarse en el coprocesador
Función/Variable offload	*__declspec(target(mic)) función*	Compila la función o reserva la variable para host y mic.
Bloques de datos	*#pragma offload_attribute(push, target(mic))* *...* *#pragma offload_attribute(pop)*	Marca grandes bloques de código para compilar en el host y en el mic

Tabla 3.4 Modelos de transferencia

OFFLOAD SÍNCRONO

El código es instrumentado con directivas, y el compilador crea un binario para la CPU y para el coprocesador, para el bloque definido.

En el flujo del programa saxpy_offload_sync, como vemos en la ilustración 3.12, el host espera a que el mic termine su ejecución,.

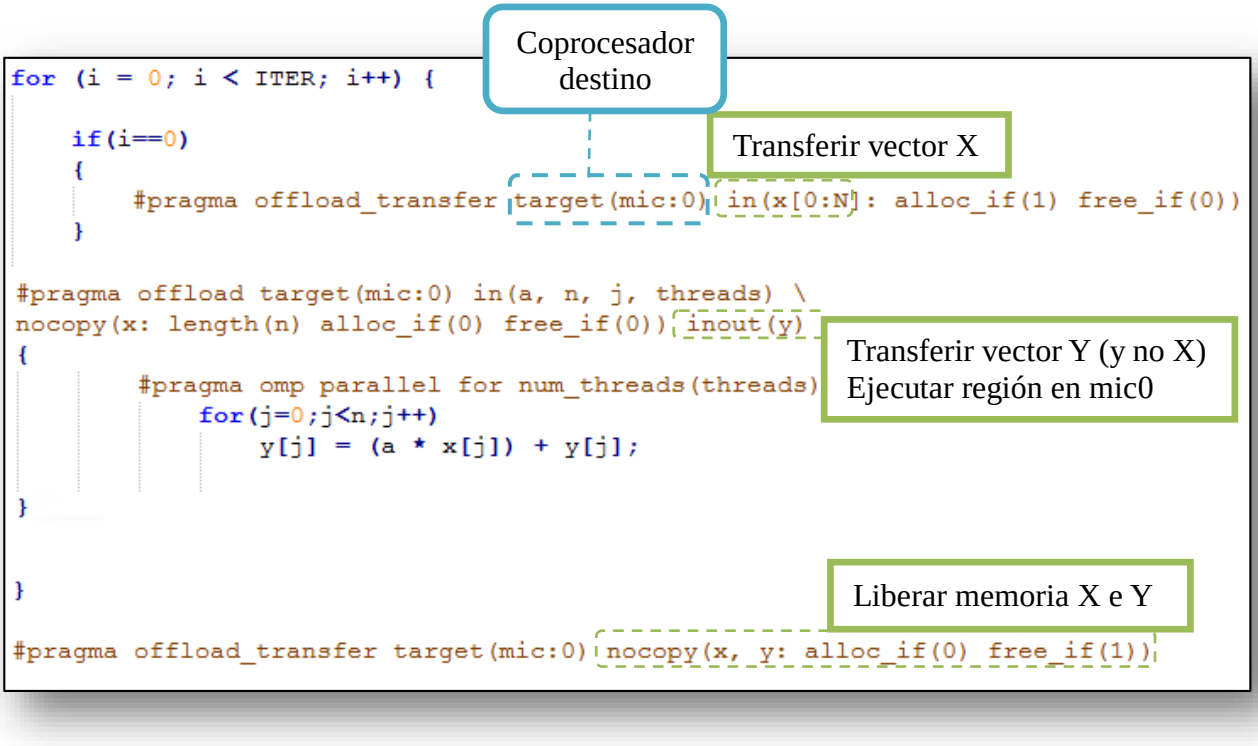


Ilustración 3.12 Offload síncrono

Las directivas que se utilizan para gestionar la memoria en el coprocesador son:

- Alloc_if(1) free_if(1) = reservar y liberar.
- Alloc_if(1) free_if(0) = reservar y no liberar.
- Alloc_if(0) free_if(1) = no reservar y liberar.
- Alloc_if(0) free_if(0) = no reservar y no liberar.

OFFLOAD ASÍNCRONO

Las cláusulas de sincronización se asocian entre ellas a través de un valor identificador o etiqueta.

- Una sentencia *offload_transfer* acompañada de la cláusula **signal** inicia una transferencia de datos asíncrona.

```
#pragma offload_transfer target(mic:n) signal(tag)
```

- Una sentencia *offload* acompañada de la cláusula **signal** inicia cómputo asíncrono.

```
#pragma offload target(mic:n) signal(tag)
```

- Una sentencia *offload_wait* acompañada de la cláusula **wait** bloquea la ejecución hasta que se complete el cómputo o la transferencia de datos asíncrona.

  `#pragma offload_wait target(mic:n) wait(tag)`

Un ejemplo trabajado con offload asícrono, se muestra en la ilustración 3.13.

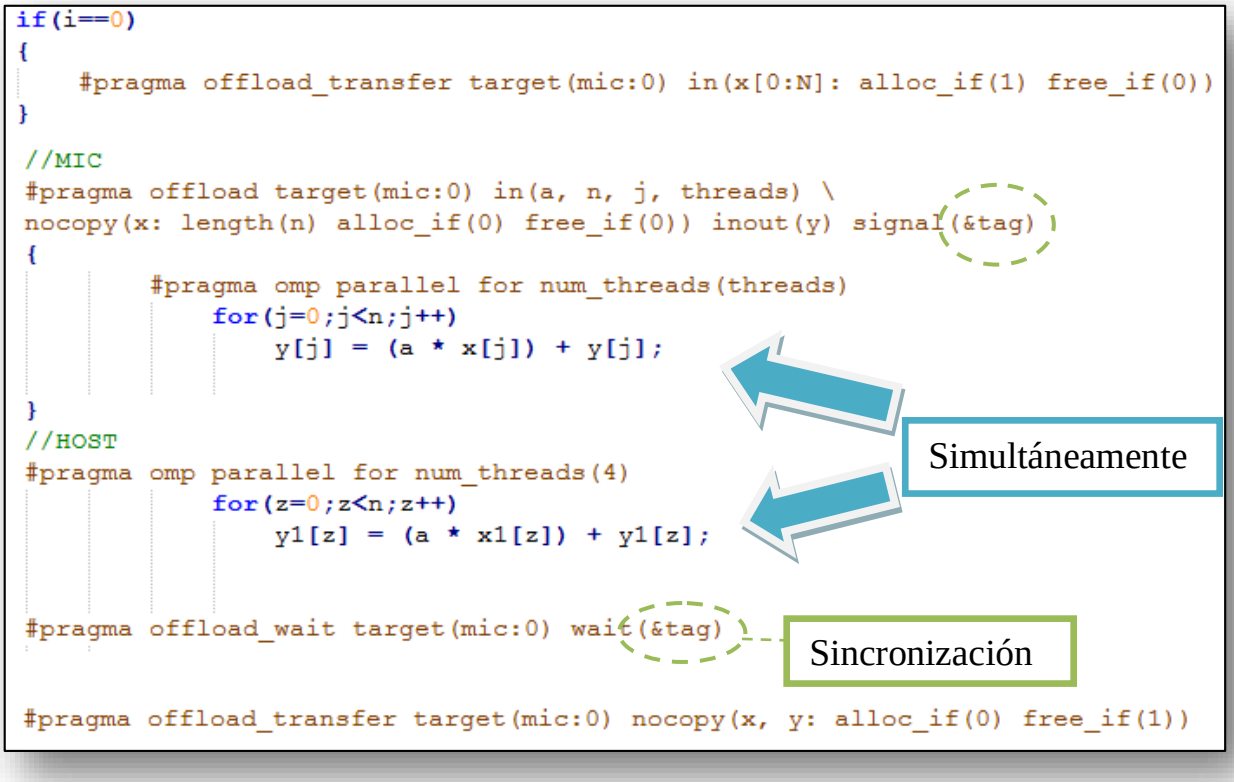


Ilustración 3.13 Offload asíncrono

3.4. Notación Array de Intel CILK

Intel Cilk es una extensión de los lenguajes C/C++ para el soporte de paralelismo de datos y tareas. Su nombre viene de *silk*, seda, y el lenguaje C en el que está basado y al que extiende para expresar bucles paralelos y el modelo de dividir y unir, o *fork-join*.

El principio detrás del diseño de Cilk es que el programador debería responsabilizarse del uso del paralelismo, identificando elementos que puedan ser ejecutados en paralelo de forma segura. Se ha trabajado dicho lenguaje para comprobar la potencia de este modelo y su fácil legibilidad en el código.

La parte de Intel Cilk que se detalla es la notación de array, que es utilizada en los programas utilizados.

- Ayuda al compilador a **vectorizar el código.**
- Se puede utilizar para **arrays estáticos o dinámicos**.
  - En los dinámicos es obligatorio especificar el inicio y la longitud, en cambio si se han declarado estáticamente estos valores se pueden omitir.
- También puede utilizarse **para condicionales** 'if' o 'switch'.

La sintaxis básica de su uso es la siguiente: `array [ inicio : longitud : paso ]`

- **Ejemplos básicos:**

```
A[:] = 5;
```

```
for (i = 0; i < 10; i++)
    A[i] = 5;
```

```
A[0:5:2] = 5;
A[1:5:2] = 4;
```

```
for (i = 0; i < 10; i += 2)
    A[i] = 5;
for (i = 1; i < 10; i += 2
    A[i] = 4;
```

```
C[:][:] = 12;
```

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        C[i][j] = 12;
```

- **Ejemplo saxpy:**

```
y[0:N:1] += a * x[0:N:1];
```

```
for (i = 0; i < N; i++)
    y[i] += a * x[i];
```

- **Ejemplo transferencia de la matriz de datos BD bidimensional:**

```
in(bd[0:nbd][0:K]: ALLOC)
```

3.5. OpenMP

OpenMP se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno. Es una interfaz de programación de aplicaciones (API, *application programming interface*) para la programación multiproceso de memoria compartida en múltiples plataformas.

Entre las distintas opciones existentes, hemos utilizado las que nos permiten un buen ajuste del entorno de ejecución en el Xeon Phi.

3.5.1 Affinity

Es posible asociar threads con unidades de procesamiento físicas utilizando la variable de entorno KMP_AFFINITY. En las ilustraciones se muestran las opciones de asignación, y utilizamos el programa multiplicación de matrices para comprobar la ejecución [configuración de muestra → matrices 4096x4096, 31 iteraciones, 112 threads]:

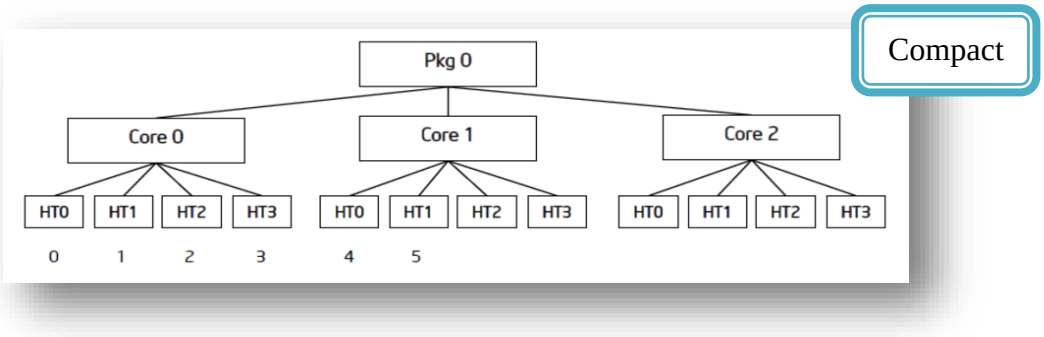


Ilustración 3.14 Afinidad compact

Podemos ver que compact asocia todos los threads a los primeros cores hasta terminar, lo que resulta en un desaprovechamiento de los recursos, y saturación de los cores asignados.

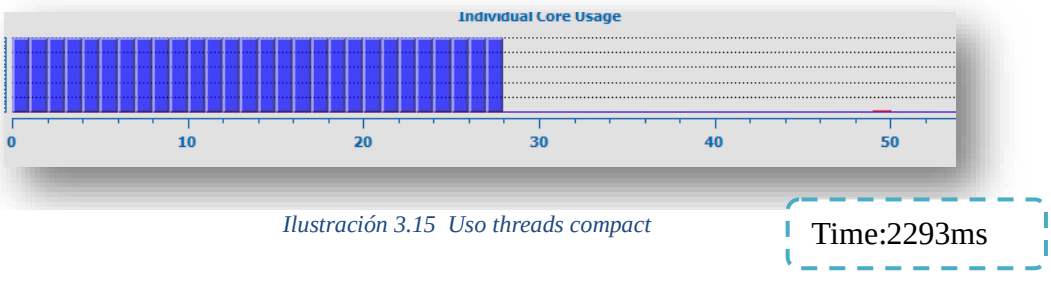


Ilustración 3.15 Uso threads compact

Time:2293ms

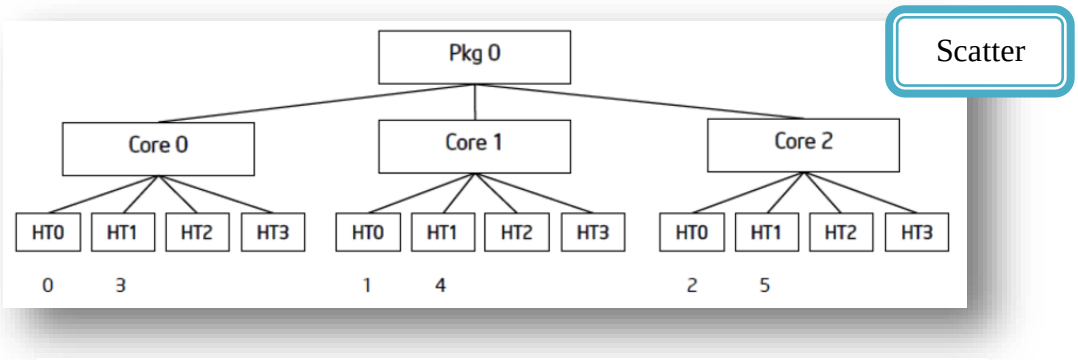


Ilustración 3.16 Afinidad scatter

La opción *scatter* va asociando progresivamente cada thread a un nuevo core para tratar de distribuir el trabajo entre todos los cores disponibles, y es la opción por defecto. Resulta en aproximadamente 1'6 veces más rápido que compact.

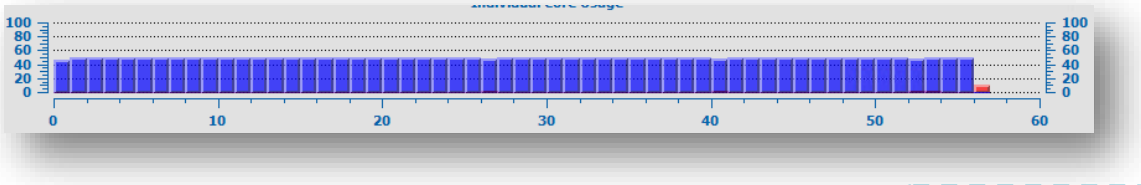


Ilustración 3.17 Uso threads scatter

Time:1451ms

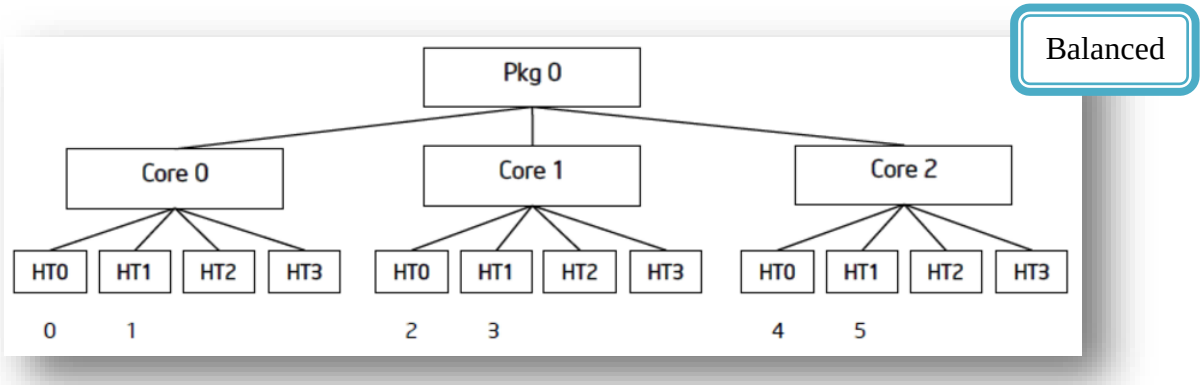


Ilustración 3.18 Afinidad balanced

La opción que hemos utilizado, *balanced*, que en nuestro caso se comporta de igual forma que scatter, obtiene igual resultado y rendimiento de los cores.

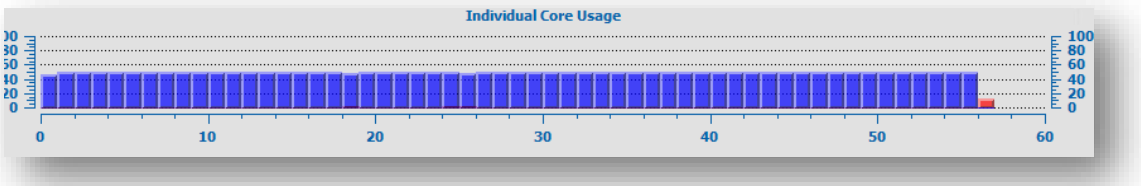


Ilustración 3.19 Uso threads balanced

Time:1447ms

3.5.2 Scheduling

Define como es la asignación de iteraciones a los threads → `schedule(tipo, chunk)`

- Static: se reparten equitativamente, chunk para tantas contiguas al mismo thread.
- Dynamic: igual que static, pero al acabar un thread con su carga ejecuta una de las pendientes.
- Guided: la cantidad de iteraciones decrece con cada nueva asignación hasta un mínimo marcado por chunk.

