❖Tema 4.2 DLP. Extensiones SIMD, GPU's y Vectorización.

➢ 4.2.1 Extensiones SIMD

Las extensiones multimedia se usan para ejecutar operaciones escalares "cuasi" convertidas en operaciones vectoriales.

Supongamos que queremos realizar la operación vectorial $\vec{Y} = a * \vec{X} + \vec{Y}$ que aprendimos a resolver con el computador VMIPS. Los vectores tienen 64 componentes, pero en vez de utilizar una arquitectura con instrucciones vectoriales realizamos los cálculos en "paquetes multimedia" de 4 componentes.

```
        L.D F0,a              ;load scalar a
        MOV F1, F0            ;copy a into F1 for SIMD MUL
        MOV F2, F0            ;copy a into F2 for SIMD MUL
        MOV F3, F0            ;copy a into F3 for SIMD MUL
        DADDIU R4,Rx,#512     ;last address to load  64 elementos
Loop:   L.4D F4,0(Rx)         ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D F4,F4,F0       ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D F8,0(Ry)         ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D F8,F8,F4       ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D F8,0(Ry)         ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU Rx,Rx,#32      ;increment index to X
        DADDIU Ry,Ry,#32      ;increment index to Y
        DSUBU R20,R4,Rx       ;compute bound
        BNEZ R20,Loop         ;check if done
```

Las instrucciones de extensión multimedia son las indicadas en color más claro. Al escribir

L.4D F4, 0(Rx)

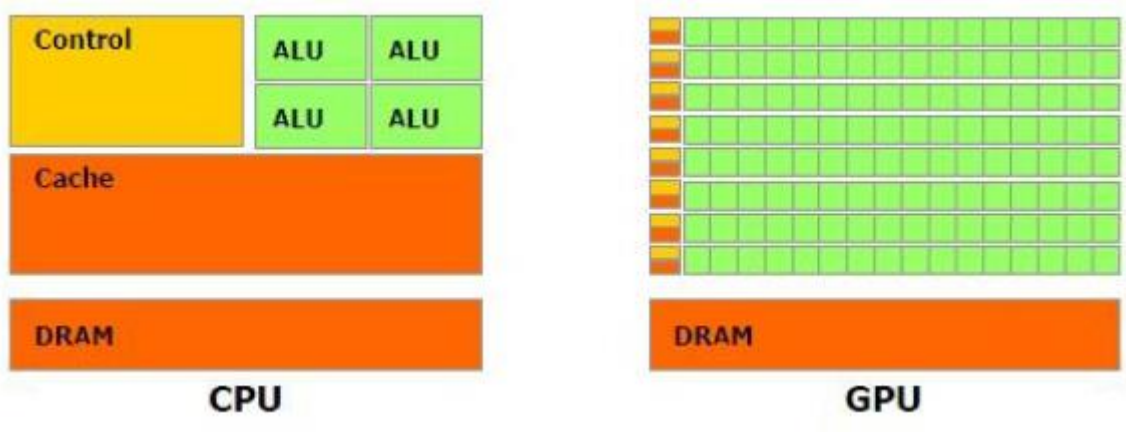
Estamos diciendo que la operación de Load se hace con la dirección de Rx, pero se carga 4 veces (4 valores consecutivos) sobre F4, F5, F6 y F7.

L.4D F8, 0(Ry)

Carga 4 componentes de Y sobre los registros F8, F9, F10 y F11.

➢ 4.2.2 Unidades para Procesamiento Gráfico (GPU)

Las GPUs son pequeños microprocesadores ideados para realizar carga de procesado de imágenes en paralelo con la CPU. Casi toda su estructura está compuesta de unidades de cálculo, son más pequeñas que las ALU de un procesador central, pero son mucho más numerosas.



Casi todo el área de una GPU está ocupada por unidades de procesado de datos. Esto nos permite procesar imágenes (que son mapas de píxeles) de una forma muy paralelizable.

¿Y si utilizamos el poder paralelizable de las GPU para realizar operaciones redundantes de la CPU que no tengan que ver con las imágenes?

Hemos visto en este mismo tema que nos encontramos con cierta frecuencia en la situación de realizar una misma operación repetida sobre una gran cantidad de datos (DLP). Las GPU son económicas, no consumen demasiada potencia y pueden complementar el trabajo de las CPU.

