CE4302 – Arquitectura de Computadores II

Extensiones SIMD

PROFESOR: ING. LUIS BARBOZA ARTAVIA

Agenda

- Introducción.
- SIMD.
- Ejemplos.
- Modelo Roofline.

Variaciones de SIMD

- **Arquitecturas vectoriales.**
  - Ejecución por *pipeline* de muchas operaciones de datos.
- **Extensión SIMD.**
  - Habilitar la posibilidad de brindar operaciones de datos en paralelo.
- **Unidades de procesamiento gráfico.**
  - Ofrecer mayor rendimiento que en los multicore tradicionales.

Extensiones SIMD

- Surgieron por la observación de que las aplicaciones multimedia operaban con tipos de datos más reducidos que los procesadores de 32 bits tenían optimizadas.
- Por ejemplo:
  - 8 bits para cada color y 8 bits para la transparencia.
  - Las muestras de audio se suelen representar por 8 bits o 16 bits.
- En un procesador de propósito general con hardware específico que permite la ejecución de instrucciones ensamblador para el manejo de operaciones vectoriales.

SIMD más comunes

- x86:
  - MMX
  - SSE1, SSE2, SSE3, SSE4, SSE5.
  - AVX, AVX-512.
- ARM:
  - Neon
  - Vector Floating-Point (VFP)

SIMD hardware

- Podemos utilizar un sumador de 256 bits y así ejecutar de manera simultánea:
  - 32 operandos de 8 bits.
  - 16 operandos de 16 bits.
  - 8 operandos de 32 bits.
  - 4 operandos de 64 bits.
- La decodificación de instrucciones se vuelve más compleja.
  - Más instrucciones y más hardware.

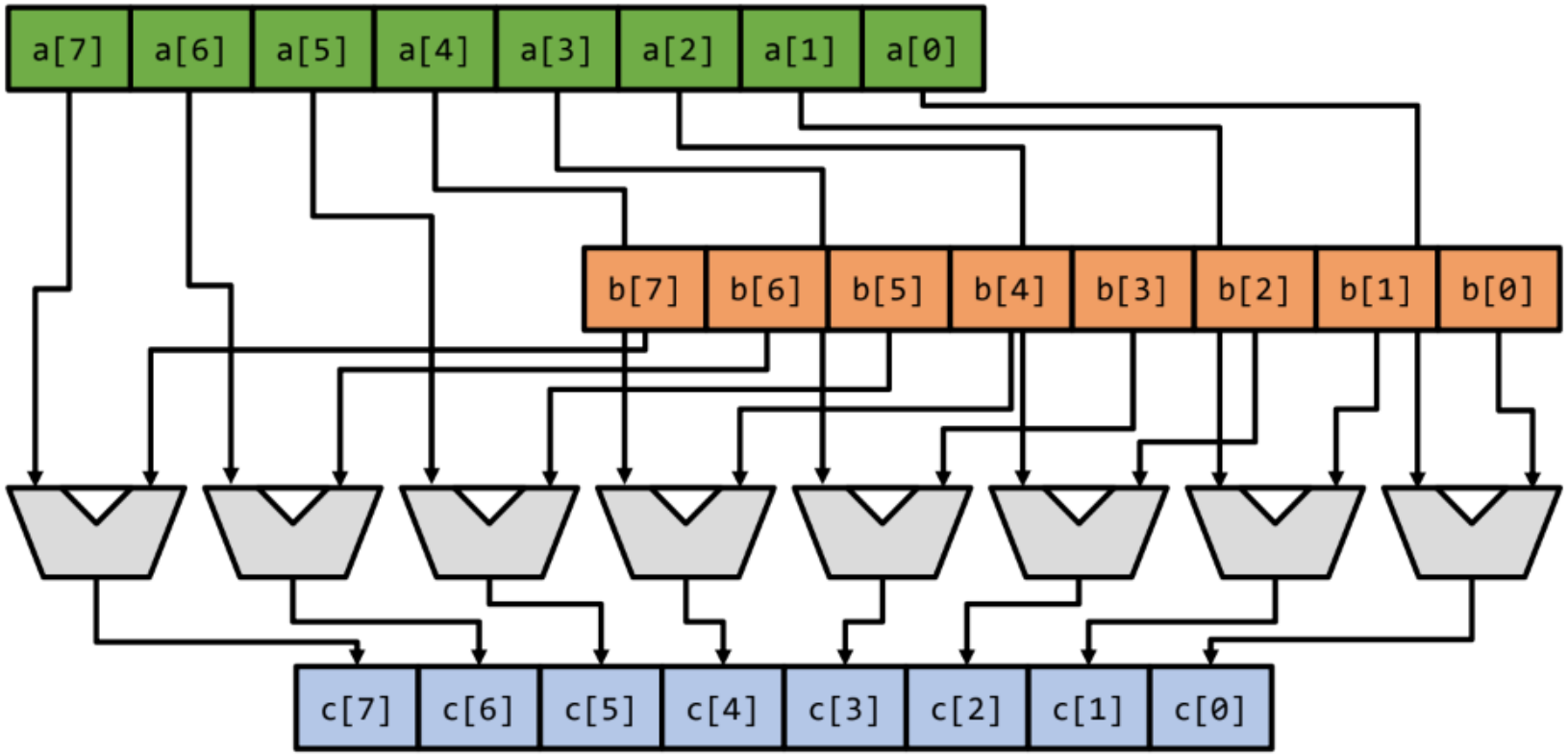
Instrucciones SIMD típicas

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

Extensiones SIMD

- Forma de procesar de forma vectorial en un CPU de propósito general.
- Propósito: obtener beneficios de rendimiento (ojalá energéticos).
- Mayor eficiencia en el procesamiento de ciertas aplicaciones: multimedia, reconocimiento de imágenes, operaciones 8-16 bit pixel.

Extensiones SIMD



SIMD vs Vector

- Al igual que en operaciones vectoriales, las instrucciones SIMD especifican la misma operación en los vectores de datos.
- SIMD utiliza menos operandos y los registros son de menor tamaño.
  - Arquitecturas vectoriales tienen registros de gran tamaño.
- Extensiones SIMD establecen el número de operandos en el código de operación.
- No ofrece registros máscara para soportar ejecución condicional.

SIMD vs Vector

SIMD	Vector
Operaciones producen datos para los elementos del vector en el mismo ciclo.	Operaciones producen datos para los elementos del vector en varios ciclos.
El tamaño del vector está “alambrado”, aumentarlo requiere más instrucciones.	Tamaño vector puede cambiar con la generación del procesador, sin requerir cambios en ISA.
Las operaciones vectoriales se utilizan para aumentar el rendimiento y, en general, aumentar una arquitectura escalar de alto rendimiento.	Mejor rendimiento cuando la mayor parte del código es vectorizable: la arquitectura escalar es relativamente simple

Ventajas extensiones SIMD

- Costo bajo para añadirlo a las unidades aritméticas.
- SIMD no requiere un alto uso de ancho de banda como el caso de las arquitecturas vectoriales (no todas las computadoras tienen ese ancho de banda).
- Facilidad para introducir instrucciones que ayuden las aplicaciones.
- Menos problemas con memoria caché.