```
void saxpy( int n, float alpha, const float *x, float *y)
{
    int i;

       #pragma omp parallel for schedule(static) num_threads(NTHR)

              for (i=0;i<n;i++)
                  y[i] = (alpha * x[i]) + y[i];
}
```

Ilustración 3.20 Scheduling static

En nuestros programas, disponiendo siempre de una carga balanceada de las iteraciones, **utilizamos el scheduling estático**, establecido por defecto, y cada thread ejecutará así el mismo número de iteraciones.

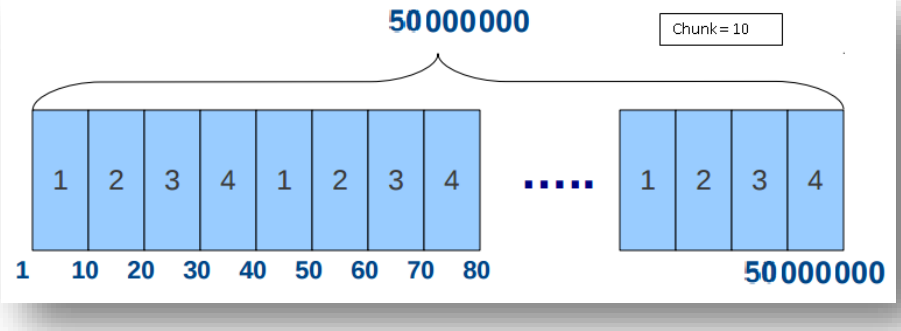


Ilustración 3.21 Distribución de iteraciones

Un alto chunk es recomendable para favorecer la **localidad de datos en la caché**.

3.5.3 Generales

También hemos utilizados las opciones más típicas de OpenMP, que vemos en la ilustración 3.22, como son:

- #pragma omp parallel (num threads): para establecer una región de código que ejecutarán concurrentemente el número de threads indicado.

- #pragma omp for: es similar a la anterior salvo que está optimizada para bucles for.

- Cláusula shared: los datos en la región paralela son compartidos, visibles y accesibles entre los threads.

- Cláusula private: los datos son privados para cada thread, es decir, tendrá una copia local que usará como variable temporal. Los contadores de iteraciones se consideran privados.

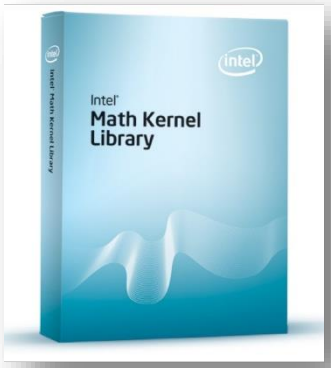
```
    int i, j, k;
#pragma omp parallel shared(a, b, c, n) private(i,j,k) num_threads(threads)
    {
    #pragma omp for schedule(static)
                for (i = 0; i < n; i++)
                    for (j = 0; j < n; j++)
                        for (k = 0; k < n; k++)
                            c[i][j] += (a[i][k]) * (b[k][j]);
    }
```

Ilustración 3.22 Cláusulas generales

3.6. Intel MKL

3.6.1 Descripción

Intel Math Kernel Library es una **librería de funciones matemáticas**, para compiladores C/C++/Fortran. Estas rutinas están muy optimizadas para obtener el más alto rendimiento (a mano), y al mismo tiempo reduce el tiempo de desarrollo.



MKL incluye múltiples funciones matemáticas (por ejemplo de álgebra lineal) y utilizan técnicas multihilo y vectorización. Únicamente se debe sustituir la función deseada por una de MKL y enlazar su librería para obtener el mejor rendimiento.

3.6.2.- Modelos de uso

- Nativo: Compilar para la arquitectura MIC enlazando esta librería como cualquier otra. El coprocesador usa por defecto todos los threads disponibles (4*num_cores).

```
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
N, N, N, alpha, A, N, B, N, beta, C, N);
```

- Offload Asistido por Compilador: También conocida como CAO (compiler assisted offload) ,consiste en añadir una sentencia con el mic objetivo y el movimiento de los datos. En este modelo se utilizará un número de threads (4* (num_cores - 1)).

```
#pragma offload target(mic:0) in(transa, transb, alpha, beta) \
nocopy(A, B: alloc_if(0) free_if(0)) inout(C)
    {
        cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
        N, N, N, alpha, A, N, B, N, beta, C, N);
    }
```

Ilustración 3.23 Offload mkl

- Automatic Offload: También llamada AO, es una herramienta de MKL que proporciona **mejoras de rendimiento sin cambios en el código**, a diferencia de *Compiler Assisted Offload*.

Cuando ejecutamos una función en el Host CPU, Intel MKL en modo *AO* puede descargar parte del cómputo a uno o varios coprocesadores **sin tener que manejarlo explícitamente**. Por defecto, Intel MKL **decide el mejor reparto** de trabajo, aunque es posible establecer una configuración propia.

Ilustración 3.24 Automatic offload mkl

Vemos en la ilustración 3.24 que usando un programa mkl ya creado, sólo deberemos activar la variable de entorno **MKL_MIC_ENABLE a 1**, para que esta herramienta esté preparada.

El tamaño de la matriz en nuestro caso SGEMM es crítico y no se activará por debajo de un tamaño de fila/columna de 2048 (el overhead de la transferencia de datos obstaculiza el rendimiento computacional).

Hay que señalar que no todas las funciones MKL soportan esta funcionalidad, aunque las irán incorporando en futuras versiones. La ilustración 3.25 muestra las compatibles de multiplicación de matrices.

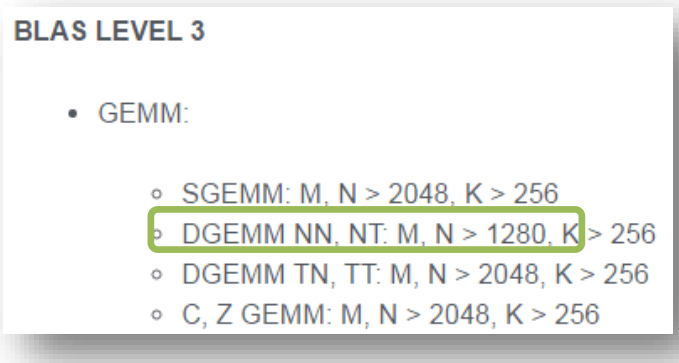


Ilustración 3.25 Automatic offload, funciones soportadas

Estas son las principales variables de entorno que se utilizan para manejar el comportamiento de Automatic Offload:

Activar o desactivar Automatic Offload.	MKL_MIC_ENABLE=[0/1]
Establecer una **lista de dispositivos.**	OFFLOAD_DEVICES=<0,1,2,…>
Activar **nivel de reporte** offload.	OFFLOAD_REPORT=[0/1/2]
Repartir carga de trabajo a realizar entre los dispositivos.	MKL_{ HOST,MIC(_#) }_WORKDIVISION=[0.0-1.0]
Establecer el **máximo de memoria** usada por dispositivo.	MKL_MIC(_#)_MAX_MEMORY=[# K/M/G/T]
Especifica el **límite máximo de threads.**	MIC(_#)_OMP_NUM_THREADS=#

Tabla 3.5 Variables de entorno Automatic Offload

Podemos ver un ejemplo de como realizar el control de este control de la carga de trabajo en la ilustración 3.26, y ver como el programa mismo se ocupa de transferir y computar sólo lo correspondiente.

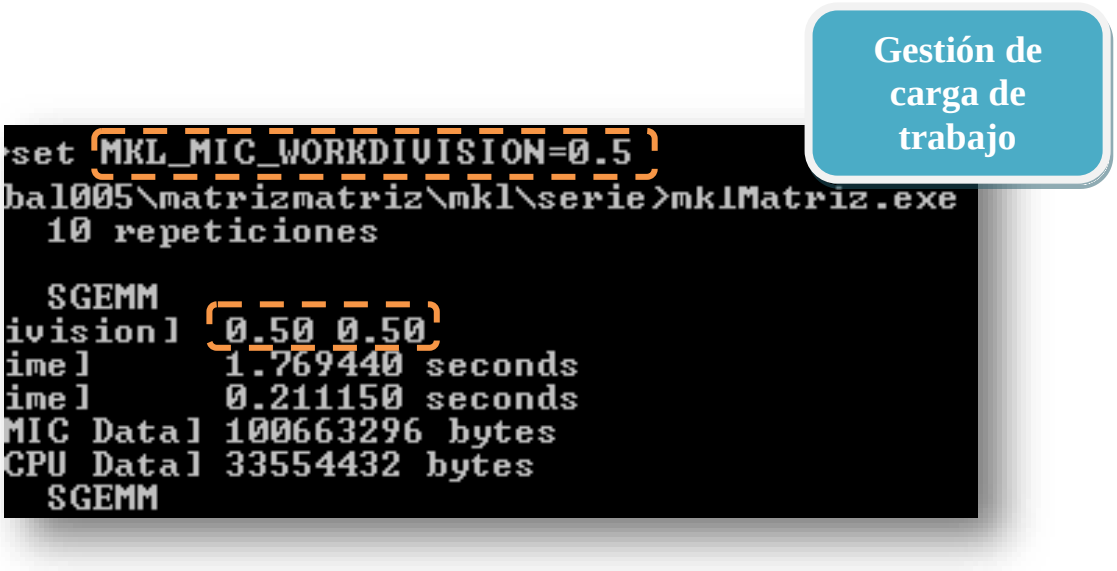


Ilustración 3.26 Automatic offload división de carga

4

Categorías de aplicaciones

En este capítulo, en primer lugar explicamos cómo **alcanzar el límite** de operaciones en coma flotante por segundo, *FLOPS*, al que se indica en la documentación oficial que el Xeon Phi es capaz de llegar.

Después principalmente hemos trabajado con aplicaciones que se sitúan en dos categorías: los programas limitados por memoria (**memory bound**) y los limitados por capacidad de proceso (**cpu bound**). Terminamos desarrollando un programa que se encuentra **entre estos dos modelos.**

4.1. Configuración del máximo teórico

Para cualquier plataforma de cálculo, las especificaciones hardware definen una capacidad máxima para el cálculo de FLOPs, que indican el número de operaciones de coma flotante que dicha plataforma puede realizar por operando recibido de memoria.
El rendimiento máximo teórico en el coprocesador Intel **Xeon Phi**, en Gflops/seg:

16[#sp-SIMD] x 2[fma] x 1.100[Ghz] x 56[cores] = 1971.2 Gflops =* ***1.97 Tflops***

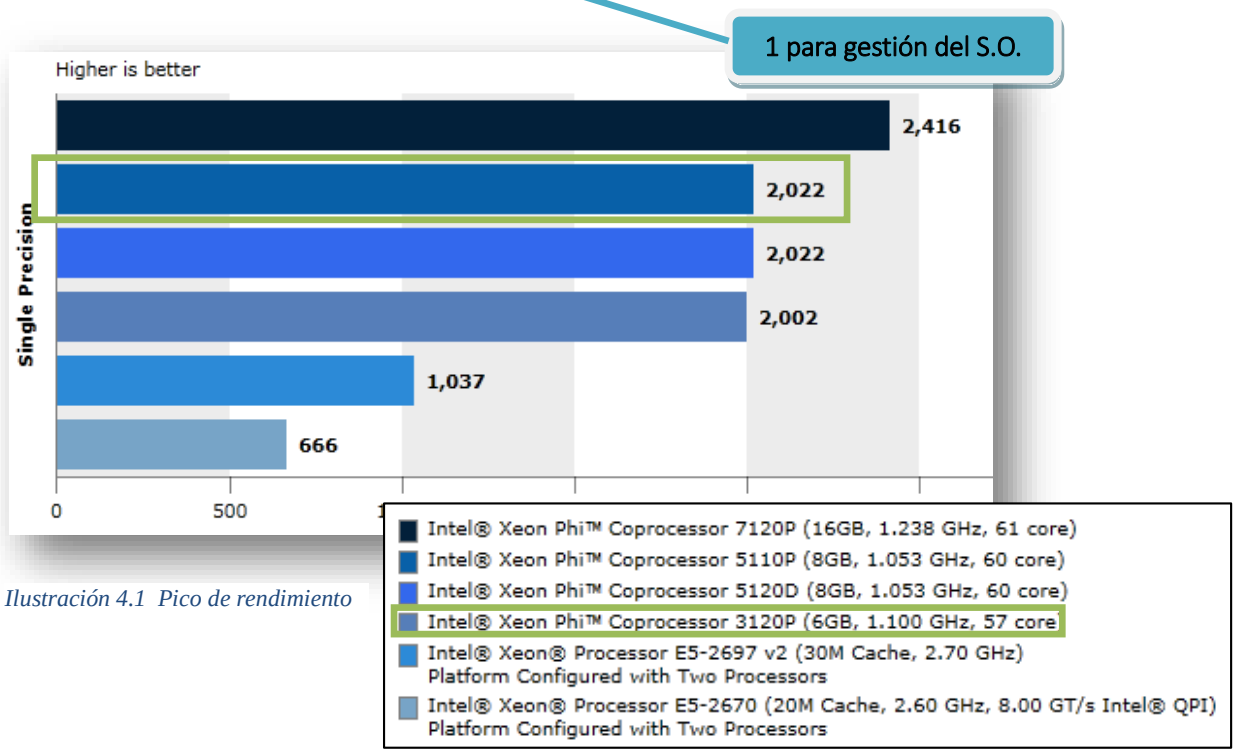


Ilustración 4.1 Pico de rendimiento

Para conseguir llegar a ese límite existen una serie de condiciones ideales que vamos a trabajar con una propuesta que ejecuta la función saxpy como la de la ilustración 4.2, donde:

- Es necesario que exista una cantidad de cálculo en coma flotante realmente grande, donde todos los cores trabajen a máximo rendimiento. Maximizamos las iteraciones y minimizamos el tamaño de datos para no generar un cuello de botella en la transferencia.

- Aprovechar la vectorización al máximo es esencial, para lo que utilizamos la notación CILK, que indica al compilador el comienzo de bucle y el número de elementos a tratar (registros vectoriales 512 bits), con paso 1.

- Alinear las estructuras de datos para evitar perder rendimiento en la carga de los mismos (con la sentencia *'#pragma vector aligned'* hacemos que el compilador confíe en la correcta alineación).

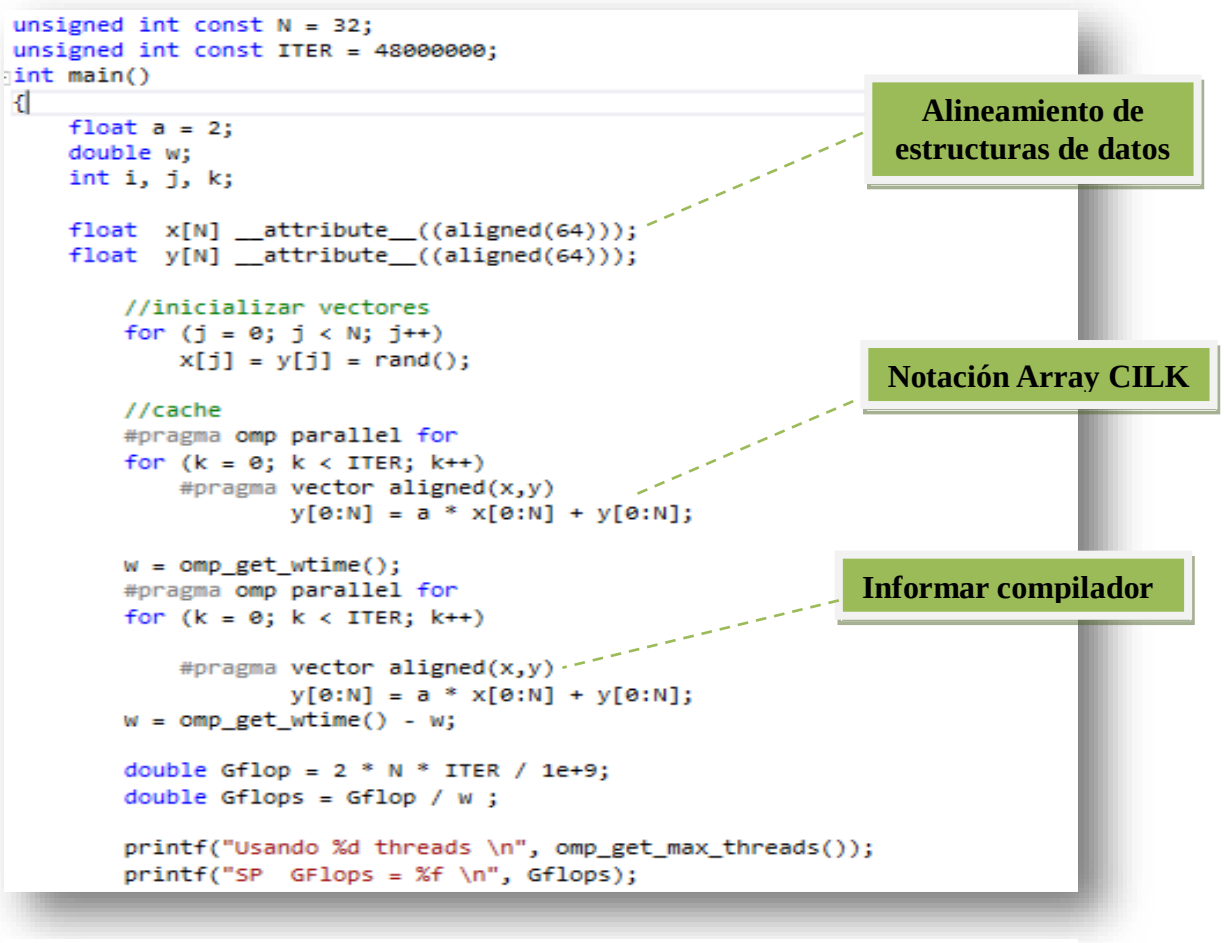


Ilustración 4.2 Programa máximo teórico

Para obtener como resultado las operaciones de coma flotante por segundo:

- Multiplicando [operaciones por iteración (add y mul) * longitud del vector * número de iteraciones].
- Dividir entre mil millones para gigaflops totales.
- Dividir entre el tiempo transcurrido.

Obtenemos finalmente el rendimiento de operaciones de coma flotante de simple precisión por segundo mostrado en la ilustración 4.3:

Ilustración 4.3 Ejecución alcance pico de rendimiento

La mayoría de las instrucciones vectoriales tiene 3-4 ciclos de latencia, y por ello lanzamos al menos **2 threads por núcleo** y **ocultar así esta latencia** de instrucciones.

Consideramos un tamaño de vector de 32 floats, para que por iteración cada thread disponga de ellos en sus registros vectoriales ZMM de 512bits de ancho. La afinidad de los threads debe ser 'balanced' para que se distribuyan entre los cores optimizando la ejecución.

4.2. Memory Bound

Las aplicaciones situadas en esta categoría es debido a que, para completar su objetivo, realizan un acceso intensivo del ancho de banda de memoria para acceder a los datos a través de un bus de , en nuestro caso 8 GB/s. Se convierte este **ancho de banda en un factor limitador** y no permite al procesador utilizar todo su potencial computacional.

Utilizaremos un programa representativo para medir el impacto que tienen las optimizaciones del compilador de Intel y el uso del Xeon Phi.

El programa elegido para esta categoría es:

- Axpy de simple precisión – contenido en la librería BLAS (Basic Linear Algebra Subprograms) de nivel 1.

$$y \leftarrow \alpha x + y$$

En la ilustración 4.4 aparece una implementación en C de la función SAXPY utilizando float.