▪ Modelo de programación escalable. NVIDIA.

Las GPU de NVIDIA son tarjetas que funcionan bajo un lenguaje de alto nivel denominado CUDA (***C**ompute **U**nified **D**evice **A**rchitecture)*. Nos da un interfaz muy similar a C para programar aplicaciones que combinen el uso de CPU + GPU.

__host__
Se ejecuta en el Host (CPU) y se llama desde el Host. Si una función no tuviera distintivo, por defecto es host.

__device__
Función que se ejecuta en el dispositivo y sólo puede ser llamada por el dispositivo. (GPU)

__global__
Es el distintivo de kernel, se ejecuta en el dispositivo, pero la llama el Host. Siempre debe ser una función void.

Por ejemplo:

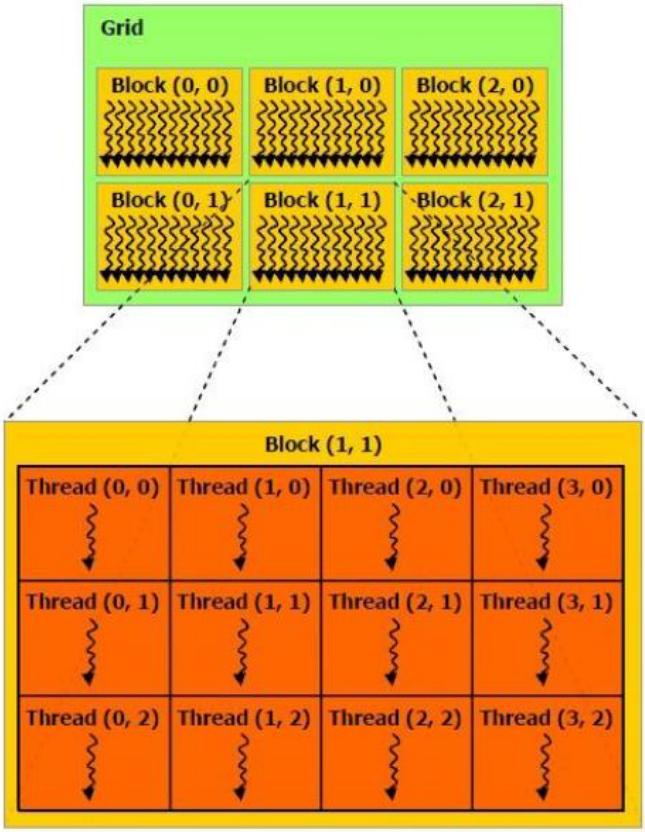
```
__global__
void Kernel_1 (int *F, int *I){ }

__device__
int minimo(int *A,int tam){ }
```

No nos vamos a extender mucho en la terminología del lenguaje CUDA, que es un lenguaje con ciertas restricciones de bajo nivel, pero hablaremos del tratamiento de los hilos.

Para aprovechar el paralelismo de las GPU se utiliza la política SIMT (***S**ingle **I**nstruction **M**ultiple **T**hreads*), que quiere decir que para cada componente de operación se usa un hilo diferente.

- Cada hilo se ejecuta sobre una componente del vector.
- Los hilos se agrupan por bloques.
- Los bloques se lanzan a ejecución a la vez, con todos sus hilos en paralelo. El conjunto de todos los bloques se denomina Grid (parrilla).



Cada bloque tiene unas componentes numéricas que lo identifican y diferencian de los demás en la parrilla. Cada hilo dentro de un bloque tiene a su vez unos índices que lo diferencian de sus hermanos dentro de un mismo bloque.

Las coordenadas que admite CUDA para el Grid y para los bloques son 3 dimensiones.

- **threadIdx.x . threadIdx.y . threadIdx.z** . Indican las coordenadas de un hilo dentro del bloque. Van desde cero hasta (blockDim.x – 1 , blockDim.y – 1 , blockDim.z – 1).

