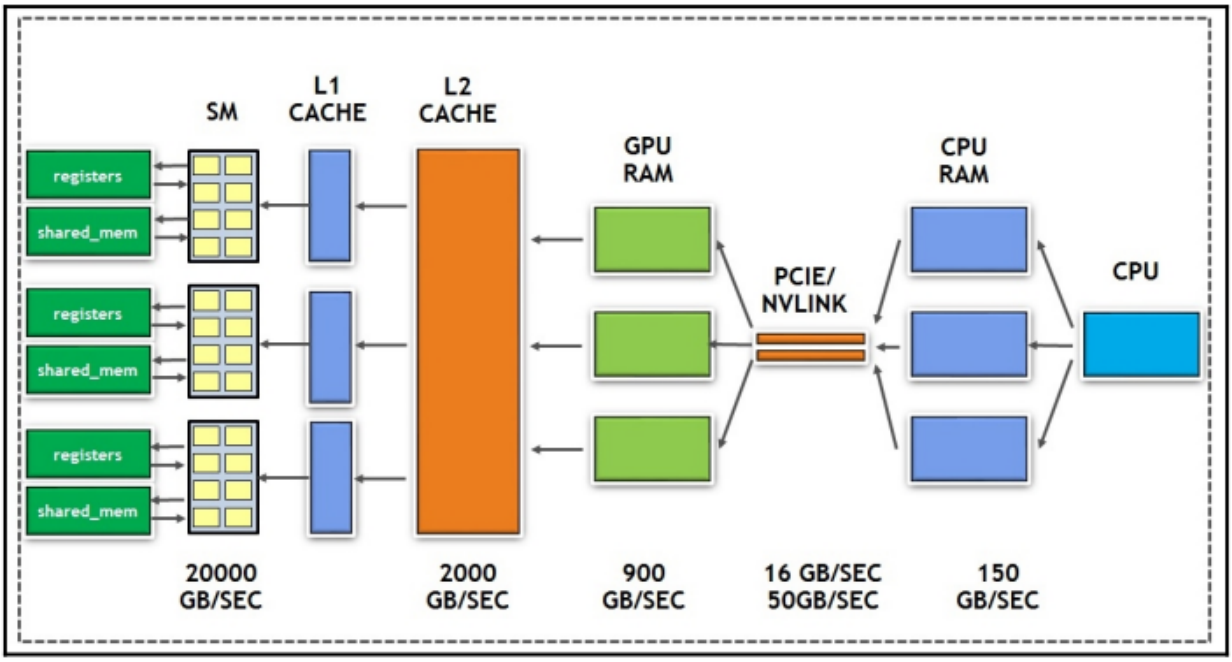
Manejo de la Memoria

El CPU y GPU cuentan con arquitecturas diferentes, incluyendo la jerarquía de su memoria. Éstas difieren no solo en tamaños y tipos, si no también en su diseño y su propósito.

Lanzar un kernel de CUDA nos puede ayudar a obtener el mejor rendimiento, pero solo cuando se utiliza el tipo correcto de memoria. Es responsabilidad del programador mapear los tipos adecuados de memoria.