```
__declspec(align(64)) float  x[N];
__declspec(align(64)) float  y[N];
void saxpy( int n, float alpha, const float *x, float *y)
{
    int i;
        for (i=0;i<N;i++)
            y[i] = (alpha * x[i]) + y[i];
}
int main()
{
    float a = 2; double w2;
    double tmedia, tacum, tcarga = 0.0;
    double tmin = DBL_MAX; double tmax = 0.0;
    double tejec = 0.0; int i, k, c;
    for (i = 0; i < ITER; i++) {

        for (k = 0; k < N; k++) {
            x[k] = k;
            y[k] = k + 1;
        }
        w2 = omp_get_wtime();
            saxpy(N, a, x, y);
        tejec = (omp_get_wtime() - w2) * 1000;
```

SAXPY

Ilustración 4.4 Implementación en C de saxpy

La descripción del flujo del programa sería la siguiente:

- Primero son declarados las estructuras de datos estáticamente.
- En la función principal, un bucle realizar un número determinado de iteraciones y en cada una:
    - Inicializa los valores de los vectores X e Y.
    - Mide el tiempo transcurrido en completar la función saxpy llamada (multiplicamos por 1000 para obtener milisegundos).
    - Para el cálculo del tiempo medio, hacemos un sumatorio del tiempo de todas las iteraciones y después la división entre las mismas (la primera iteración es descartada, así como las dos que se hayan completado en el mayor y menor tiempo).

4.2.1.- Compilación y Reportes

Utilizando lo trabajado en el apartado de optimizaciones, mostramos el uso de las mismas con **ficheros de compilación** .bat (batch file) de Windows.

Añadimos **algunos reportes** que ayuden a comprobar si en efecto se aplican técnicas de paralelismo, vectorización, etc. Para generarlos utilizamos el flag /Qopt-report-phase = [vec: para comprobar vectorización] [openmp: para comprobar paralelismo].

Si ninguna optimización general (O0, O1, O2, O3) es especificada, se compila O2 por defecto.

- **Optimización O0**

```
icl /Od -openmp /Qopt-report-phase:vec,openmp saxpy_o0.c
saxpy_o0.exe >> results.txt
move saxpy_o0.obj "C:\Users\ehu\Documents\jaldazabal005\sa
move saxpy_o0.exe "C:\Users\ehu\Documents\jaldazabal005\sa
```

Genera reporte vacío: sin vectorización, sin paralelización.

- **Optimización O1**

```
icl /O1 -openmp /Qopt-report-phase:vec,openmp saxpy_o1.c
saxpy_o1.exe >> results.txt
```

Genera reporte vacío: sin vectorización, sin paralelización.

- **Optimización O2_no-vec**

```
icl /O2 /Qvec- -openmp /Qopt-report-phase:vec,openmp saxpy_o2_novec.c
saxpy_o2_novec.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\saxpy\entero
\saxpy_o2_novec.c(22,9) inlined into C:\Users\ehu\Documents
\jaldazabal005\saxpy\entero\saxpy_o2_novec.c(45,4)
   remark #15540: loop was not vectorized: auto-vectorization is
disabled with /Qvec- flag
LOOP END
```

- **Optimización O2**

```
icl /O2 -openmp /Qopt-report-phase:vec,openmp saxpy_o2.c
saxpy_o2.exe >> results.txt
```

```
\Users\ehu\Documents\jaldazabal005\sa
)
v1>
: vectorization support: reference y has aligned
ers\ehu\Documents\jaldazabal005\saxpy\entero
3) ]
: vectorization support: reference x has aligned
ers\ehu\Documents\jaldazabal005\saxpy\entero
3) ]
: vectorization support: reference y has aligned
ers\ehu\Documents\jaldazabal005\saxpy\entero
3) ]
: LOOP WAS VECTORIZED
: entire loop may be executed in remainder
: unmasked aligned unit stride loads: 2
: unmasked aligned unit stride stores: 1
: --- begin vector loop cost summary ---
: scalar loop cost: 13
: vector loop cost: 2.000
: estimated potential speedup: 6.490
: lightweight vector operations: 7
```

Estructuras de datos alineadas a 64 bytes

- **Optimización O3**

```
icl /O3 -openmp /Qopt-report-phase:vec,openmp saxpy_o3.c
saxpy_o3.exe >> results.txt
```

```
   LOOP BEGIN at C:\Users\ehu\Documents\jalda
\entero\saxpy_o3.c(26,9) inlined into C:\User
\jaldazabal005\saxpy\entero\saxpy_o3.c(61,6)
      remark #15300: LOOP WAS VECTORIZED
   LOOP END
LOOP END
```

- **Optimización xHost**

```
icl /QxHOST -openmp /Qopt-report-phase:vec,openmp saxpy_xhost.c
saxpy_xhost.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\saxpy\
\saxpy_xhost.c(26,9) inlined into C:\Users\ehu\Documents
\jaldazabal005\saxpy\entero\saxpy_xhost.c(61,6)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- **Optimización avx2**

```
icl /QxCORE-AVX2 -openmp /Qopt-report-phase:vec,openmp saxpy_avx2.c
saxpy_avx2.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\saxpy
\saxpy_avx2.c(26,9) inlined into C:\Users\ehu\Documents
\jaldazabal005\saxpy\entero\saxpy_avx2.c(61,6)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- **Optimización openmp**

```
icl /Qopenmp /Qopt-report-phase:vec,openmp saxpy_omp.c
saxpy_omp.exe >> results.txt
```

```
    Report from: OpenMP optimizations [openmp]

OpenMP Construct at C:\Users\ehu\Documents\jaldazabal005\s
\entero\saxpy_omp.c(26,5) inlined into C:\Users\ehu\Docume
\jaldazabal005\saxpy\entero\saxpy_omp.c(57,6)
   remark #16200: OpenMP DEFINED LOOP WAS PARALLELIZED

    Report from: Vector optimizations [vec]

LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\saxpy\e
\saxpy_omp.c(49,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- **Optimización mkl:sequential**

```
icl /Qopenmp /Qmkl:sequential /Qopt-report-phase:vec,openmp saxpy_mkl_seq.c
saxpy_mkl_seq.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\s
\saxpy_mkl_seq.c(41,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- **Optimización mkl:parallel**

```
icl /Qopenmp /Qmkl:parallel /Qopt-report-phase:vec,openmp saxpy_mkl_par.c
saxpy_mkl_par.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\saxpy\entero
\saxpy_mkl_par.c(41,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- **Optimización mkl: offload**

```
icl /Qopenmp /Qoffload-attribute-target=mic /Qmkl /Qopt-report-phase:vec,offload saxpy_mkl_off.c
set OFFLOAD_REPORT=2
saxpy_mkl_off.exe >> results.txt
```


```
OpenMP Construct at off.c(47,5)
   remark #16200: OpenMP DEFINED LOOP WAS PARALLELIZED

    Report from: Offload optimizations [offload]

OFFLOAD:main(45,3):  Outlined offload region
 Data received by target from host
       a, scalar size 4 bytes
       n, scalar size 4 bytes
       j, scalar size 4 bytes
       threads, scalar size 4 bytes
       y_V$a, array size 200000000 bytes
 Data sent from target to host
       y_V$a, array size 200000000 bytes

LOOP BEGIN at off.c(48,6)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- **Ejecución native**

```
icl /Qmic -openmp saxpy_nativo.c
```

La compilación para plataforma nativa no genera reporte.

- **Ejecución offload**

```
icl /Qopenmp /Qoffload-attribute-target=mic /Qopt-report-phase:vec,openmp,offload off.c
```

```
   remark #15300: LOOP WAS VECTORIZED
LOOP END

    Report from: OpenMP optimizations [openmp]

OpenMP Construct at off.c(47,5)
   remark #16200: OpenMP DEFINED LOOP WAS PARALLELIZED

    Report from: Offload optimizations [offload]

OFFLOAD:main(45,3):  Outlined offload region
 Data received by target from host
       a, scalar size 4 bytes
       n, scalar size 4 bytes
       j, scalar size 4 bytes
       threads, scalar size 4 bytes
       y_V$a, array size 200000000 bytes
```

- **Opción guided**: no genera ejecutable, sino un reporte de ayuda con recomendaciones en paralelización/vectorización. Es necesario el flag /Qparallel.