Instrucciones AVX

AVX Instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

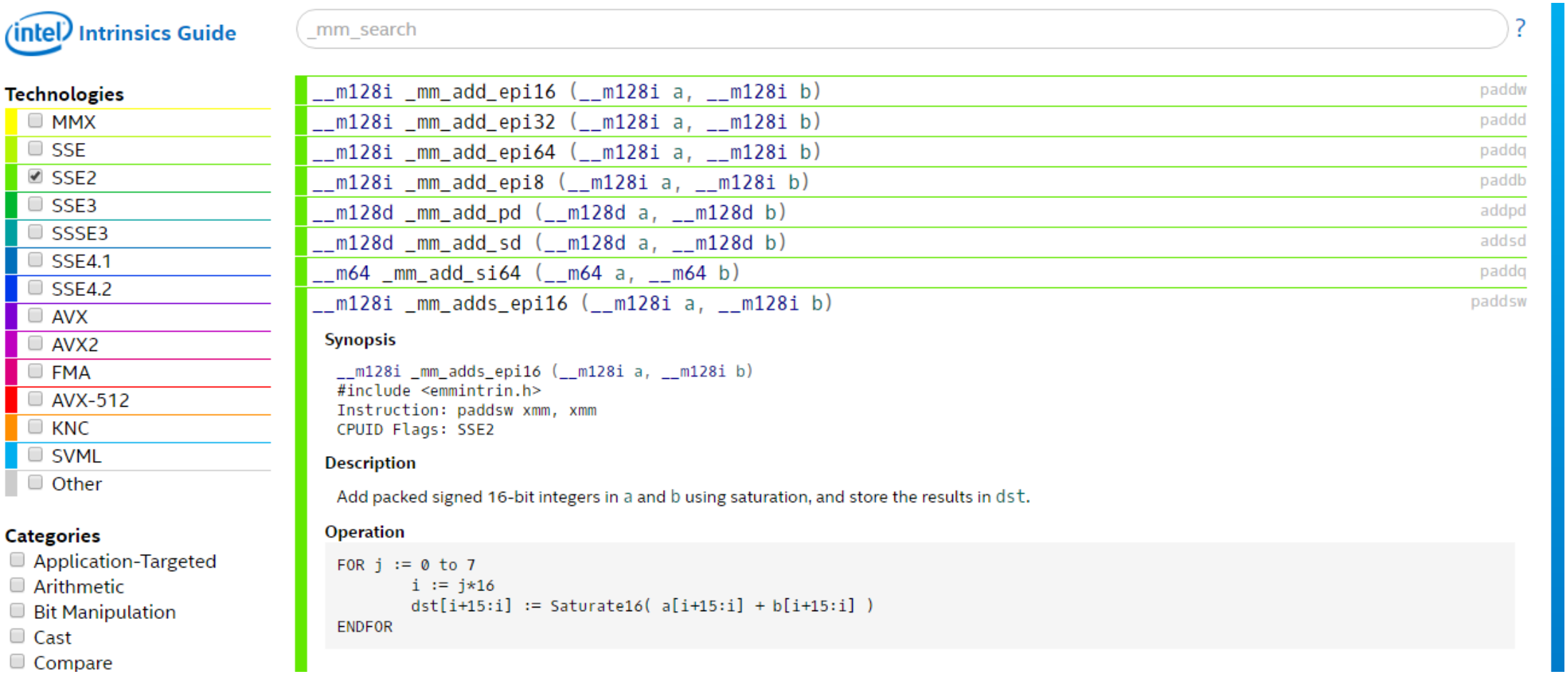
Instrucciones Neon

Mnemonic	Brief description
VABA, VABD	Absolute difference, Absolute difference and Accumulate
VABS	Absolute value
VACGE, VACGT	Absolute Compare Greater than or Equal, Greater Than
VACLE, VACLT	Absolute Compare Less than or Equal, Less Than (pseudo-instructions)
VADD	Add
VADDHN	Add, select High half
VAND	Bitwise AND
VAND	Bitwise AND (pseudo-instruction)
VBIC	Bitwise Bit Clear (register)
VBIC	Bitwise Bit Clear (immediate)
VBIF, VBIT, VBSL	Bitwise Insert if False, Insert if True, Select
VCEQ, VCLE, VCLT	Compare Equal, Less than or Equal, Compare Less Than

Intel Intrinsics

- Funciones codificadas en ensamblador que son llamadas por medio de estas instrucciones.
- Facilitan la generación de código ensamblador.

Intel Intrinsics



https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=173

Neon Intrinsics

- Llamadas del sistema que el compilador reemplaza con una instrucción Neon o un conjunto de instrucciones Neon [1].
- Permiten al desarrollador enfocarse en los algoritmos y Neon se encarga de la asignación de registros [1].
- Facilidad de mantenimiento del código fuente [1].

[1] https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics

Neon Intrinsics

`int8x8_t vadd_s8 (int8x8_t a, int8x8_t b)`	Add
`int8x16_t vaddq_s8 (int8x16_t a, int8x16_t b)`	Add
`int16x4_t vadd_s16 (int16x4_t a, int16x4_t b)`	Add
`int16x8_t vaddq_s16 (int16x8_t a, int16x8_t b)`	Add

Description

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

A64 Instruction

```
ADD Vd.8H,Vn.8H,Vm.8H
```

https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics

Código en ARM

```
for (i=0; i<8;i+=1)
{
   acc+=array[i]; // a + b + c + d + e + f + g + h
}
```

Loop unrolling

```
for (i=0; i<8;i+=4)
{
   acc1+=array[i];   // (a, e)
   acc2+=array[i+1]; // (b, f)
   acc3+=array[i+2]; // (c, g)
   acc4+=array[i+3]; // (d, h)
}
acc1+=acc2;          // (a+e) + (b+f)
acc3+=acc4;          // (c+g) + (d+h)
acc1+=acc3;          // ((a+e) + (b+f))+((c+g) + (d+h))
```

Código en ARM

```
uint32_t vector_add_of_n(uint32_t* ptr, uint32_t items)
{
    uint32_t result,* i;
    uint32x2_t vec64a, vec64b;
    uint32x4_t vec128 = vdupq_n_u32(0); // clear accumulators

    for (i=ptr; i<(ptr+items);i+=4)
    {
        uint32x4_t temp128 = vld1q_u32(i); // load 4x 32 bit values
        vec128=vaddq_u32(vec128, temp128); // add 128 bit vectors
    }

    vec64a = vget_low_u32(vec128); // split 128 bit vector
    vec64b = vget_high_u32(vec128); // into 2x 64 bit vectors

    vec64a = vadd_u32 (vec64a, vec64b); // add 64 bit vectors together

    result = vget_lane_u32(vec64a, 0); // extract lanes and
    result += vget_lane_u32(vec64a, 1); // add together scalars

    return result;
}
```

Pasos para desarrollar algoritmo

- Preparación datos (tipo, memoria, formato).
- Operación.
- Guardado (conversión).

Ejecutando primer programa

https://github.com/kshitijl/avx2-examples/blob/master/examples/01-hello.c

Modelo Roofline

- Forma intuitiva de comparar el rendimiento potencial de operaciones en punto flotante en variaciones de arquitecturas SIMD.
- En un gráfico de dos dimensiones se muestra el rendimiento de operaciones de punto flotante (**FLOPS**), memoria e **intensidad aritmética**.

Modelo Roofline

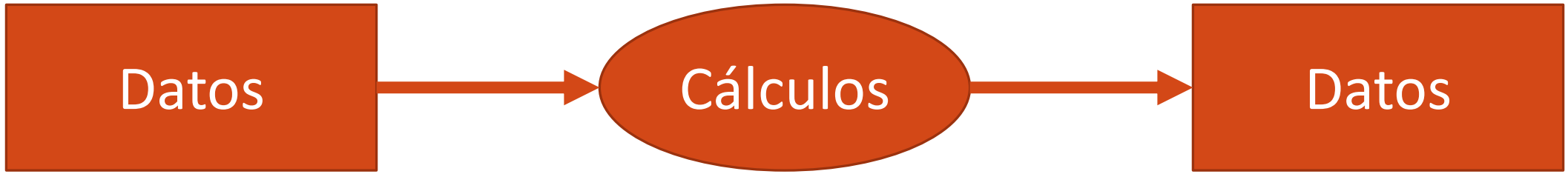
- “…Al combinar localidad, **ancho de banda** y diferentes paradigmas de paralelización en una única cifra de rendimiento, el modelo puede ser una alternativa efectiva para **evaluar la calidad del rendimiento obtenido** en lugar de utilizar simples estimaciones de porcentaje de pico, ya que proporciona información tanto sobre la implementación como las limitaciones de rendimiento inherentes..”
- Utiliza escala logarítmica.