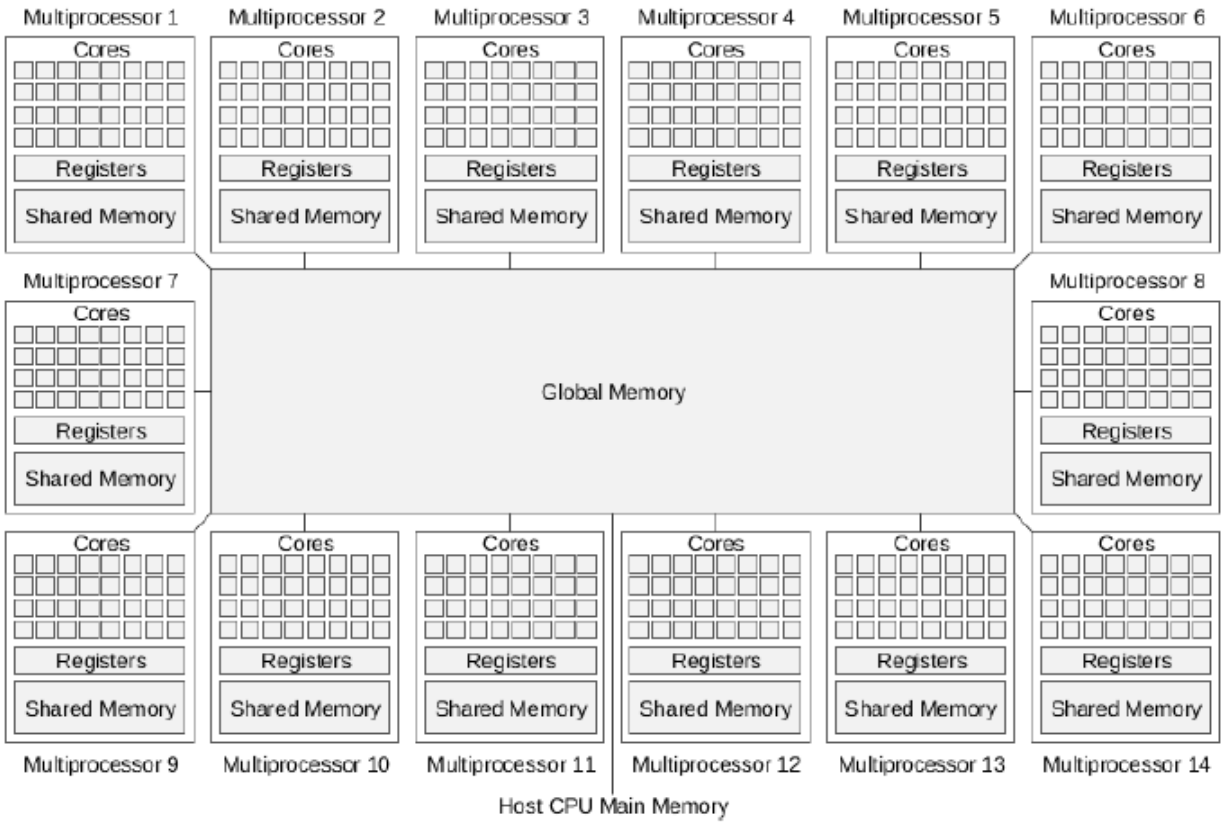
- **blockIdx.x. blockIdx.y. blockIdx.z .** Al igual que para los hilos, estos índices sitúan a los bloques dentro del Grid. Van desde cero hasta (gridDim – 1).

La forma de decirle al dispositivo cómo queremos configurar el Kernel es la siguiente:

```
dim3 dimGrid(100, 50, 1); // 5000 CUDA Blocks
dim3 dimBlock(4, 8, 8);   // 256 Threads por Block
Mi_Kernel <<< dimGrid, dimBlock >>> (int n, double a, double *x, double *y)
```

▪ Multiprocesadores de GPU y WARPs

Cuando se ejecuta una función en la GPU se van lanzando los bloques a ejecución de forma secuencial. El paralelismo se consigue a 2 niveles diferentes , nivel de bloque de forma que se opera con los hilos dentro de un mismo bloque en un Multiprocesador (MT) agrupado en la GPU y a nivel de multiprocesador porque suele tener un buen número de ellos.