```
icl /Qopenmp /Qguide /Qparallel saxpy_omp.c
```

```
saxpy_omp.c
GAP REPORT LOG OPENED ON Mon Jun 13 17:42:40 2016

C:\Users\ehu\Documents\jaldazabal005\saxpy\entero\saxpy_omp.c(28): remark #30525
: (PAR) Insert a "#pragma loop count min(256)" statement right before the loop a
t line 28 to parallelize the loop. [VERIFY] make sure that the loop has a minimu
m of 256 iterations.
Number of advice-messages emitted for this compilation session: 1.
END OF GAP REPORT LOG
```

Sugerencias

Establecer número de iteraciones mínimo

Ilustración 4.5 Recomendación de opción guide

4.2.2.- Resultados y conclusiones

		Tamaño de vector					
		5.000.000		50.000.000			
		T_medio	SpeedUp	T_medio	SpeedUp	VEC	PAR
HOST	O0	14,0	0,24	139,0	0,22	✗	✗
	O1	4,0	0,85	35,3	0,85	✗	✗
	O2_no-vec	3,9	0,87	35,7	0,84	✗	✗
	O2	3,4	1,00	29,9	1,00	✓	✗
	xHost	3,4	1,00	30,9	0,97	✓	✗
	Avx2	3,4	1,00	31,0	0,96	✓	✗
	O3	3,3	1,03	29,9	1,00	✓	✗
	Omp	1,7	2,00	17,5	1,71	✓	✓
	mkl:seq	3,4	1,00	30,3	0,99	✓	✗
	mkl:par	1,8	1,89	17,5	1,71	✓	✓
MIC	Native	0,6	5,67	6,2	4,82	✓	✓
	mkl:offload	66	0,05	368	0,08	✓	✓
	Offload	6,7	0,51	64,7	0,46	✓	✓

Tabla 4.1 Resultados saxpy

0,00 < SpeedUp < 1,00
1,00 < SpeedUp < 2,00
2,00 < SpeedUp < 10,00

HOST:

- La optimización **O0** se utiliza para depuración y verificación de programas y **O1** en casos de sensibilidad al tamaño del código.
- **O2** será la referencia, que activa optimizaciones comunes y vectorización del compilador. Deshabilitando la vectorización se aprecia una bajada de rendimiento, mientras las optimizaciones de **O3** no tienen efecto.
- Las optimizaciones **xhost** y **avx2** reflejan cómo efectivamente utilizan la misma arquitectura de instrucciones.
- Vemos que utilizar la librería **mkl** no supone diferencia en **secuencial**, mientras que la opción mkl en **paralelo** nos ofrece un speedup de aproximadamente el doble, igual que el uso de **OpenMP** en el host lanzando 4 threads.

MIC:

- La ejecución del programa en **nativo** es aquí la más eficaz puesto que consigue ir entre 5 y 6 veces más rápido, utilizando los 57 cores del coprocesador.
- Vemos que el uso de **offload** en este caso no es nada rentable debido al coste temporal de transferir los datos al Xeon Phi.

Este último caso es el más ilustrativo para demostrar el tipo de programa limitado por memoria (**memory bound**), para el programa saxpy:

La transferencia de datos del host al coprocesador es un bus PCIe v2 a **8 GB/s**. Activando la monitorización offload con OFFLOAD_REPORT=2, observamos el tiempo de transferencia y de cómputo en la ilustración 4.6, concluyendo que el acceso a memoria se convierte en el cuello de botella:

Y[50.000.000] → 200.000.000B = 200MB

PCIexpress: 8GB/s

8192MB	---	1s
200MB	---	x

x = 200/8192 = 0,0244s = 24,4ms

Input(y) = 24,4 + Output(y) = 24,4 → 48,8ms

```
                Tag 91
[CPU Time]       0.006607(seconds)
[CPU->MIC Data]  20000016 (bytes)
[MIC Time]       0.000487(seconds)
[MIC->CPU Data]  20000000 (bytes)
```

Ilustración 4.6 Reporte offload saxpy

[Reporte] CPUtime – MICtime = 66ms – 5ms = **61ms**

* Vemos que el 92,5% de la ejecución lo ocupa la transferencia de datos y el 7,5% restante el cómputo.

4.3. CPU Bound

En esta categoría encontramos programas donde el tiempo requerido para completar su objetivo viene determinado principalmente por la velocidad de CPU, puesto que para la gran mayoría del tiempo realizando cálculos.

Utilizaremos un programa representativo para medir el impacto que tienen en el rendimiento, por un lado, las distintas optimizaciones de ofrece el compilador de Intel, y por otro, la utilización del coprocesador Xeon Phi para realizar el cómputo requerido.
El programa elegido para esta categoría es:

- Multiplicación de matrices – contenido en la librería BLAS (Basic Linear Algebra Subprograms) de nivel 3.

$$C \Leftarrow A * B + C$$

Hemos elegido la opción más genérica de multiplicación de matrices, por delante de casos especiales como el tipo simétrico o el tipo triangular.

La función matemática de multiplicación de matrices, utilizará elementos floats de 32 bits en cada una de ellas. La implementación básica utilizada en lenguage C es la mostrada en la ilustración 4.7:

La descripción del flujo programa sería la siguiente:

- Primero son declarados las estructuras de datos estáticamente, A, B y C.
- En la función principal, un bucle realizar un número determinado de iteraciones y en cada una:
    - Inicializa los valores de los arrays A, B y C.
    - Mide el tiempo transcurrido en completar la función saxpy llamada (multiplicamos por 1000 para obtener milisegundos).
- Para el cálculo del tiempo medio, realizamos exactamente el mismo procedimiento que para la función saxpy.

```
__declspec(align(64)) float  A[N][N];
__declspec(align(64)) float  B[N][N];
__declspec(align(64)) float  C[N][N];

void mm( int n, float Matriz_A[N][N], float Matriz_B[N][N], float Matriz_C[N][N])
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                Matriz_C[i][j] += Matriz_A[i][k] * Matriz_B[k][j];
}

int main()
{
    int i, j, k;
    double w2;
    double tmedia, tacum, tcarga = 0.0;
    double tmin = DBL_MAX;
    double tmax = 0.0;
    double tejec = 0.0;

        for (i = 0; i < ITER; i++) {
            /*** Inicializar matrices ***/

            for (k= 0; k < N; k++)
                for (j = 0; j < N; j++){
                    A[k][j] = k + j;
                    B[k][j] = k - j;
                    C[k][j] = 0;
                }

            w2 = omp_get_wtime();

                mm(N, A, B, C);

            tejec = (omp_get_wtime() - w2) * 1000;
```

MULTIPLICACIÓN DE MATRICES

Ilustración 4.7 Implementación en C de multiplicación de matrices

4.3.1.- Compilación y Reportes

Mostramos el uso de las optimizaciones utilizadas con **ficheros de compilación** .bat (batch file) de Windows.

Añadimos **algunos reportes** para comprobar cuales de ellas aplican técnicas de paralelismo, vectorización, etc. Para generarlos utilizamos el flag /Qopt-report-phase= [vec: para comprobar vectorización] [openmp: para comprobar paralelismo].

Si ninguna optimización general (O0, O1, O2, O3) es especificada, se compila O2 por defecto.

- **Optimización O0:**

```
icl /Od -openmp /Qopt-report-phase:vec,openmp mxm_o0.c
mxm_o0.exe >> results.txt
```

Genera reporte vacío: sin vectorización, sin paralelización.

- Optimización O1:

```
icl /O1 -openmp /Qopt-report-phase:vec,openmp mxm_o1.c
mxm_o1.exe >> results.txt
```

Genera reporte vacío: sin vectorización, sin paralelización.

- Optimización O2_no-vec:

```
icl /O2 /Qvec- -openmp /Qopt-report-phase:vec,openmp mxm_o2_no_vec.c
mxm_o2_no_vec.exe >> results.txt
```

```
      LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_o2_no_vec.c(19,3) inlined into
\ehu\Documents\jaldazabal005\matrizmatriz\entero\mxm_o2
_no_vec.c(47,5)
         remark #15540: loop was not vectorized: auto-
vectorization is disabled with /Qvec- flag
```

- Optimización O2:

```
icl /O2 -openmp /Qopt-report-phase:vec,openmp mxm_o2.c
mxm_o2.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\matrizmatriz
\entero\mxm_o2.c(18,2) inlined into C:\Users\ehu\Documents
\jaldazabal005\matrizmatriz\entero\mxm_o2.c(47,5)
   remark #15542: loop was not vectorized: inner loop was already
vectorized

   LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_o2.c(20,4) inlined into C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_o2.c(47,5)
      remark #15542: loop was not vectorized: inner loop was
already vectorized

      LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_o2.c(19,3) inlined into C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_o2.c(47,5)
         remark #15301: PERMUTED LOOP WAS VECTORIZED
      LOOP END
   LOOP END
LOOP END
```

i j k → i k j

El compilador detecta automáticamente que en la instrucción de la multiplicación de matrices, hay sucesivos **accesos a memoria** para leer elementos de B con **stride n**. Utiliza por tanto una técnica de **intercambio de los bucles j y k** para que esta lectura tenga **stride 1** y permitir así la vectorización.

- Optimización O3:

```
icl /O3 -openmp /Qopt-report-phase:vec,openmp mxm_o3.c
mxm_o3.exe >> results.txt
```

```
                  LOOP BEGIN at C:\Users\ehu\Documents
\jaldazabal005\matrizmatriz\entero\mxm_o3.c(19,3) inlined into C:
\Users\ehu\Documents\jaldazabal005\matrizmatriz\entero
\mxm_o3.c(47,5)
                     remark #15301: PERMUTED LOOP WAS VECTORIZED
                  LOOP END
```

- Optimización xHost:

```
icl /QxHOST -openmp /Qopt-report-phase:vec,openmp mxm_xhost.c
mxm_xhost.exe >> results.txt
```

```
      LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_xhost.c(19,3) inlined into C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_xhost.c(47,5)
         remark #15301: PERMUTED LOOP WAS VECTORIZED
      LOOP END
```

- Optimización AVX2:

```
icl /QxCORE-AVX2 -openmp /Qopt-report-phase:vec,openmp mxm_avx2.c
mxm_avx2.exe >> results.txt
```

```
      LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_avx2.c(19,3) inlined into C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_avx2.c(47,5)
         remark #15301: PERMUTED LOOP WAS VECTORIZED
```

- Optimización OpenMP:

```
icl /Qopenmp /Qopt-report-phase:vec,openmp mxm_omp.c
mxm_omp.exe >> results.txt
```

```
      LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_omp.c(25,6) inlined into C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_omp.c(60,4)
         remark #15301: PERMUTED LOOP WAS VECTORIZED
      LOOP END

    Report from: OpenMP optimizations [openmp]

OpenMP Construct at C:\Users\ehu\Documents\jaldazabal005
\matrizmatriz\entero\mxm_omp.c(20,1)
   remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED
```

- Optimización Matmul: se activa también con los flags /O3 + /Qparallel.

```
icl /Qopt-matmul -openmp /Qopt-report-phase:vec,openmp mxm_matmul.c
mxm_matmul.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\matrizmatriz
\entero\mxm_matmul.c(34,3)
   remark #15543: loop was not vectorized: loop with function
call not considered an optimization candidate.   [ C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_matmul.c(44,9) ]
```

Las llamadas a funciones externas no se marcan vectorizadas, aunque luego lo hagan, como matmul o mkl

- Optimización mkl:seq:

```
icl /Qopenmp /Qmkl:sequential /Qopt-report-phase:vec,openmp mxm_mkl_seq.c
mxm_mkl_seq.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\matrizmatriz
\entero\mxm_mkl_seq.c(32,3)
   remark #15543: loop was not vectorized: loop with function
call not considered an optimization candidate.   [ C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero
\mxm_mkl_seq.c(45,9) ]
```

- Optimización mkl:par:

```
icl /Qopenmp /Qmkl:parallel /Qopt-report-phase:vec,openmp mxm_mkl_par.c
mxm_mkl_par.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\matrizmatriz
\entero\mxm_mkl_par.c(35,3)
   remark #15543: loop was not vectorized: loop with function
call not considered an optimization candidate.   [ C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero
\mxm_mkl_par.c(46,9) ]
```

- Optimización mkl:off:

```
icl /Qopenmp /Qoffload-attribute-target=mic /Qmkl:parallel /Qopt-report-phase:vec,openmp mxm_mkl_off.c
mxm_mkl_off.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\matrizmatriz
\entero\mxm_mkl_off.c(35,3)
   remark #15543: loop was not vectorized. loop with function
call not considered an optimization candidate.    [ C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero
\mxm_mkl_off.c(45,5) ]
```

- Optimización offload:

```
icl /Qopenmp /Qoffload-attribute-target=mic /Qopt-report-phase:vec,openmp off.c
off.exe 112 >> results.txt
```

```
OpenMP Construct at off.c(48,18)
   remark #16200: OpenMP DEFINED LOOP WAS PARALLELIZED

    Report from: Vector optimizations [vec]

      LOOP BEGIN at off.c(50,19)
         remark #15301: PERMUTED LOOP WAS VECTORIZED
      LOOP END
```

- Optimización AO:

```
set MKL_MIC_ENABLE=1
icl /Qopenmp /Qmkl /Qopt-report-phase:vec,openmp mxm_AO.c
mxm_AO.exe >> results.txt
```

```
LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005\matrizmatriz
\entero\mxm_AO.c(35,3)
   remark #15543: loop was not vectorized: loop with function
call not considered an optimization candidate.    [ C:\Users\ehu
\Documents\jaldazabal005\matrizmatriz\entero\mxm_AO.c(48,9)
```

La ventaja de esta opción es no tener que cambiar el código, solo añadir una variable de entorno. La desventaja es que transfiere A y B en todas las iteraciones, sin ofrecer la flexibilidad de enviar una matriz y no dos.

```
[AO SGEMM CPU Time]       3.154813
[AO SGEMM MIC Time]       0.356776 seconds
[AO SGEMM CPU->MIC Data] 130023424 bytes
[AO SGEMM MIC->CPU Data] 62914560 bytes
```

2 matrices

- Optimización native:

```
icl /Qmic /Qopenmp nativeMatriz.c
```

La compilación para plataforma nativa no genera reporte.

- Opción guided: no genera ejecutable, sino un reporte de ayuda con recomendaciones en paralelización/vectorización. Es necesario el flag /Qparallel.