Intensidad aritmética

- Es la relación entre operaciones de punto flotante y bytes de memoria accedidos.
- Se calcula de la siguiente manera:

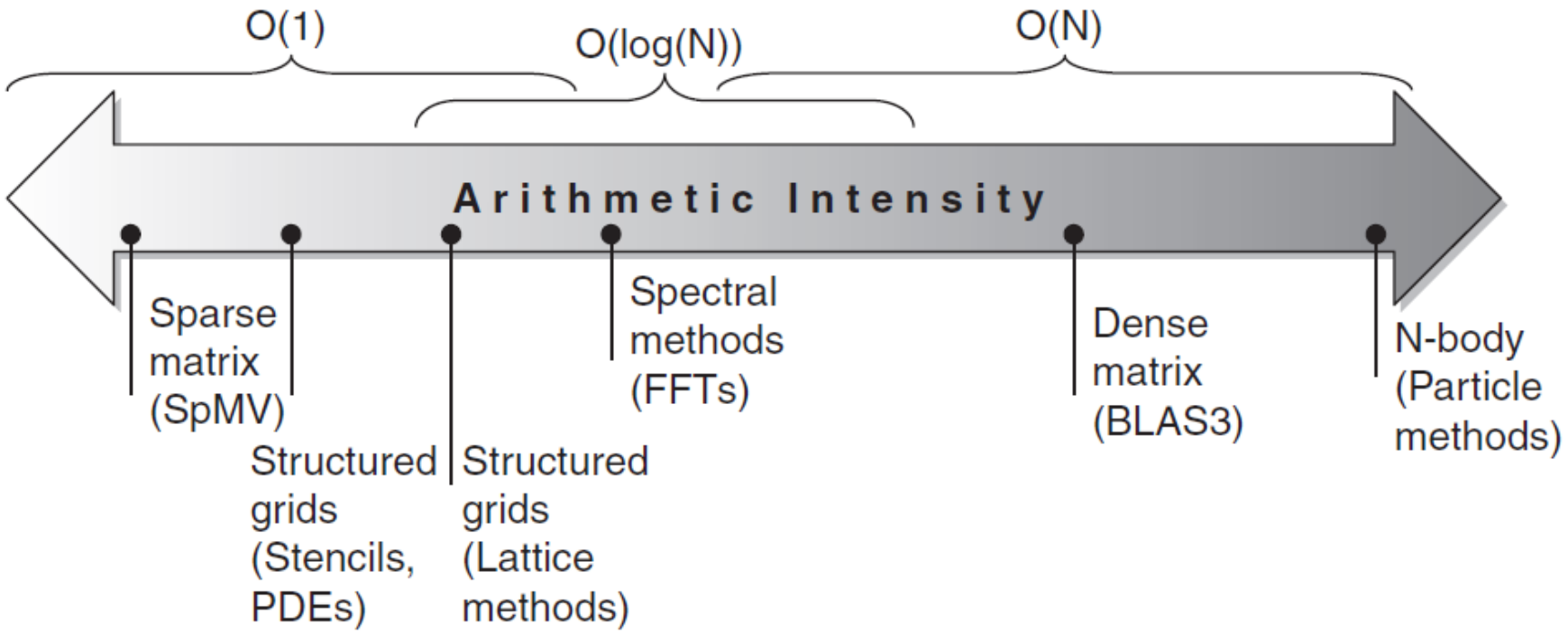
$$A_i = \frac{Total\ op\ FP}{Total\ bytes\ transferred}$$

Intensidad aritmética



Intensidad aritmética

- La intensidad aritmética puede ser constante $O(1)$ o aumentar con el tamaño del problema $O(\log(N))$, $O(N)$



Ejemplo Intensidad aritmética

Calcule la intensidad aritmética para el siguiente código:

```
for (i=0;i<300;i++) {
        c_re[i] = a_re[i] * b_re[i] – a_im[i] * b_im[i];
        c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
}
```

$$A_i = \frac{Total\ op\ FP}{Total\ bytes\ transferred} = \frac{Total\ op\ FP}{bytes_r + bytes_w} = \frac{300 * 6}{300 * (4 * 4bytes + 2\ * 4bytes)}$$

$$= \frac{300 * 6}{300 * 4 * 6} = \frac{1}{4}$$

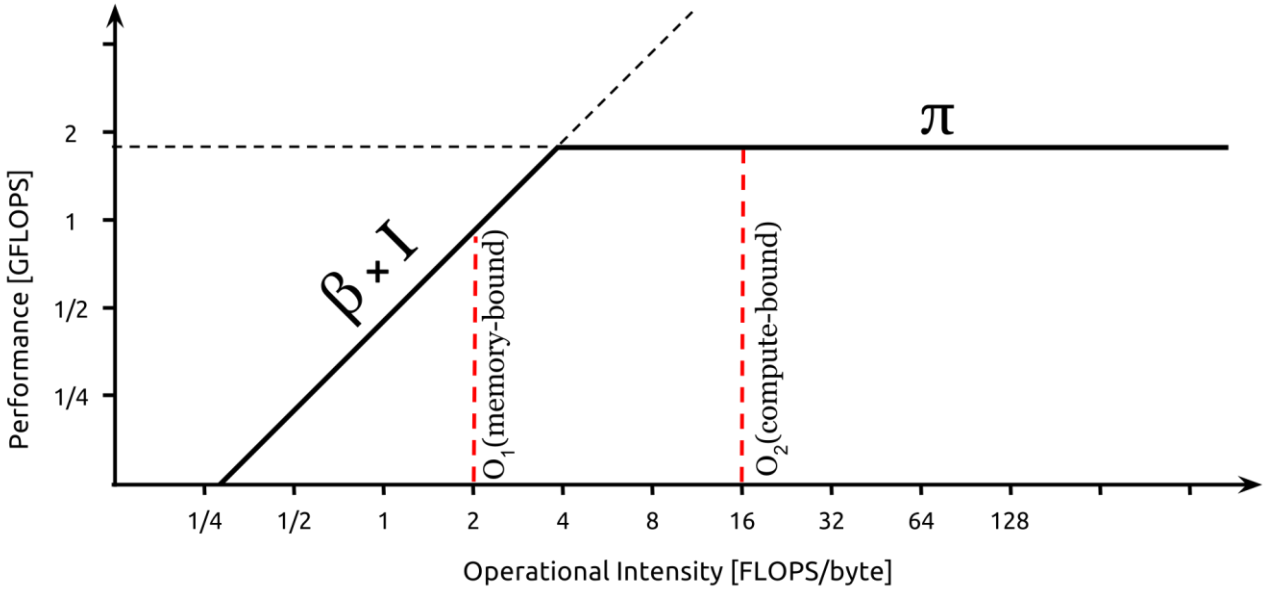
Ejemplo Intensidad aritmética

El siguiente código forma parte del método para el cálculo de las ecuaciones de Maxwell en 3D, en el benchmark SPEC06fp. Determine su intensidad aritmética :

```
for (int x=0; x<NX-1; x++) {
 for (int y=0; y<NY-1; y++) {
 for (int z=0; z<NZ-1; z++) {
 int index = x*NY*NZ + y*NZ + z;
 if (y>0 && x >0) {
 material = IDx[index];
 dH1 = (Hz[index] – Hz[index-incrementY])/dy[y];
 dH2 = (Hy[index] – Hy[index-incrementZ])/dz[z];
 Ex[index] = Ca[material]*Ex[index]+Cb[material]*(dH2-dH1);
}}}}
```

- Asuma que dH1, dh2, Hy, Hz, dy, dz, Ca, Cb y Ex son arreglos de número de punto flotante en precisión simple (IEEE754) y IDx es un arreglo de enteros sin signo.