Un MT ejecuta sus hilos de forma interna. Al recibir un bloque (o varios) produce conjuntos de hilos de 32 en 32 llamados "warps" aunque el bloque tenga miles de hilos. El warp es un paquete de hilos que es la unidad mayor de procesamiento paralelo de un MT. Esto quiere decir que un MT puede procesar a la vez hasta 32 hilos. Sin embargo una GPU puede tener un buen número de MTs de manera que puede alcanzar tasas de paralelismo de x512 ó mayores.

Cuando un MT recibe uno o varios bloques los particiona en warps. Estos warps serán organizados y lanzados a ejecución por el "*warp scheduler*". Los bloques se particionan siempre de la misma manera, se cogen los 32 primeros hilos en orden del bloque que tenga a su vez menor orden dentro de los lanzados en el MT. El orden que se le otorga a cada hilo y a cada bloque se calcula mediante una sencilla fórmula.

De esta forma podemos decir que en un mismo MT pueden coexistir diferentes warps de diferentes bloques. La limitación del tamaño de warps que puede gestionar un MT depende de los recursos que usen los hilos (no puede ser mayor de 32, pero sí menor), cuantos más registros y más memoria por hilo, menos pueden coexistir. Por lo tanto un único bloque no podrá exceder estos requisitos ya que es imperativo que todos los hilos de un bloque estén en el mismo SM, los bloques no se reparten.

Un warp puede ejecutar una única instrucción al mismo tiempo, para maximizar la eficiencia del warp sus hilos deben tener las mismas instrucciones y el mismo camino, sólo una diferencia de accesos a memoria (eso es la estructura SIMT). Si los hilos no presentan divergencias, el warp realmente estará ejecutando 32 instrucciones paralelamente en los 32 "cores". Sin embargo si hay dependencias de datos o saltos condicionales se crearán diferentes caminos dentro del warp, éste ejecutará cada camino de forma secuencial poniendo en espera a todos los hilos que quedasen fuere del camino en concreto, perdiendo paralelismo.

Instrucción PTX (Parallel Thread Execution) **:** una instrucción PTX es aquella que se ejecuta en cada ciclo de warp, es exactamente la misma para todos los hilos, pero con la diferencia de que cada hilo coge sus datos de un registro diferente.

Warp scheduler(planificador de warps) : selecciona un warp para ejecutar cuando todos sus hilos están listos. En el momento en que los recursos se liberan, se lanza y ocupa los cores del MT hasta que termina su ejecución. Los MT pueden tener uno o dos warp schedulers. Existen diferentes configuraciones en función de la versión de hardware conocidas como "Compute Capability".

➢ 4.2.3 Vectorización de bucles.

Al margen de las implementaciones arquitectónicas para aprovechar el DLP, tenemos este pequeño apartado que nos habla de la posibilidad de convertir bucles de operaciones con dependencias LDE en un problema vectorizable (y por tanto trasladable a la arquitectura VMIPS). Existen dos tipos de dependencias:

Dependencia directa:

```
for (i=999; i>=0; i=i-1){
        x[i] = x[i] + s;        /* S1 */
        z[i] = z[i] + x[i];     /* S2 */
}
```

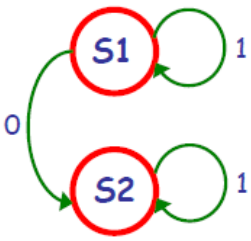
Dependencia en el espacio de iteraciones:

```
for (i=0; i<100; i=i+1) {
    A[i] = A[i] + B[i];       /* S1 */
    B[i+1] = C[i] + D[i];     /* S2 */
}
```

En la dependencia directa existen LDE dentro de la misma iteración entre dos sentencias. En el espacio de iteraciones la dependencia es entre iteraciones cercanas.

Para visualizar el problema se usan grafos de dependencias

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];     /* S1 */
        B[i+1] = B[i] + A[i+1];   /* S2 */
}
```



- **Grafos**

  - Nodo: Sentencias del bucle.
  - Arco: Dependencias entre sentencias.
  - Etiqueta del arco: Nº de iteraciones que separan dos operaciones dependientes.

Si en el grafo aparecen dependencias **LOOP-CARRIED** (con valor 1 o más) entonces el bucle **NO es vectorizable**.