```
icl /Qopenmp /Qparallel /Qguide mxm_omp.c
```

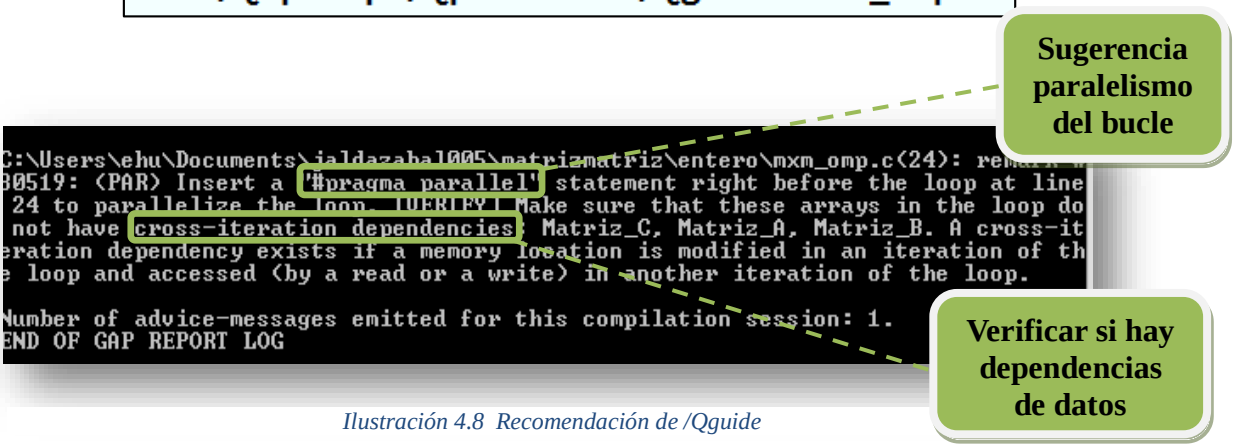


Ilustración 4.8 Recomendación de /Qguide

Las optimizaciones matmul y los distintos tipos de mkl no se muestran como vectorizadas/paralelizadas en el reporte, debido a que es una **llamada a una función externa**, ya optimizada en estas técnicas internamente.

4.3.2.- Resultados y conclusiones

HOST:

- En este caso, **O2** vuelve a ser **la referencia**, a la vez que **O0**, **O1**, **O2_no-vec** muestran un bajo rendimiento, debido principalmente a la falta de vectorización.

- Las optimizaciones de **AVX2**, **xHost** y **O3**, dejan un rendimiento similar a O2. Sin embargo, podemos añadir a O3 la opción parallel, que identifica la función como multiplicación de matrices y genera una llamada a una potente librería que en este caso muestra un speedup aproximado de 35. Esta llamada es equivalente a utilizar la opción **matmul** que es la que muestran los datos.

- Cuando ejecutamos el programa utilizando **OpenMP**, aprovechamos los 4 cores del host, y conseguimos aproximadamente el triple de speedup que la referencia.

- En el uso de las librerías matemáticas **mkl**, la función cblas_sgemm muestra en **secuencial** un speedup de 12 debido a que realiza optimizaciones y comprobaciones en los parámetros (transposición, alpha/beta == 0, etc) para elegir la versión más adecuada. La ejecución **paralelo** consigue entonces 4 veces más que la secuencial.

MIC:

- Es la función **mkl** diseñada para un coprocesador (**offload**) la que consigue aprovechar realmente la arquitectura, superando 90 de speedup.

- Teniendo activa la variable MKL_MIC_ENABLE, vemos que en cuanto el tamaño de las matrices supera el especificado (en nuestro caso fila/columna > 2048), automáticamente se distribuye la carga entre host y mic de la mejor manera (en nuestro caso 0.09/0.91, y conseguimos un excelente speedup de 70.

- El programa **offload** diseñado gestiona el movimiento de los datos al coprocesador, para después computar la multiplicación de matrices. Utilizando todos los cores con 4 threads por core, conseguimos un buen speedup entre 15 y 20. El programa en **nativo** tiene resultados similares, sin llegar a las muy optimizadas funciones mkl.

		1.024		2.048		4.096			
		T_medio	SpeedUp	T_medio	SpeedUp	T_medio	SpeedUp	VEC	PAR
HOST	O0	7743	0,02	170206	0,01	2164255	0,01	✗	✗
	O1	2825	0,06	61261	0,04	956857	0,02	✗	✗
	O2_no-vec	556	0,32	4745	0,49	38645	0,49	✗	✗
	O2	178	1,00	2307	1,00	18930	1,00	✓	✗
	O3	290	0,61	2320	0,99	18536	1,02	✓	✗
	xHost	119	1,50	2079	1,11	17432	1,09	✓	✗
	Avx2	120	1,48	2089	1,10	17407	1,09	✓	✗
	Omp	56	3,18	798	2,89	6421	2,95	✓	✓
	matmul	13	13,69	69	33,43	517	36,62	✓	✓
	mkl:seq	26	6,85	192	12,02	1507	12,56	✓	✗
	mkl:par	8	22,25	53	43,53	389	48,66	✓	✓
MIC	mkl:off	4	44,50	25	92,28	212	89,29	✓	✓
	Offload	12	14,83	118	19,55	1145	16,53	✓	✓
	AO					266	71,17	✓	✓
	Native	13	13,69	142	16,25	1459	12,97	✓	✓

Tabla 4.2 Resultados multiplicación de matrices

0,00 < SpeedUp < 1,00
1,00 < SpeedUp < 10,00
10,00 < SpeedUp < 20,00
20,00 < SpeedUp < 10,00

Mostraremos un ejemplo del tipo de programa limitado por capacidad de procesamiento (**cpu bound**), como la elegida multiplicación de matrices:

La transferencia de datos del host al coprocesador es un bus PCIe v2 a **8 GB/s**. Activando el reporte de offload con OFFLOAD_REPORT=2, fijándonos en el tiempo de transferencia y de cómputo, vemos que son los cálculos los que dominan el tiempo de la ejecución.

C[4096][4096] → 16777216 elems x 4B/elem = 67108864B = 64MB

PCIexpress: 8GB/s

8192MB	---	1s
64MB	---	x

x = 64/8192 = 0,0078s = 7,8ms

Input(y) = 7,8 + Output(y) = 7,8 → 15,6ms

```
                Tag 4
[CPU Time]      0.972488(seconds)
[CPU->MIC Data] 67108880 (bytes)
[MIC Time]      0.950004(seconds)
[MIC->CPU Data] 67108880 (bytes)
```

Ilustración 4.9 Reporte offload multiplicación de matrices

[Reporte] CPUtime – MICtime = 972,5ms – 950ms = **22,5ms**

* Por el contrario, en este caso, el 2,3% de la ejecución lo ocupa la transferencia de datos y el 97,7% restante el cómputo.

4.4. Mixed bound

En este apartado hemos desarrollado una aplicación que por su comportamiento se situaría en un punto intermedio entre las aplicaciones memory bound y cpu bound.

Hemos cambiado el enfoque y en lugar de ver su comportamiento en base a optimizaciones como anteriormente, lo hemos hecho con diferentes tamaños de entrada de peticiones y base de datos para comprobar la escalabilidad.

La tarea que realiza este programa es que, recogiendo vectores de elementos de un fichero de peticiones entrante, realice una comparación (usando para ello la distancia euclediana) contra una matriz de datos, almacenando cada resultado y mostrando la mínima de todas ellas.

El código se lista en el Anexo A, primero el que contiene serie/paralelo, y después el de serie/ offload.

El flujo del programa se explica en detalle a continuación, teniendo 3 variantes de ejecución.

- Euclidean distance

- Serie
    1. Crea un fichero que actuará como peticiones y otro como base de datos.
    2. Por cada petición, calcula la distancia euclediana con cada una de las entradas de la base de datos, y devuelve el índice de la que sea menor.
    3. Guarda por tanto en un vector resultante los índices de las menores distancias para cada petición.

- Paralelo
    1. Igual que Serie(1).
    2. Por cada petición, cada thread calcula la distancia euclediana con una entrada de la base de datos y cuando termina lo vuelve a hacer con otra mientras no se hayan comprobado todas las entradas.
    3. Almacenan los índices y sus menores distancias eucledianas en dos vectores de longitud número_de_threads (4 en este caso), y posteriormente se recorren los mismos para quedarse con el adecuado menor índice.
    4. Igual que Serie(3).

- Offload
    1. Igual que Paralelo(1).
    2. Se transfiere la BD al coprocesador.
    3. Simultáneamente [ver 4.2.1.1]:
        a. [MIC] Por cada petición, se calcula el índice de la mínima distancia eucledianа de la misma manera que en Paralelo(2), con la diferencia de que se asignará un trozo de la base de datos cada uno de los múltiples en ejecución.
        b. [HOST] Lee nueva petición del fichero de peticiones.
    4. Obtiene el menor índice de la misma manera que Paralelo(3).
    5. Igual que Paralelo(4).

* En el caso de offload, se ha realizado unrolling de grado 2 en el bucle con el objetivo de evitar los errores de sobreescritura en el vector de entrada que si aparecen con iteraciones del bucle unitarias.

4.4.1.- Compilación y Reportes

Mostramos los **ficheros de compilación** .bat (batch file) de Windows utilizados para generar los ejecutables de nuestro programa.

Añadimos **algunos reportes** para comprobar cuales de ellas aplican técnicas de paralelismo y vectorización. Para generarlos utilizamos el flag /Qopt-report-phase = [vec: para comprobar vectorización] [openmp: para comprobar paralelismo].

- Serie y paralelo

```
icl /Qopenmp /Qopt-report-phase=vec,openmp de.c
de.exe bd pet
```

```
OpenMP Construct at C:\Users\ehu\Documents\jaldazabal005
\euclidean\de.c(147,5) inlined into C:\Users\ehu\Documents
\jaldazabal005\euclidean\de.c(225,10)
   remark #16200: OpenMP DEFINED LOOP WAS PARALLELIZED

    Report from: Vector optimizations [vec]
```

4 threads se dividen la BD para el cálculo

```
#pragma omp parallel for private(res,id) firstprivate(nbd) num_threads(NTHR)
    for(i=0; i<nbd; i++)
        {
            id = omp_get_thread_num();
            res= calcular_distancia(vector, &bd[i][0],K);
            if (elMenor[id*64] > res)
            {
                elMenor[id*64]=res; menor[id*64]=i;
```

Ilustración 4.10 Región paralela Xeon

```
for (i=0; i<nvector; i++)
{
    acum += (vec1[i]-vec2[i]) * (vec1[i]-vec2[i]);
}
distancia = sqrt(acum);
```

```
      LOOP BEGIN at C:\Users\ehu\Documents\jaldazabal005
\euclidean\de.c(106,2) inlined into C:\Users\ehu\Documents
\jaldazabal005\euclidean\de.c(215,10)
         remark #15300: LOOP WAS VECTORIZED
      LOOP END
   LOOP END
LOOP END
```

Ilustración 4.11 Vectorización avx2

- Serie y offload

```
icl /Qopenmp /Qopt-report-phase=vec,openmp offmic.c
offmic.exe bd pet
```

```
OpenMP Construct at C:\Users\ehu\Documents\jaldazabal005
\euclidean\off\offmic.c(183,11)
   remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED

    Report from: OpenMP optimizations [openmp]

OpenMP Construct at C:\Users\ehu\Documents\jaldazabal005
\euclidean\off\offmic.c(207,11)
   remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED

    Report from: OpenMP optimizations [openmp]

OpenMP Construct at C:\Users\ehu\Documents\jalda
\euclidean\off\offmic.c(233,11)
   remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED
```

224 threads calculan cada iteración en la BD

```
    #pragma offload target(mic:0) nocopy(bd: REUSE) in(vector:length(K)
{
        #pragma omp parallel num_threads(NTHR)
    {
        parcomparar(vector, bd, nbd, resultadosthreads, indicethreads);
    }
}

leer_entrada(fptr, vector1, fname2, i+1, K);


#pragma offload_wait target(mic:0) wait(&tag)
```

Ilustración 4.12 Región paralela Xeon Phi

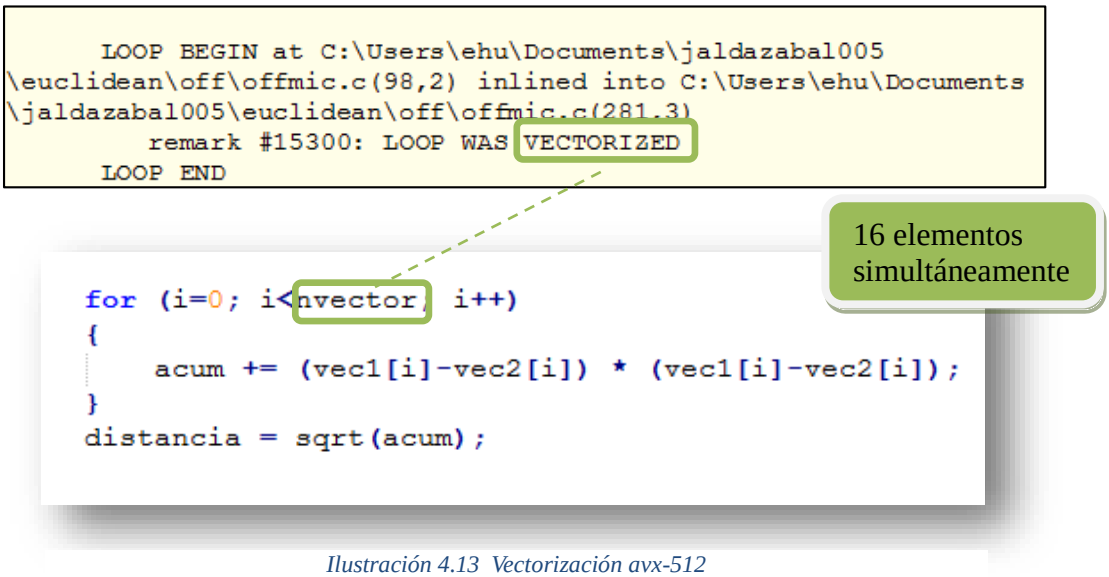


Ilustración 4.13 Vectorización avx-512

PARALELISMO

Podemos ver como en los dos programas desarrollados, conseguimos la **paralelización** a la hora de comparar contra la matriz de datos (BD):

- En el caso del paralelismo en el host es la matriz de datos la que se divide en 4 partes y cada thread trabaja sobre una (ilustración 4.10).
- En cuanto al paralelismo en el coprocesador, en cambio, cada uno de los múltiples threads (224) trabaja con su vector contra una entrada de la matriz de datos y trabajan sobre ella de manera iterativa (ilustración 4.12).

VECTORIZACIÓN

La **vectorización** también se consigue en el momento de calcular, de los 160 elementos de ancho, la distancia para 16 elementos a la vez usando los registros de 512 bits, de la misma manera en ambos modelos como vemos en las ilustraciones 4.11 y 4.13.

4.4.2.- Resultados y conclusiones

Para analizar la función, realizamos varias ejecuciones considerando serie, paralelo en host y offload en el coprocesador, para ver su evolución con distintos tamaños de entrada de peticiones y de base de datos como los siguientes [BD – PET]:

BD	1.000	10.000	100.000	100.000	1.000.000	1.000.000	1.000.000	2.000.000
PET	100	100	100	1.000	100	1.000	10.000	10.000

		BD	PET	Tmedio	Speed-Up
HOST	Serie	1000	100	0,007	1
		10000		0,035	1
		100000		0,473	1
		1000000	1000	4,713	1
			100	4,661	1
			1000	47,160	1
			10000	469,000	1
		2000000		937,422	1
	Parallel	1000	100	0,009	0,78
		10000		0,026	1,35
		100000		0,207	2,29
		1000000	1000	1,950	2,42
			100	1,925	2,42
			1000	18,800	2,51
			10000	187,000	2,51
		2000000		375,255	2,50
MIC	Offload	1000	100	0,460	0,02
		10000		0,728	0,05
		100000		1,443	0,33
		1000000	1000	4,424	1,07
			100	3,886	1,20
			1000	11,048	4,27
			10000	88,000	5,33
		2000000		166,113	5,64

Tabla 4.3 Resultados distancia euclediana

0,00 < SpeedUp < 1,00
1,00 < SpeedUp < 2,00
2,00 < SpeedUp < 3,00
3,00 < SpeedUp < 10,00

Esta vez resumiremos los aspectos más importantes que se pueden extraer de la batería de ejecuciones realizada en los modelos en serie, paralelo y offload:

- En cuanto al modelo paralelo en el host puede apreciarse que prácticamente desde el inicio supera en rendimiento a la ejecución serie independientemente del tamaño de los datos.

- Una de las conclusiones que vemos es que la utilización de offload no es rentable con un uso de datos no lo suficientemente grande cuya causa es el overhead que supone la activación de 224 threads.

- Observando los resultados obtenidos, apreciamos una limitación en paralelo a un valor de speedup de 2,50, mientras que utilizando el offload la convergencia resultante se sitúa en torno a 6,00.

4.5. Evaluación Rendimiento Energético

4.5.1.- Mediciones de consumo

Las mediciones se han realizado con Intel Power Gadget 3.0 para el Xeon E5, y con MicSMC 3.6.51 para la Xeon Phi, como se puede observar en las ilustraciones 4.14, 4.15 y 4.16.