Modelo Roofline

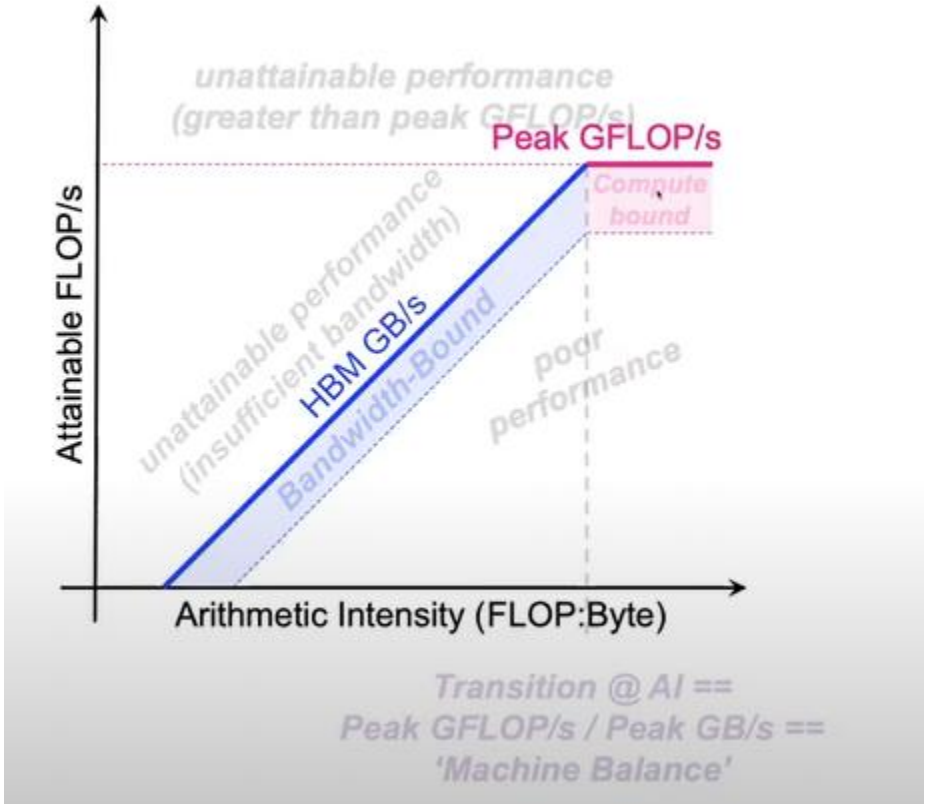


2 parámetros: peak performance **π**, peak bandwidth **β**
1 variable: arithmetic intensity I

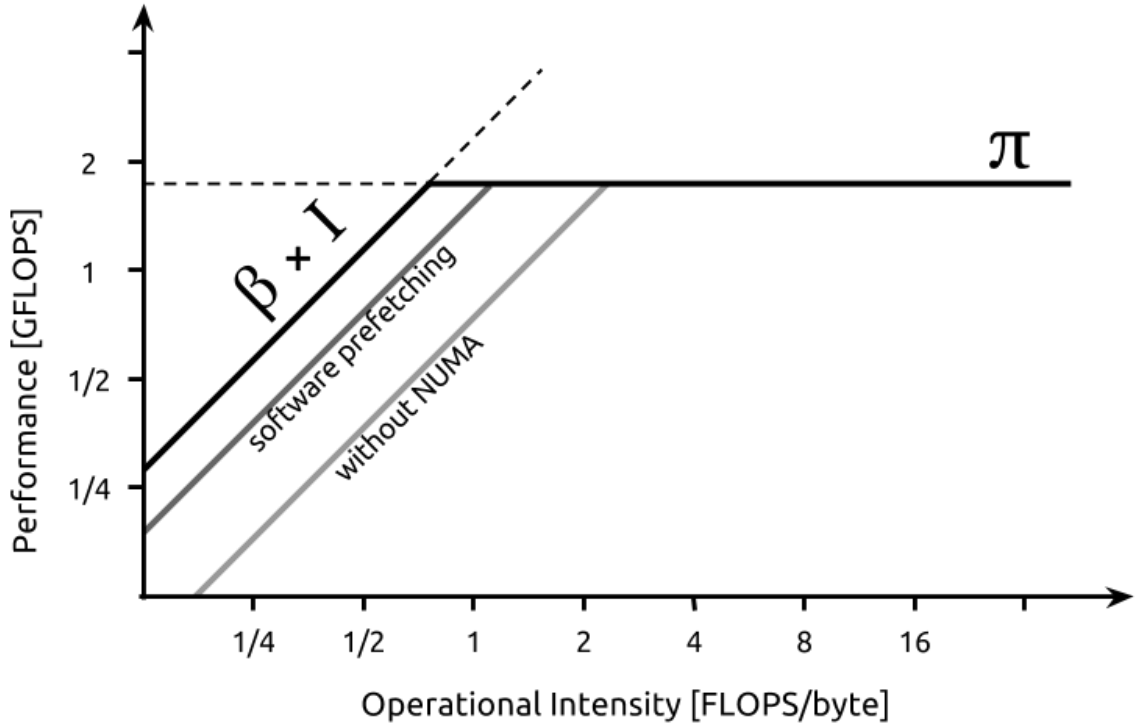
$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