```
149  Total Elapsed Time (sec) = 29.537905
150  Measured RDTSC Frequency (GHz) = 3.093
151
152  Average Processor Power_0 (Watt) = 21.230508
153
154  Average DRAM Power_0 (Watt) = 14.921738
```

Ilustración 4.14 Medición consumo serie Xeon mxm

```
107  Total Elapsed Time (sec) = 21.045348
108  Measured RDTSC Frequency (GHz) = 3.093
109
110  Average Processor Power_0 (Watt) = 56.079807
111
112  Average DRAM Power_0 (Watt) = 15.832780
113
```

Ilustración 4.15 Medición consumo paralelo Xeon mxm

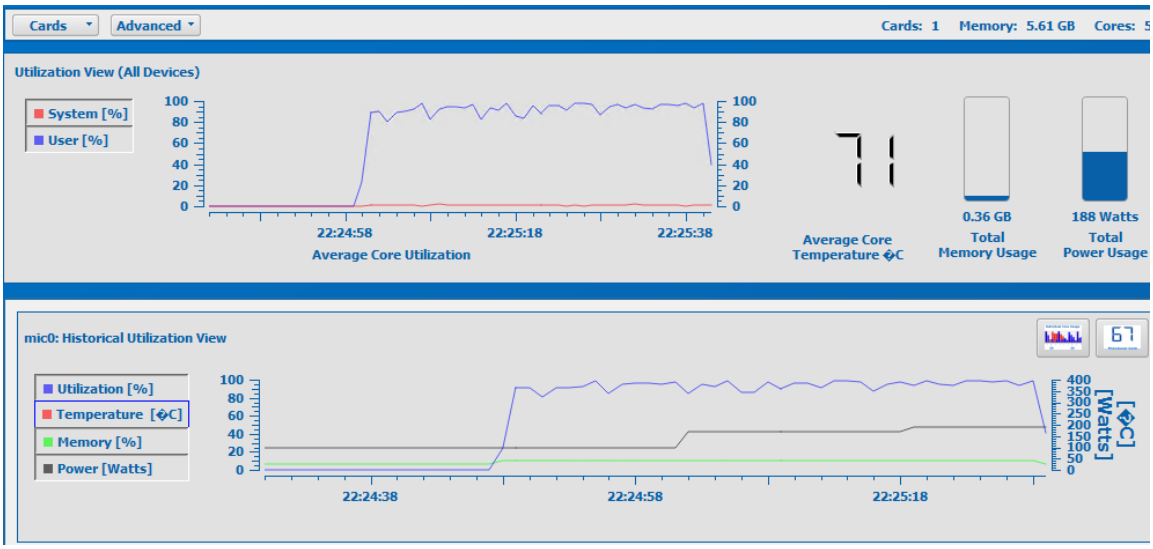


Ilustración 4.16 Medición consumo offload Xeon Phi mxm

4.5.2.- Resultados

En este apartado se muestran los resultados de consumo energético para los programas descritos en nuestras dos plataformas, comparando el modelo **paralelo en HOST** y el modelo **offload del MIC** . Los valores de energía consumida se expresan en Wh (vatio-hora).

t0 : Tiempo de ejecución paralela en HOST (seg)
t1 : Tiempo idle en host (seg)
t2 : Tiempo de ejecución de offload a MIC (seg)
t3 : Tiempo idle en xeon phi (seg)

El consumo es calculado considerando el tiempo de ejecución:

- Si es mayor, este tiempo será multiplicado por el consumo de energía en ejecución.
- Si es menor, el consumo en el estado de ejecución lo sumaremos al consumo en estado idle que resta hasta igualar el tiempo máximo.

SAXPY

	Pot_ejec (w)	Pot_idle (w)
Host_paralelo	73	19
Mic_offload	144	40

Tabla 4.4 Consumo saxpy

1.- Paralelo en HOST

2.- Offload en MIC

- Consumo_HOST= (t0 * Pot_ejec) + (t1 * Pot_idle) = (0,018s*73w) + (0,047s*19w) = 2,2 Ws = **0,6 * 10^{-3} Wh**

- Consumo_MIC = t2 * Pot_ejec = 0,065s * 144w = 9,4 Ws = **2,6 * 10^{-3} Wh**

MXM

	Pot_ejec (w)	Pot_idle (w)
Host_paralelo	72	19
Mic_offload	188	40

Tabla 4.5 Consumo multiplicación de matrices

1.- Paralelo en HOST

2.- Offload en MIC

- Consumo_HOST= t0 * Pot_ejec = (6,421s*72w) = 462,3 Ws = **128 * 10^{-3} Wh**

- Consumo_MIC = (t2 * Pot_ejec) + (t3 * Pot_idle) = (1,445s*188w) + (5,276s*40w) = 482,7 Ws = **134 * 10^{-3} Wh**

EUCLIDEAN

	Pot_ejec (w)	Pot_idle (w)
Host_paralelo	76	19
Mic_offload	154	40

Tabla 4.6 Consumo Euclidean

1.- Paralelo en HOST

2.- Offload en MIC

- Consumo_HOST= t0 * Pot_ejec = (18,800s*76w) = 1428,8 Ws = **397 * 10^{-3} Wh**

- Consumo_MIC = (t2 * Pot_ejec) + (t3 * Pot_idle) = (11,048s*154w) + (7,752s*40w) = 2011,5 Ws = **559 * 10^{-3} Wh**

Podemos observar que, **en cuanto a consumo** para los programas considerados:

- En saxpy el consumo energético es mayor en el Phi en un **433%**, y como además requiere más tiempo concluimos que la ejecución en el host es de lejos preferible para *memory bound*.

- En cuanto a la multiplicación de matrices, vemos que ambos usan prácticamente la misma energía (**mic un 5%** más), necesitando el host más del cuádruple de tiempo para completarse por lo que el uso del Phi en *cpu bound* es una opción muy buena.

- Para el caso del programa desarrollado, el **Phi** requiere un **40%** más de energía que el host, y unido a que consigue el doble de speedup podemos considerar que en programas de tipo mixto usar el Xeon Phi es una elección que merece la pena.

4.5.3.- Plan de energía

Todas las ejecuciones del host han sido realizadas con la configuración de energía **equilibrada**. Sin embargo, mostraremos en este apartado el efecto de activar la opción **alto rendimiento** en la gestión de energía del host.

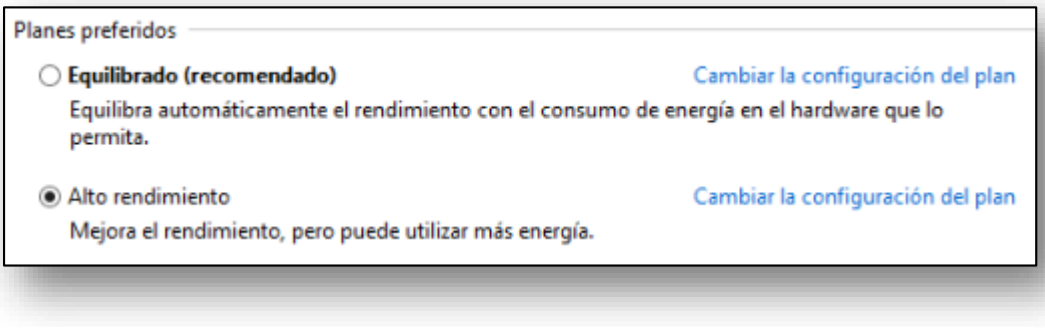


Ilustración 4.17 Planes de energía

Comparamos la ejecución del mismo programa (p.e. distancia euclediana) en serie con las dos configuraciones para ver el impacto del plan de energía.

	Pot_ejec (w)	Pot_idle (w)
Serie_equilibrado	35	19
Serie_alto_rend.	56	19

Tabla 4.7 Consumo plan de energía

1.- Equilibrado_s

2.- Alto_rendimiento_s

- Consumo_Equil = t0 * Pot_ejec = (81,226s*35w) = 2842,9 Ws = **790 * 10^{-3} Wh**

- Consumo_Alto = (t2 * Pot_ejec) + (t3 * Pot_idle) = (47,580s*56w) + (33,646s*19w) = 3303,8 Ws = **918 * 10^{-3} Wh**

Equilibrado

* El plan de energía **equilibrado**, requiere de casi el doble de tiempo para ejecutarse que en alto rendimiento.

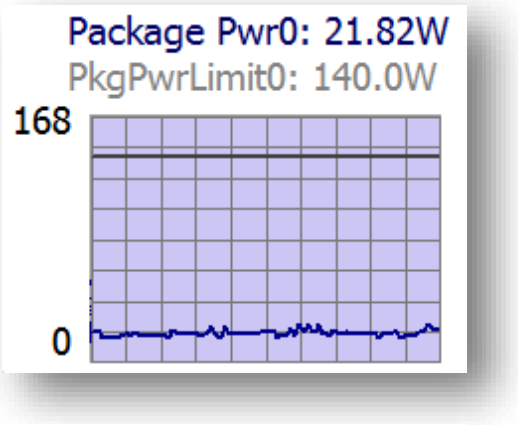


Ilustración 4.18 Equilibrado

Alto rendimiento

* El plan de energía de **alto rendimiento** muestra que necesita sólo un 16% más de energía para completarse en la mitad de tiempo que en equilibrado. Es por tanto una configuración recomendable.

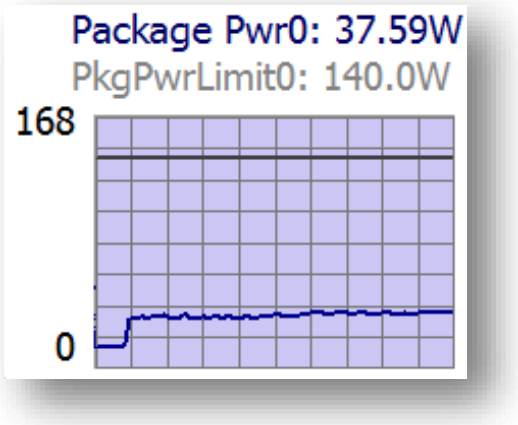


Ilustración 4.19 Alto rendimiento

5

Planificación

5.1. Planificación

El diagrama de Gantt de la ilustración 5.1 muestra una visión general de la realización del proyecto.

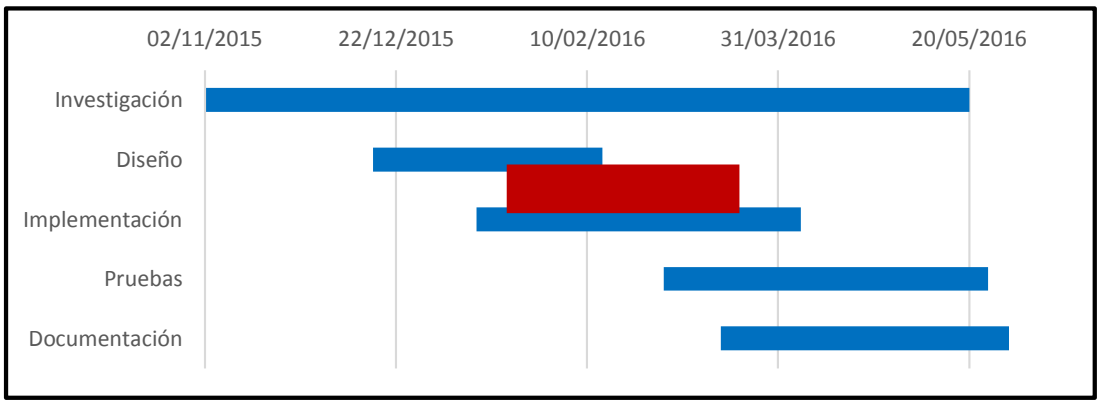


Ilustración 5.1 Planificación del proyecto

Durante el transcurso del proyecto, hemos tenido una contingencia grave para el transcurso del mismo.

La instalación inicial y puesta en marcha del coprocesador Intel Xeon Phi fue correcta. Sin embargo en mitad del desarrollo y pruebas de ejecución surgieron **cuelgues aleatorios**, que impedían continuar con el trabajo hasta que el responsable técnico del centro reinstalara el sistema. Esto provocaba un parón y posterior búsqueda de soluciones a los problemas que creíamos causar.

Finalmente se detectó, tras realizar comprobaciones, un **fallo en el hardware**. Se informó al proveedor y el grupo de investigación de Alex Mendiburu **pudo proporcionar otro coprocesador** parecido que pidió **el director,** que estaba en correcto estado para poder utilizarlo. Esta contingencia tuvo un impacto en el alcance, reduciéndolo, y evitando la inclusión de otras ideas.

5.2. Reuniones de progreso

La siguiente tabla muestra una serie de reuniones de progreso concertadas entre director y alumno.

Fecha	Descripción
02/11/2015	Inicio del proyecto; Designación de tareas a realizar
11/11/2015	Puesta en marcha de escenario; Comentarios sobre programa1
24/11/2015	Mostrar guía de configuración y acceso; Progreso de programas1,2
01/12/2015	Modelos a desarrollar (serie, parallel, native, offload)
15/12/2015	Presentar multiplicación matrices; Dudas versión tiled programa2
21/01/2016	Optimizaciones en compilador; Contingencia Xeon Phi
09/02/2016	Script launcher; modelos MKL; Contingencia Xeon Phi
08/03/2016	Contingencia Xeon Phi
14/03/2016	Contingencia Xeon Phi; Diseño programa3 euclidean distance
28/03/2016	Defecto fabricación Xeon Phi, sustituir; Continuar desarrollo programa
18/04/2016	Mostrar resultados ejecuciones; Depuración del programa3
28/04/2016	Avance de la memoria y recopilar resultados
16/05/2016	Cerrar Resultados para gráficas; Comentar consumo energético
08/06/2016	Entregar prototipo de memoria
22/06/2016	Entregar memoria final

Tabla 5.1 Reuniones de progreso

Además de varias explicaciones recibidas por parte del director, debemos señalar que aproximadamente el 90% de los contenidos de este proyecto son totalmente nuevos para el alumno, desde el uso de un sistema heterogéneo síncrono/asíncrono, o el compilador ICC y sus optimizaciones, pasando por herramientas como Intel Cilk / MKL hasta la medición de consumo energético. Ha sido necesaria por tanto una investigación exhaustiva de todos estos temas estudiando su documentación y manuales.

6

Conclusiones

6.1. Conclusiones

Las conclusiones concretas que podemos obtener tras la realización de este proyecto las resumimos a continuación:

- En cuanto al sistema empleado, después de trabajar intensamente con él, hemos visto que una de las principales ventajas es la **facilidad a la hora de su utilización**.

  Hemos comprobado cómo la **portabilidad del código** existente para su ejecución en plataformas heterogéneas o nativas es casi inmediata (introducir pocas directivas o añadir un flag para compilación).

  > "*Moving a code to Intel Xeon Phi might involve sitting down and adding a couple lines of directives that takes a few minutes. Moving a code to a GPU is a project*"
  >
  > -- Subdirector del Texas Advanced Computing Centre, Dan Stanzione –

- En la misma línea, las herramientas utilizadas para el desarrollo de aplicaciones que exploten **el paralelismo y la vectorización** han resultado muy intuitivas, donde tenemos que destacar la **generación de reportes** ofreciendo una información esencial que nos ayuda en tal propósito.

  También hemos podido ilustrar **las posibilidades que ICC** nos ofrece, y que conociendo sus opciones podemos **aumentar el rendimiento** de nuestras aplicaciones en gran medida (y si éstas utilizan funciones matemáticas aprovechar las existentes en la medida de los posible con simples llamadas).

- En lo referente a las aplicaciones desarrolladas, hemos visto que el tipo de aplicaciones limitadas por memoria , memory bound, no consigue sacar provecho de una arquitectura como el Xeon Phi, debido a la baja velocidad del bus de datos que provoca un alto porcentaje en tiempo de transferencia.

  Sin embargo, en las aplicaciones limitadas por proceso, *cpu bound*, sí merece la pena el uso de coprocesadores como los descritos gracias a su alto número de *cores* con la arquitectura especial explicada, destacando el cálculo de operaciones de coma flotante.

  En relación a las aplicaciones de tipo mixto como la propuesta, vemos que podemos conseguir mayor rendimiento aunque no muy significativo, y aumentando el consumo energético por lo que tendremos que evaluar la rentabilidad en cada caso concreto.
  Es posible que sin utilizar este sistema, y mediante paralelismo en el host junto con Turbo-Boost (aumento de la frecuencia por encima de lo normal cuando es requerida), podamos acercarnos bastante, de momento, al rendimiento ofrecido por estos coprocesadores.

Como conclusiones generales, debemos señalar que son varios los factores a estudiar para la adopción de un sistema como el utilizado:

1. El rendimiento. Es vital que previamente consideremos si nuestras aplicaciones son aptas para aprovechar los beneficios de un sistema de este tipo (alto grado de paralelismo, carga de trabajo vectorizable, escalable…).
2. El consumo. Será necesario evaluar si el rendimiento conseguido utilizando esta plataforma no es a expensas de un consumo energético que no consideremos asumible.
3. El precio. Como en muchas ocasiones, el coste de adquirir este tipo de hardware puede ser notable, además de factores como adaptación y posibles contingencias (por ejemplo un defecto del hardware como el aparecido en este proyecto).

Todas estas cuestiones deben ser tenidas en cuenta. Sin embargo, la utilizada es la primera generación de coprocesadores x100, cuyo relevo toman los inminentes *Knights Landing*, por lo que muchas de las limitaciones desaparecen debido a la superioridad de estos últimos.

6.2. Líneas futuras

El alcance de este proyecto no contempla investigar en profundidad los siguientes apartados, aunque sin duda sería muy interesante. La evaluación en este proyecto del sistema descrito puede ser mejorada ya sea **añadiendo más coprocesadores** formando un cluster MPI, o bien **actualizando el modelo** al de nueva generación, *Knights Landing*.