P: Rendimiento alcanzable

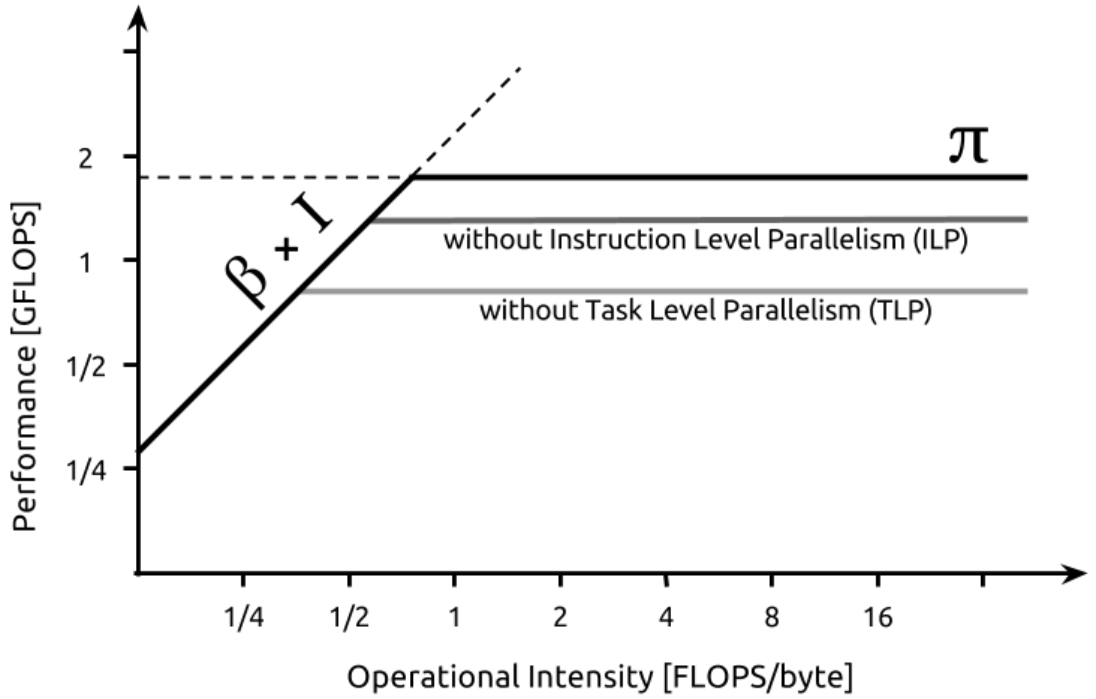
Modelo Roofline



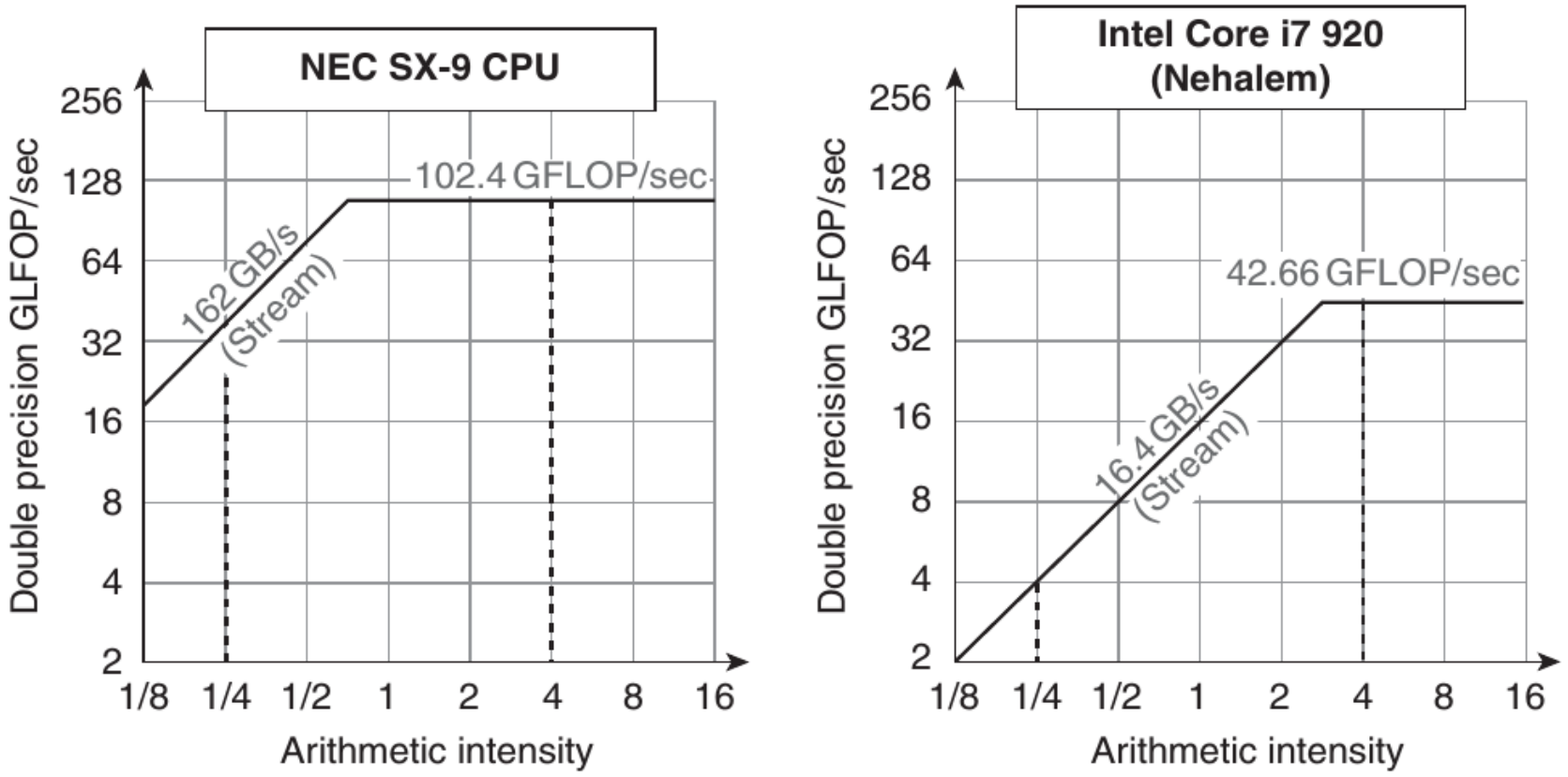
Modelo Roofline - Implementación



Modelo Roofline - Implementación



Modelo Roofline



Ejemplos

2- Usando los *Roof-line Models* de las implementaciones **X** y **Y** mostrado en las figuras 1 y 2 determine:

a- Identifique las zonas y calcule los parámetros relevantes del Roof-line Model de la figura 1. (π, β$x$$I$, etc) (5 puntos)

b- Intensidad aritmética (I) de los siguientes extractos, suponga que constantes **<u>son obtenidas durante tiempo de compilación.</u>** (10 puntos)

c- En qué zona de operación se encuentran cada uno de los extractos en cada uno de los modelos, cuál de las implementaciones presenta mejor desempeño (Justifique con cálculos, argumentos de costo de implementación) (10 puntos)

```
1   const int WINDOW = 8192;
2   float a[WINDOW], b[WINDOW], c[WINDOW], d[WINDOW];
3
4   for(int i = 0; i < WINDOW; i++) {
5     const float tmp_a = a[i];
6     const float tmp_b = b[i];
7     c[i] += tmp_a*tmp_b;
8     d[i] = tmp_a+tmp_b;
9   }
```

extracto 2

```
1   const int WINDOW = 8192;
2   float a[WINDOW];//random data
3   float b[WINDOW];//random data
4   float c[WINDOW];//random data
5   float d[WINDOW];//random data
6   float s[WINDOW];//random data
7   float k[WINDOW];//random data
8   int index[WINDOW/2];
9
10  for(int i = 0; i < sizeof(index)/sizeof(int); i+=2) {
11      if((i % 4) == 0) {
12        index[i]++;
13      }
14
15      s[index[i]] = (a[index[i]] + b[index[i]] + c[index[i]] + d[index[i]])/(4);
16      k[index[i]] = -1.0 * a[index[i]];
17  }
```

extracto 3

Ejemplos

c- En qué zona de operación se encuentran cada uno de los extractos en cada uno de los modelos, cuál de las implementaciones presenta mejor desempeño (Justifique con cálculos, argumentos de costo de implementación) (10 puntos)