6.2.1 MPI Cluster

Una alternativa es utilizar una serie de coprocesadores (xeon phi) que trabajan conjuntamente donde la comunicación se realiza a través de paso de mensajes (MPI), que es una librería de funciones que permite el envío y recepción de mensajes entre procesos (movimiento de datos + sincronización). Los modelos de ejecución en este caso serían:

- Nativamente: donde cada Xeon Phi es un nodo y ven al resto de coprocesadores como otro nodo en la red.
- Simétricamente: donde el programa mpi se ejecuta en modelo heterogéneo de host y coprocesadores en igual jerarquía.
- Offload: donde las funciones mpi ocurren en los hosts Xeon, y cada uno descarga el cómputo a sus coprocesadores.

El esquema de un sistema de estas características sería de la manera que se puede apreciar en la tercera opción de la ilustración 6.1.

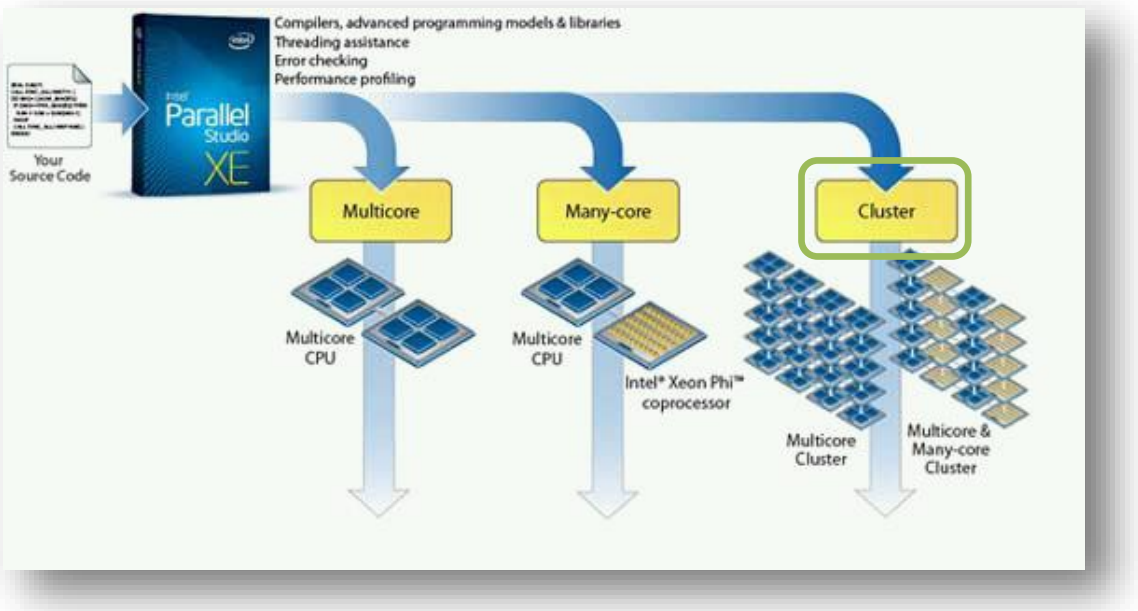


Ilustración 6.1 Modelo de cluster

6.2.2 Knights Landing

El nuevo coprocesador previsto para el tercer trimestre de 2016, supone un gran avance en prestaciones respecto a la anterior generación, y no sólo será sistema de expansión, sino que podrá utilizarse como una plataforma autónoma. Se listan a continuación las características principales:

- 36 *tiles* interconectadas por una topología de malla 2D.
- Cada tile contiene 2 cores (Silvermont 14nm 1.3GHz), con 2 VPUs/core y 1MB de cache L2 (total: 36MB, 72cores).
- 16BG Multi Channel DRAM (400+ GB/s) y hasta 384GB DDR4 (90+ GB/s).
- 3 modos de memoria de gran ancho de banda o HBM, *high memory bandwidth*;
    - Utilizar MCDRAM como si fuera caché L3.
    - Como un nodo de memoria distinto (NUMA).
    - Una mezcla de ambos.
- 36 canales PCIe version 3.
- 6+Tflops en simple precisión (vs 2TF en Knights Corner).
- 5x eficiencia energética.

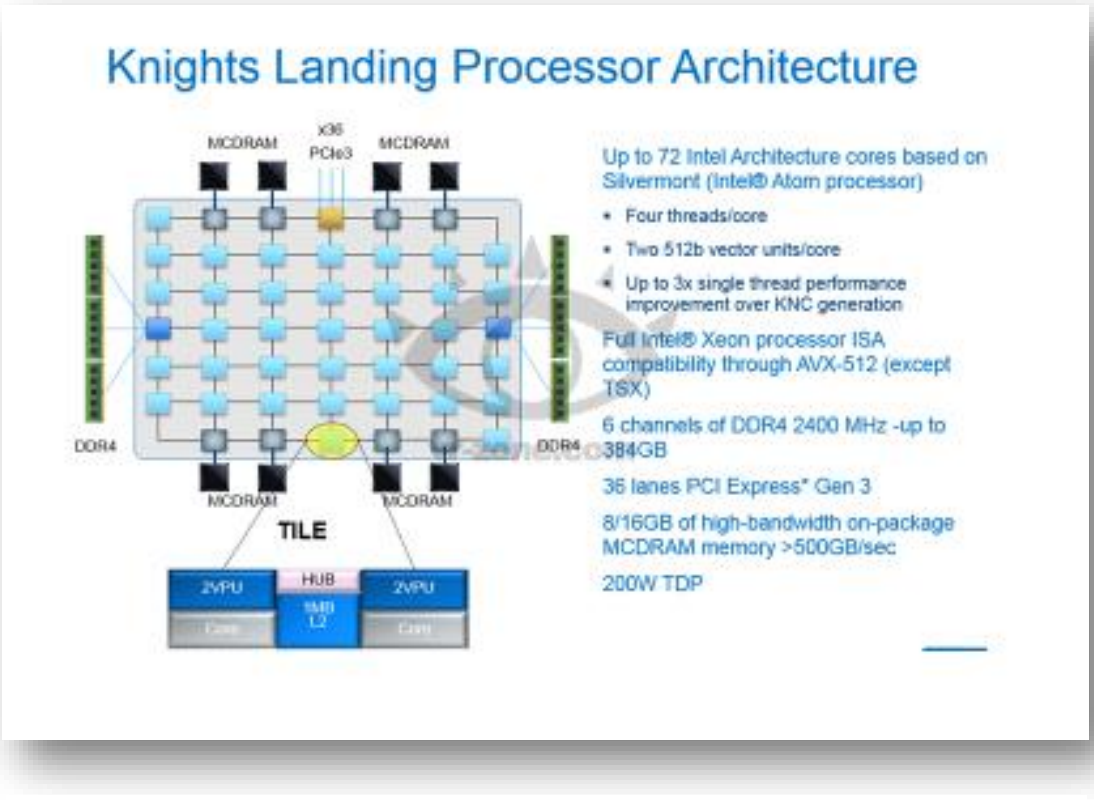


Ilustración 6.2 Esquema de arquitectura Knights Landing

Virtualmente **todas las aplicaciones** que corren actualmente en procesadores basados en la plataforma **Intel Xeon funcionan sin modificación en Knights Landing**. Una buena recomendación es recompilar para KNL con el fin de obtener el máximo rendimiento, y con esta arquitectura dependerá todo de:

- Cómo de paralelizable sean las tareas para aprovechar el **número de cores**,
- Del uso de las dos **unidades de proceso vectoriales**
- De la sensibilidad al **ancho de banda de memoria**.

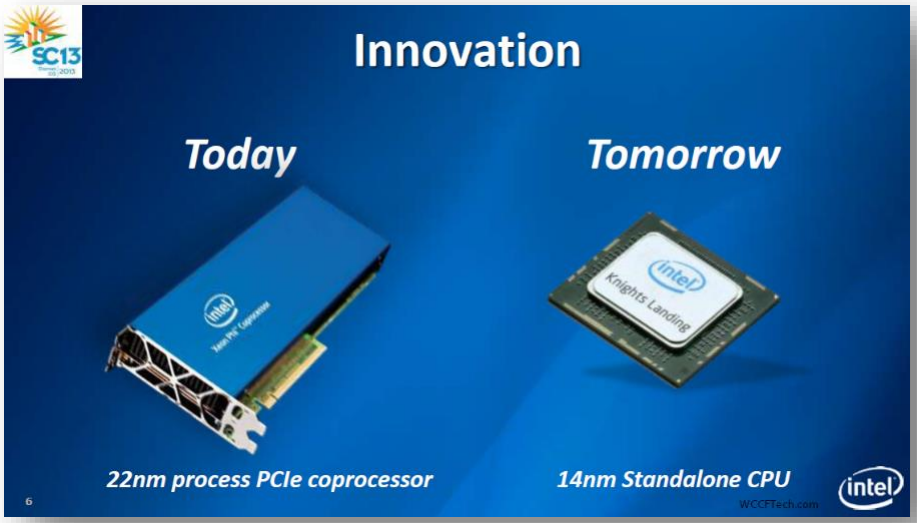


Ilustración 6.3 Salto generacional Knights Corner a Knights Landing

Bibliografía

[1] Datasheet Intel Xeon Phi. www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html

[2] Guía práctica Intel Xeon Phi. www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/

[3] Midiendo el rendimiento en HPC. software.intel.com/en-us/articles/measuring-performance-in-hpc

[4] Quick overview Xeon Phi. www.cism.ucl.ac.be/XeonPhi.pdf

[5] Start guide for Windows. software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide-windows-v1-2.pdf

[6] Generación de reportes. software.intel.com/en-us/articles/vectorization-and-optimization-reports

[7] Modelos de programación. software.intel.com/sites/default/files/ee/90/offload-compiler-runtime-for-the-intel-xeon-phi-coprocessor-130315.pdf

[8] Alineamiento de datos. software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

[9] Preguntas más comunes. software.intel.com/en-us/forums/intel-many-integrated-core/topic/360754

[10] Uso de Intel MKL. www.training.prace-ri.eu/uploads/tx_pracetmo/MKL_4_MIC.pdf

[11] Intel Cilk Plus. www.cilkplus.org/cilk-plus-tutorial

[12] Notación array de Intel. www.cilkplus.org/tutorial-array-notation

[13] Notación array de Intel en profundidad. software.intel.com/sites/default/files/managed/f6/aa/Intel%20Cilk%20Plus%20Array%20Notation%20-%20Technology%20and%20Case%20Study%201.2.pdf

[14] Diagrama Xeon Phi. www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-block-diagram.html

[15] Especificaciones Xeon Phi. ark.intel.com/products/75797/Intel-Xeon-Phi-Coprocessor-3120A-6GB-1_100-GHz-57-core

[16] Compilar y optimizar. www.cs.unc.edu/~prins/Classes/633/Readings/Vecpar-Intel-Phi.pdf

[17] Herramienta web de apoyo para compilación MKL. software.intel.com/en-us/articles/intel-mkl-link-line-advisor

[18] Modelo asíncrono de offload. software.intel.com/sites/default/files/article/326700/6.2.1-asynchronous-offload.pdf

[19] Resumen offload. portal.tacc.utexas.edu/c/document_library/get_file?uuid=e8da0d9d-257b-4053-9998-1f017aee048f&groupId=13601

[20] Modelo offload heterogéneo. software.intel.com/sites/default/files/managed/05/ba/heterogeneous-programming-model.pdf

[21] Descripción Knights Landing 2016. wccftech.com/intel-knights-landing-detailed-16-gb-highbandwidth-ondie-memory-384-gb-ddr4-system-memory-support-8-billion-transistors/

de.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <float.h>
#include <limits.h>
#include <string.h>

#define NTHR 4
#define NBD  1000000
#define NPET 1000
#define K 160
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)
__declspec(align(64)) static float bd[NBD][K];
__declspec(align(64)) static int serieresul[NPET];
__declspec(align(64)) static int paraleloresul[NPET];

void leer_entrada(FILE *fptr, float *vector, const char *name, int indice, int n)
{
        int i, j;
        if(( fptr = fopen(name, "r")) == NULL)
        {
                printf("No es posible abrir el fichero");
        }
                fseek(fptr, indice+(indice * sizeof(float) * K), SEEK_SET);
        //PUNTERO DE ARCHIVO (+ind para el \n)
        //printf("ftell = %d \n", ftell(fptr)); //imprime posicion apuntada en el fichero
                for(i=0;i<n;i++)fscanf(fptr, "%f", &vector[i]);
        fclose(fptr);
}

void leer_BD(FILE *fptr, float bd[][K], const char *name){
        int i, j;
        if(( fptr = fopen(name, "r")) == NULL)
        {
                printf("No es posible abrir el fichero");
        }
        printf("Leyendo BD...\n");
                for(i=0;i<NBD;i++)
                        for(j=0;j<K;j++)
                                fscanf(fptr, "%f", &bd[i][j]);
        fclose(fptr);
}

void escribir(FILE *fptr, int rango, const char *name,int n){
        int i, j;
        double num;
        if(( fptr = fopen(name, "wt")) == NULL)
        {
                printf("No es posible abrir el fichero");
        }
        printf("Escribiendo fichero %s\n", name);
                for(i=0;i<n;i++)
                {
                        for(j=0;j<K;j++){
                                num = rand() % rango;
                                fprintf(fptr, "%4.0f", num);
                        }
                        fprintf(fptr, "%s", "\n");
                }
        fclose(fptr);
}
```

```
void results(FILE *fptr, const char *name, int n, int *v1){
        int i, j;
        double num;
        if(( fptr = fopen(name, "wt")) == NULL)
        {
                printf("No es posible abrir el fichero");
        }
        printf("Escribiendo fichero %s\n", name);
                for(i=0;i<n;i++)
                {
                        fprintf(fptr, "%d  ", v1[i]);
                }
        printf("\n");

        fclose(fptr);
}

float calcular_distancia(float *vec1, float *vec2, int nvector)  //pasar 16 y 16 elems y
calcular "DE" entre ellos
{
        int i;
        float acum, distancia = 0;

        for (i=0; i<nvector; i++)
        {
                acum += (vec1[i]-vec2[i]) * (vec1[i]-vec2[i]);
        }
        distancia = sqrt(acum);

return distancia;
}

int comparar(float *vector, float bd[][K],int nbd)
{
        int i;
        float elMenor = FLT_MAX, res;
        int menor = INT_MAX;
                for(i=0; i<nbd; i++)
                        {
                            res= calcular_distancia(vector, &bd[i][0],K);
                            if (elMenor > res)
                            {
                                elMenor=res; menor=i;
                            }
                        }
        return menor;
}

int parcomparar(float *vector, float bd[][K],int nbd)
{
        int i,id;
        __declspec(align(64)) float elMenor [NTHR*64];
        float res;
    __declspec(align(64)) int menor[NTHR*64];  // evitar falsa comparticion
    float localmenor;
    int localquien;
    for (i=0;i<NTHR;i++)
        menor[i*64]=INT_MAX;
    for (i=0;i<NTHR;i++)
        elMenor[i*64]= FLT_MAX;
    #pragma omp parallel for private(res,id) firstprivate(nbd) num_threads(NTHR)
                for(i=0; i<nbd; i++)
                        {
                            id = omp_get_thread_num();
                            res= calcular_distancia(vector, &bd[i][0],K);
                            if (elMenor[id*64] > res)
                            {
                                elMenor[id*64]=res; menor[id*64]=i;
                            }
                        }
    localmenor = elMenor[0];
```

```
    localquien = menor[0];
    for (i=1;i<NTHR;i++)
        if (elMenor[i*64]<localmenor)
        {
           localmenor=elMenor[i*64];
           localquien=menor[i*64];
        }
        return localquien;
}

__declspec(align(64)) float vector[K];

void serie (int npet, FILE *fptr, const char *fname2, float *vector, float bd[][K], int
*resul,int nbd)
{
    int i, elmin;
        for(i=0;i<npet;i++)
        {
                leer_entrada(fptr, vector, fname2, i,K);
                elmin=comparar(vector, bd,nbd);
                resul[i] = elmin;
        }
}

void paralelo (int npet, FILE *fptr, const char *fname2, float *vector, float bd[][K], int
*resul,int nbd)
{
    int i;
        for(i=0;i<npet;i++)
        {
                leer_entrada(fptr, vector, fname2, i,K);
        resul[i]=parcomparar(vector, bd, nbd);
        }
}

int main(int argc, char *argv[])
{
        int i, j;
        FILE *fptr = NULL;
        FILE *fptr2 = NULL;
        char tag;
        char fname1[100];
        char fname2[100];
        long tej;
        double w1;
    strcpy (fname1,argv[1]);
    strcpy (fname2,argv[2]);
        escribir(fptr, 256, fname1, NBD);       // crear Base datos Mic
        leer_BD(fptr, bd, fname1);                  // fichero bd en estructura de datos bd
        escribir(fptr, 256, fname2, NPET);              //fichero fuente de peticiones
        printf("\nTerminada lecutra BD \n");
        w1=omp_get_wtime();
        serie (NPET,fptr,fname2,vector,bd,serieresul,NBD);
        w1=(omp_get_wtime()-w1)*1000;
        printf("\nTiempo transcurrido serie: %f mseg \n", w1);
                Sleep(10);
        w1=omp_get_wtime();
        paralelo(NPET,fptr,fname2,vector,bd,paraleloresul,NBD);
        w1=(omp_get_wtime()-w1)*1000;
        printf("\nTiempo transcurrido paralelo: %f mseg \n", w1);
     for (i=0;i<NPET;i++)
        if (serieresul[i]!=paraleloresul[i])
        {
                printf ("error %d %d %d\n",i,serieresul[i],paraleloresul[i]);
                return 1;
        }
        return 0;
}
```

offmic.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <float.h>
#include <limits.h>
#include <string.h>

#define NTHR 224
#define NBD  1000000
#define NPET 1000
#define K 160
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)
__declspec(align(64)) static float bd[NBD][K];
__declspec(align(64)) static int serieresul[NPET];
__declspec(align(64)) static int paraleloresul[NPET];
__declspec(align(64)) static float bd[NBD][K];
__declspec(target(mic:0)) __declspec(align(64)) static float
resultadosthreads[NTHR*64];
__declspec(target(mic:0)) __declspec(align(64)) static int indicethreads[NTHR*64];
__declspec(align(64)) static int serieresul[NPET];
__declspec(align(64)) static int paraleloresul[NPET];
__declspec(align(64)) float vector[K];
__declspec(align(64)) float vector1[K];

void leer_entrada(FILE *fptr, float *vector, const char *name, int indice, int n)
{
	int i, j;
	if(( fptr = fopen(name, "r")) == NULL)
	{
		printf("No es posible abrir el fichero");
	}
		fseek(fptr, indice+(indice * sizeof(float) * K), SEEK_SET);
	//PUNTERO DE ARCHIVO (+ind para el \n)
	//printf("ftell = %d \n", ftell(fptr));    //imprime posicion apuntada en
el fichero
		for(i=0;i<n;i++)     fscanf(fptr, "%f", &vector[i]);
	fclose(fptr);
}

void leer_BD(FILE *fptr, float bd[][K], const char *name){
	int i, j;
	if(( fptr = fopen(name, "r")) == NULL)
	{
		printf("No es posible abrir el fichero");
	}
	printf("Leyendo BD...\n");
		for(i=0;i<NBD;i++)
			for(j=0;j<K;j++)
				fscanf(fptr, "%f", &bd[i][j]);
	fclose(fptr);
}

void escribir(FILE *fptr, int rango, const char *name,int n){
	int i, j;
	double num;
	if(( fptr = fopen(name, "wt")) == NULL)
	{
		printf("No es posible abrir el fichero");
	}
```

```
	printf("Escribiendo fichero %s\n", name);
		for(i=0;i<n;i++)
		{
			for(j=0;j<K;j++){
				num = rand() % rango;
				fprintf(fptr, "%4.0f", num);
			}
			fprintf(fptr, "%s", "\n");
		}
	fclose(fptr);
}

__declspec(target(mic:0))  float calcular_distancia(float *vec1, float *vec2, int
nvector)	//pasar 160 y 160 elems y calc "DE"
{
	int i;
	float acum, distancia = 0;
	for (i=0; i<nvector; i++)
	{
		acum += (vec1[i]-vec2[i]) * (vec1[i]-vec2[i]);
	}
	distancia = sqrt(acum);
return distancia;
}

int comparar(float *vector, float bd[][K], int nbd)
{
	int i;
	float elMenor = FLT_MAX;
	float res;
	int menor = INT_MAX;
		for(i=0; i<nbd; i++)
		{
				res= calcular_distancia(vector, &bd[i][0], K);
				if (elMenor > res)
				{
				    elMenor = res;
				   menor = i;
				}
		}
	return menor;
}

__declspec(target(mic:0)) void parcomparar(float *vector, float bd[][K], int nbd,
float *elMenor, int *menor)
{
	int i, id;
	float localmenor;
	int localquien, desde, hasta;
	int trozo = ceil((float)nbd/(float)NTHR);
	id = omp_get_thread_num();
	desde= id*trozo;
	hasta = (((id+1)*trozo) < nbd)  ? (id+1)*trozo : nbd ;
	menor[id*64]=INT_MAX;
	elMenor[id*64]= FLT_MAX;
		for(i=desde; i<hasta; i++)
		{
			localmenor = calcular_distancia(vector, &bd[i][0], K);

			    if (elMenor[id*64] > localmenor)
			    {
			        elMenor[id*64] = localmenor;
			       menor[id*64] = i;
			    }
```

```
               }//cada thread ya tiene su menor, el host calcula el min de todos
los threads
}

void serie (int npet, FILE *fptr, const char *fname2, float *vector, float
bd[][K], int *resul, int nbd)
{
    int i, elmin, k;
       for(i=0;i<npet;i++)
       {
              leer_entrada(fptr, vector, fname2, i, K);
              elmin=comparar(vector, bd, nbd);
              resul[i] = elmin;
       }
}

int paralelo (int npet, FILE *fptr, const char *fname2, float *vector, float
*vector1, float bd[][K], int *resul, int nbd)
{
    int i, j, k;
    char tag, tag2, tag1;
    float min = FLT_MAX;
    int ind;
       #pragma offload_transfer target(mic:0) in(bd[0:nbd][0:K]: ALLOC)
       leer_entrada(fptr, vector, fname2, 0,K);
       printf("\n -HOST- Transferida la BBDD(comprobacion correcta)\n\n");
       for(i=0;i<npet;i+=2)
       {
              if(i == (npet-1))    break;
              #pragma offload target(mic:0) nocopy(bd: REUSE) in(vector:length(K))
inout(resultadosthreads, indicethreads) signal(&tag)
              {
                     #pragma omp parallel num_threads(NTHR)
                     {
                            parcomparar(vector, bd, nbd, resultadosthreads,
indicethreads);
                     }
              }
              leer_entrada(fptr, vector1, fname2, i+1, K);
              #pragma offload_wait target(mic:0) wait(&tag)
              min = FLT_MAX;
              for(j=0;j<NTHR;j++)
                     if(min > resultadosthreads[j*64]){
                            min = resultadosthreads[j*64];
                            ind = indicethreads[j*64];
                     }
              resul[i] = ind;
              #pragma offload target(mic:0) nocopy(bd: REUSE)
in(vector1:length(K)) inout(resultadosthreads, indicethreads) signal(&tag1)
              {
                     #pragma omp parallel num_threads(NTHR)
                     {
                            parcomparar(vector1, bd, nbd, resultadosthreads,
indicethreads);
                     }
              }
              if( i+2 < npet ) leer_entrada(fptr, vector, fname2, i+2, K);
              #pragma offload_wait target(mic:0) wait(&tag1)
              min = FLT_MAX;
              for(j=0;j<NTHR;j++)          //reduccion de menor de primeros i-1
elems
                     if(min > resultadosthreads[j*64]){
                            min = resultadosthreads[j*64];
                            ind = indicethreads[j*64];
```

```
                        }
                resul[i+1] = ind;
        }
                if((npet % 2) == 0)   return 0;
                #pragma offload target(mic:0) nocopy(bd: FREE) in(vector:length(K))
inout(resultadosthreads, indicethreads)
                {
                        #pragma omp parallel num_threads(NTHR)
                        {
                                parcomparar(vector, bd, nbd, resultadosthreads,
indicethreads);
                        }
                }
                        min = FLT_MAX;
                for(j=0;j<NTHR;j++){
                        if(min > resultadosthreads[j*64]){
                                min = resultadosthreads[j*64];
                                ind = indicethreads[j*64];
                        }
                }
                resul[npet-1] = ind;
        return 0;
}

int main(int argc, char *argv[])
{
        int i, j;
        FILE *fptr = NULL;
        FILE *fptr2 = NULL;
        char fname1[100];
        char fname2[100];
        double w1;
    strcpy (fname1,argv[1]);
    strcpy (fname2,argv[2]);
        escribir(fptr, 256, fname1,NBD);    // crear Base datos Mic
        leer_BD(fptr, bd, fname1);              // fichero bd en estructura de
datos bd
        for(i=0;i<3;i++){
                for(j=0;j<3;j++)
                        printf("%1.f ", bd[i][j]);
                printf("\n");
        }
        escribir(fptr, 256, fname2,NPET);          //fichero fuente de peticiones
        w1=omp_get_wtime();
                serie (NPET,fptr,fname2,vector,bd,serieresul,NBD);
        w1=(omp_get_wtime()-w1)*1000;
        printf("\nTiempo transcurrido serie: %f mseg \n", w1);
        w1=omp_get_wtime();
                paralelo(NPET,fptr,fname2,vector, vector1, bd,paraleloresul,NBD);
        w1=(omp_get_wtime()-w1)*1000;
        printf("\nTiempo transcurrido paralelo: %f mseg \n", w1);
    for (i=0;i<NPET;i++)
        if (serieresul[i] != paraleloresul[i])
        {
          printf ("error %d %d %d\n", i, serieresul[i], paraleloresul[i]);
          return 1;
        }
        return 0;
}
```