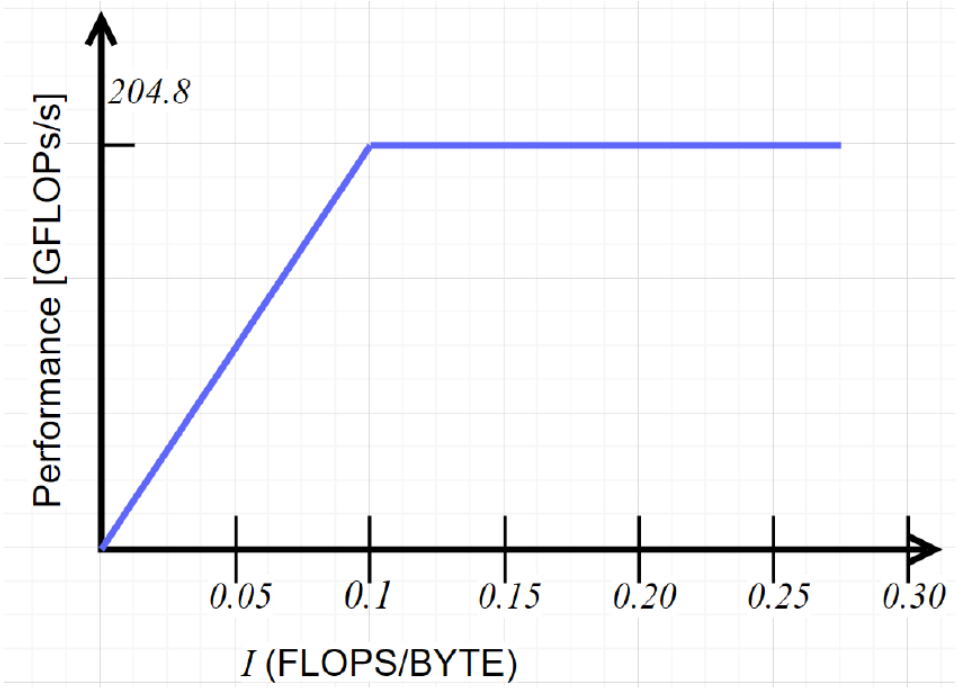


Figura 1. Roof-line Model para una implementación X.

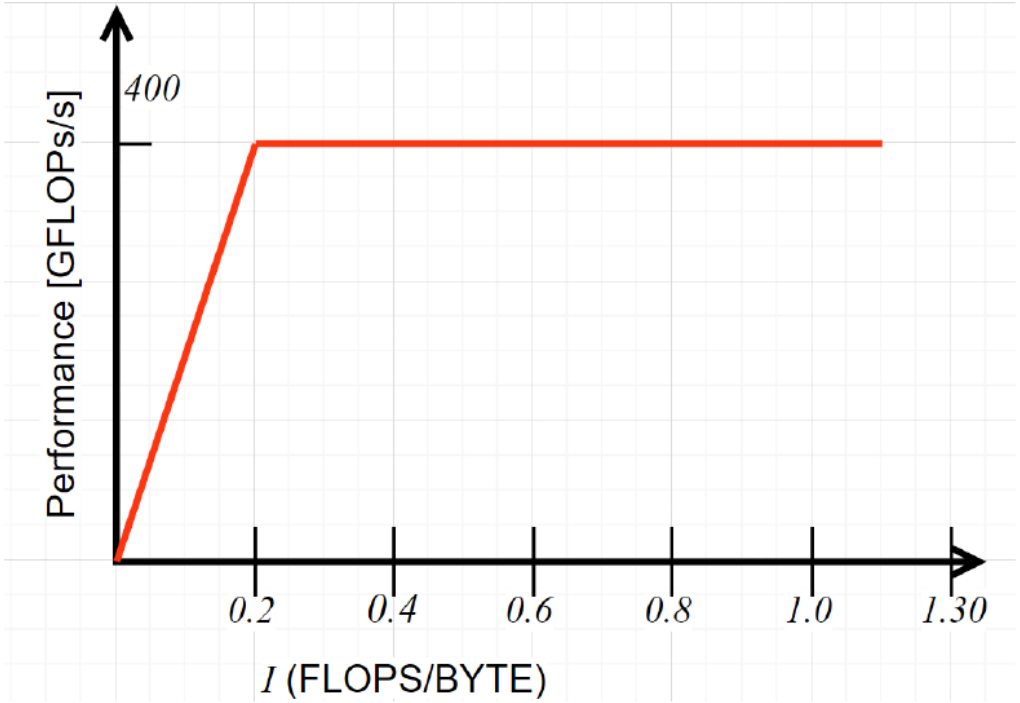


Figura 2. Roof-line Model para una implementación Y.

Ejemplos

Identifique las zonas y calcule los parámetros relevantes del Roof-line Model de la figura 1. (π, βxI, etc) (5 puntos)

$\pi =$

$\beta =$

Perf inalcanzable > π

Perf inalcanzable
no hay suf BW

BW bound

Compute bound

Perf pobre

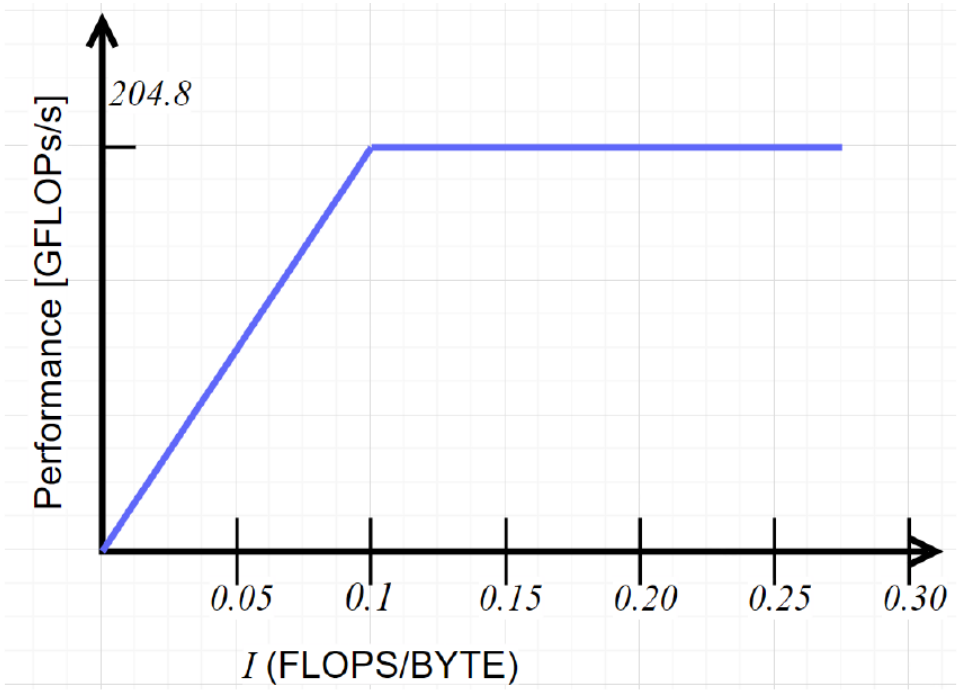


Figura 1. Roof-line Model para una implementación X.

Ejemplos

Identifique las zonas y calcule los parámetros relevantes del Roof-line Model de la figura 1. (π, βxI, etc) (5 puntos)

$\pi =$

$\beta =$

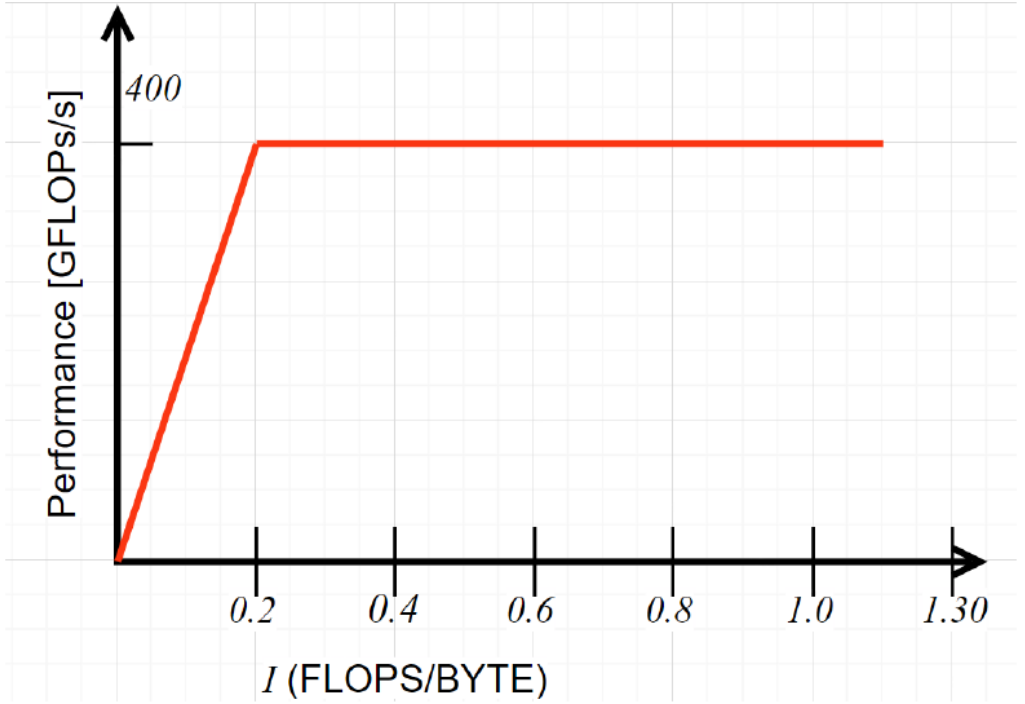


Figura 2. Roof-line Model para una implementación Y.

Perf inalcanzable > π

Perf inalcanzable
no hay suf BW

BW bound

Compute bound

Perf pobre

Ejemplos

Intensidad aritmética (I) de los siguientes extractos, suponga que constantes **son obtenidas durante tiempo de compilación.**

```
1   const int WINDOW = 8192;
2   float a[WINDOW], b[WINDOW], c[WINDOW], d[WINDOW];
3
4   for(int i = 0; i < WINDOW; i++) {
5     const float tmp_a = a[i];
6     const float tmp_b = b[i];
7     c[i] += tmp_a*tmp_b;
8     d[i] = tmp_a+tmp_b;
9   }
```

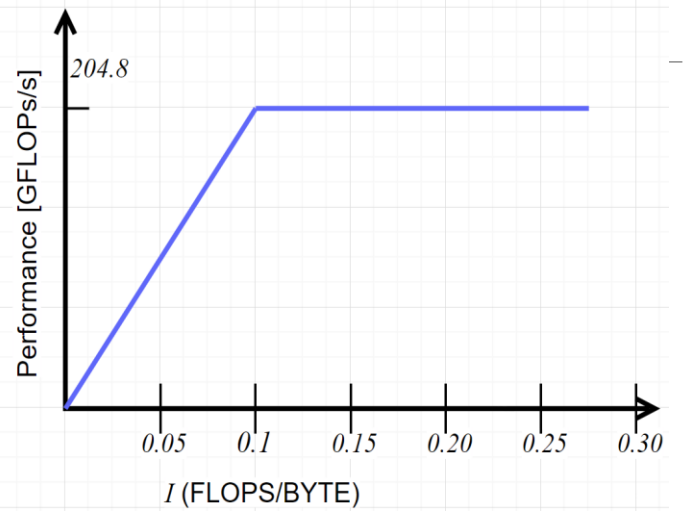
Ejemplos

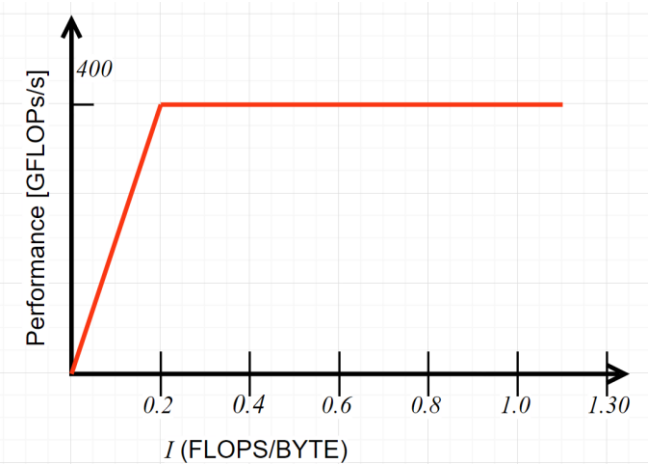
Intensidad aritmética (I) de los siguientes extractos, suponga que constantes **son obtenidas durante tiempo de compilación.**

```
1   const int WINDOW = 8192;
2   float a[WINDOW];//random data
3   float b[WINDOW];//random data
4   float c[WINDOW];//random data
5   float d[WINDOW];//random data
6   float s[WINDOW];//random data
7   float k[WINDOW];//random data
8   int index[WINDOW/2];
9
10  for(int i = 0; i < sizeof(index)/sizeof(int); i+=2) {
11      if((i % 4) == 0) {
12          index[i]++;
13      }
14
15      s[index[i]] = (a[index[i]] + b[index[i]] + c[index[i]] + d[index[i]])/(4);
16      k[index[i]] = -1.0 * a[index[i]];
17  }
```

Ejemplos

En qué zona de operación se encuentran cada uno de los extractos en cada uno de los modelos, cuál de las implementaciones presenta mejor desempeño (Justifique con cálculos, argumentos de costo de implementación)

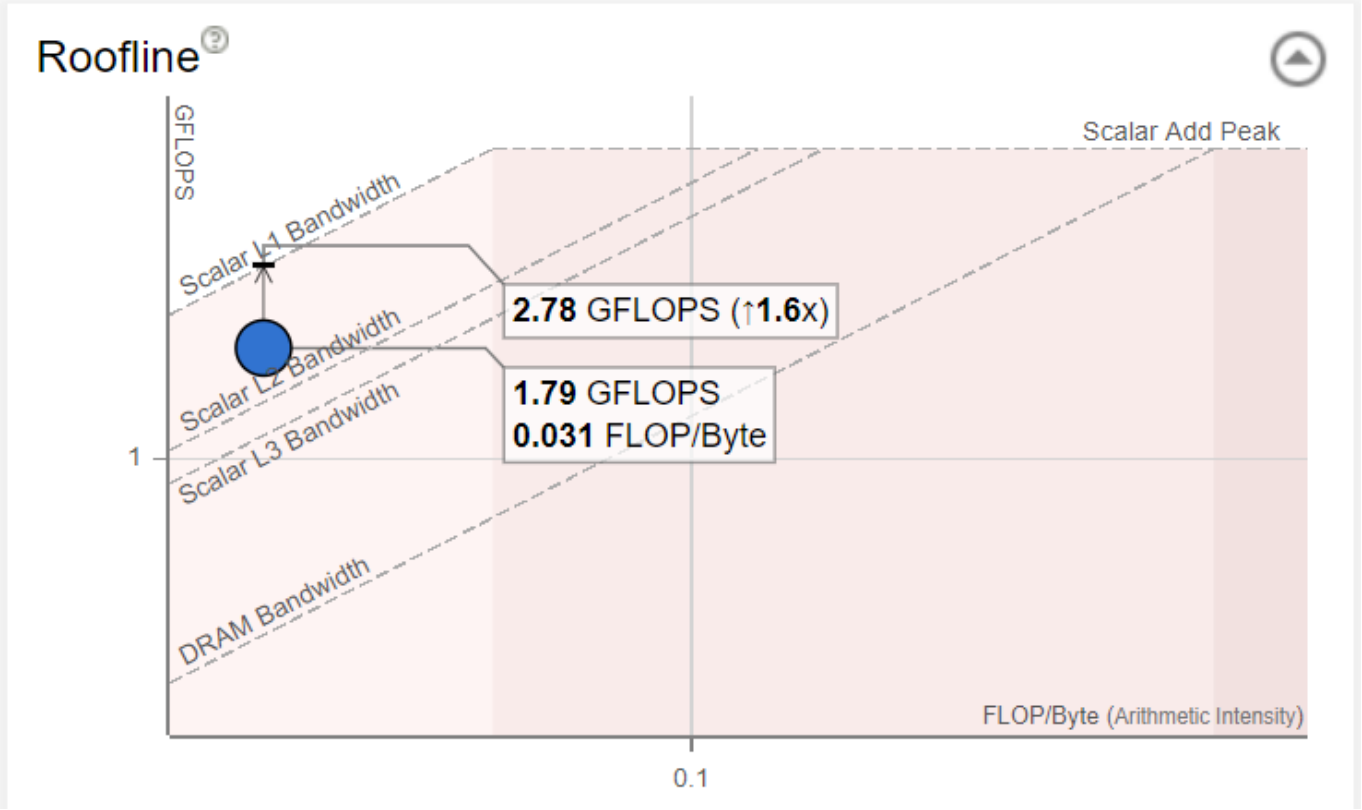




Ejemplos

En la figura 1 se muestra un *compute kernel* para ser ejecutado en una implementación de una arquitectura X, además se muestra el *roof-line model* e intensidad aritmética obtenidos experimentalmente

```
1   const int N = 10000000;
2   int main() {
3
4       float* a = new float[N];
5       float* b = new float[N];
6       float* x = new float[N];
7       float* y = new float[N];
8
9
10
11      for (int i = 0; i < N; ++i) {
12          y[i] = (a[i] + b[i]) / 2.0f;
13          x[i] = (y[i] * a[i]) - x[i];
14      }
15
16
17      delete[] a;
18      delete[] b;
19      delete[] x;
20      delete[] y;
21
22
23      return 0;
24  }
```



Ejemplos

Respecto a las figuras anteriores se le solicita:

a- Calcule la intensidad aritmética ideal (I_i) del código presentado en la figura 1. (5 puntos)

b- Realice un análisis comparativo del *compute kernel* bajo las condiciones ideales y los resultados experimentales, incluya en su análisis la zona en la que se ubican en el modelo mostrado en la figura 2, considerando un *Scalar L1 Bandwidth* de 299.23GB/s con un *peak performance* (***Scalar Add peak***) de 5.16GFLOPS (10 puntos)

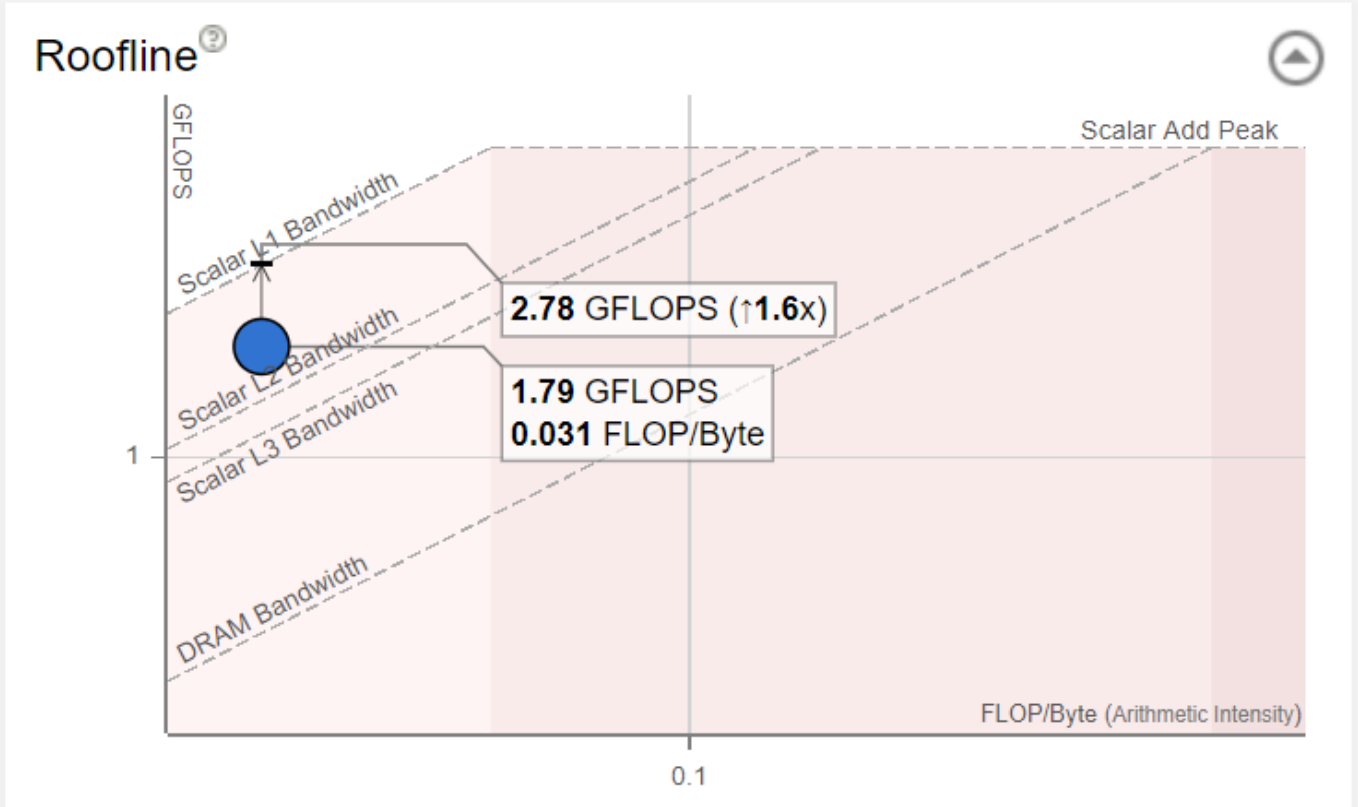
c- Basándose en el análisis ideal, cómo podría mejorar el código de forma tal que se obtenga un aumento de intensidad aritmética 20% a la calculada en el punto **a** (10 puntos)

d- Para el caso intensidad aritmética experimental (I_e) al emplear la herramienta Intel® Advisor la primer recomendación para mejorar el desempeño fue implementar la vectorización del código (*SIMD/intrinsics*) obteniendo I_e=0,25FLOP/byte y un nuevo *peak performance* (***Vector Add peak***) de 16.44GFLOPS .
Basándose en la definición de intensidad aritmética y seudocódigo, justifique como la vectorización de código puede influir potencialmente en mejorar el desempeño (sugerencia use el código de la figura 1 de referencia). (15 puntos)

Ejemplos

```
1   const int N = 10000000;
2   int main() {
3
4       float* a = new float[N];
5       float* b = new float[N];
6       float* x = new float[N];
7       float* y = new float[N];
8
9
10
11      for (int i = 0; i < N; ++i) {
12          y[i] = (a[i] + b[i]) / 2.0f;
13          x[i] = (y[i] * a[i]) - x[i];
14      }
15
16
17      delete[] a;
18      delete[] b;
19      delete[] x;
20      delete[] y;
21
22
23      return 0;
24  }
```

Ejemplos


Roofline
GFLOPS
Scalar Add Peak
Scalar L1 Bandwidth
Scalar L2 Bandwidth
Scalar L3 Bandwidth
DRAM Bandwidth
2.78 GFLOPS (↑1.6x)
1.79 GFLOPS
0.031 FLOP/Byte
1
0.1
FLOP/Byte (Arithmetic Intensity)

Referencias

- Stallings, W. (2003). Computer organization and architecture: designing for performance. Pearson Education India.
- Hennessy, J., & Patterson, D. (2012). Computer Architecture: A Quantitative Approach (5th ed.). Elsevier Science.

CE4302 – Arquitectura de Computadores II

Extensiones SIMD

PROFESOR: ING. LUIS BARBOZA ARTAVIA