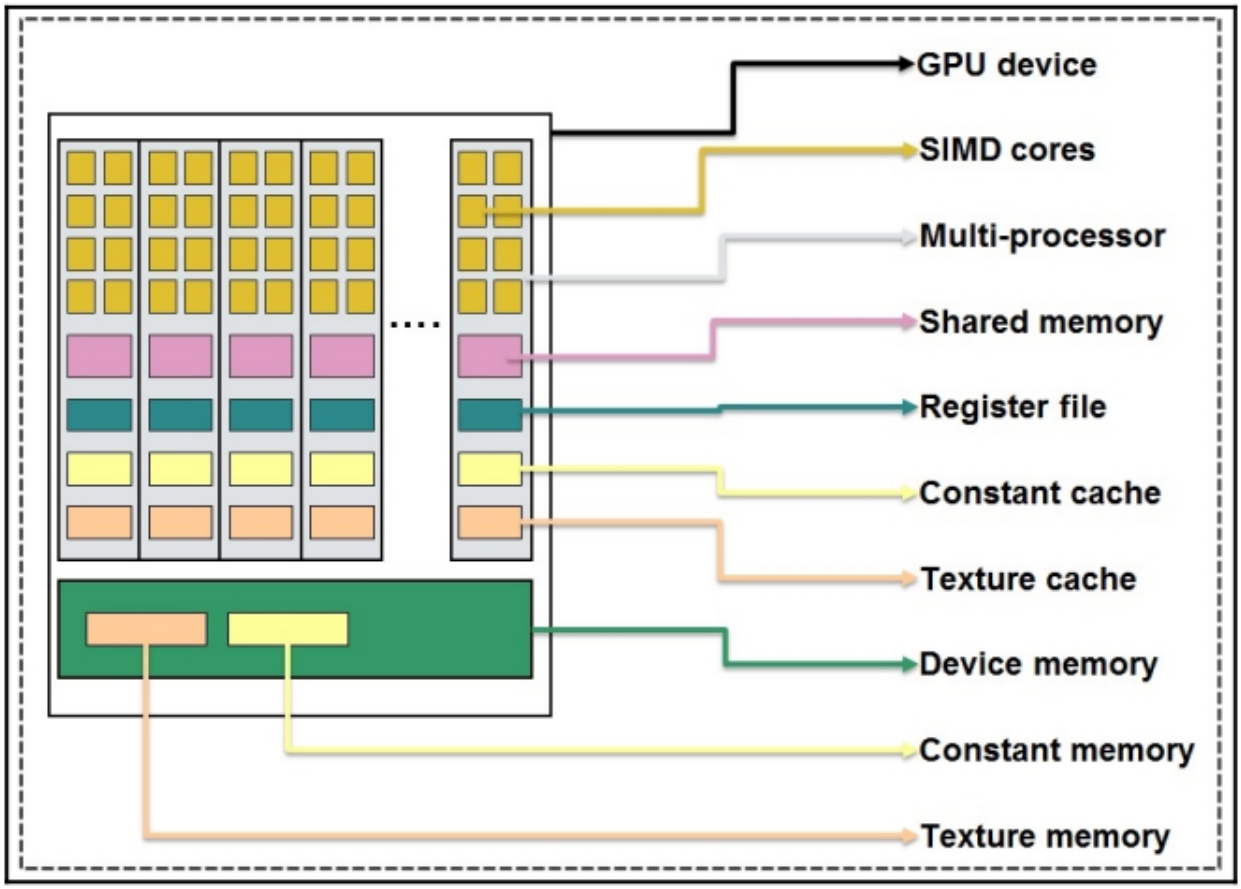
Al analizar la arquitectura de las GPUs nos damos cuenta que podemos encontrarnos con cuellos de botella si existen demasiados accesos a memoria, ocasionando que algunos hilos se queden en espera. la arquitectura CUDA nos brinda varios métodos para acceder a la memoria y resolver estos problemas.

El camino que sigue la memoria desde el CPU hasta que pueda ser utilizado por el SM se muestra en el siguiente diagrama. Cada ancho de banda es diferente, de igual manera lo es la latencia para acceder a ella.



En el siguiente diagrama se muestra la jerarquía de las memorias en la arquitectura de los GPU. Cada memoria tiene un distinto tamaño, latencia, rendimiento y accesibilidad

para el desarrollador.



Memoria Global / Memoria del “Device”

La memoria global está en una etapa donde toda la información se copia desde la memoria del CPU, por lo que utilizarla de manera correcta es primordial. Este tipo de memoria es visible para todos los hilos del kernel. También es visible para el CPU.

El programador maneja este tipo de memoria con `cudaMalloc` y `cudaFree`. La memoria global es elegida por defecto cuando se transfieren datos desde el CPU usando la API `cudaMemcpy`

Tomemos como ejemplo el programa de suma de vectores para ver cómo se utiliza la memoria del dispositivo.

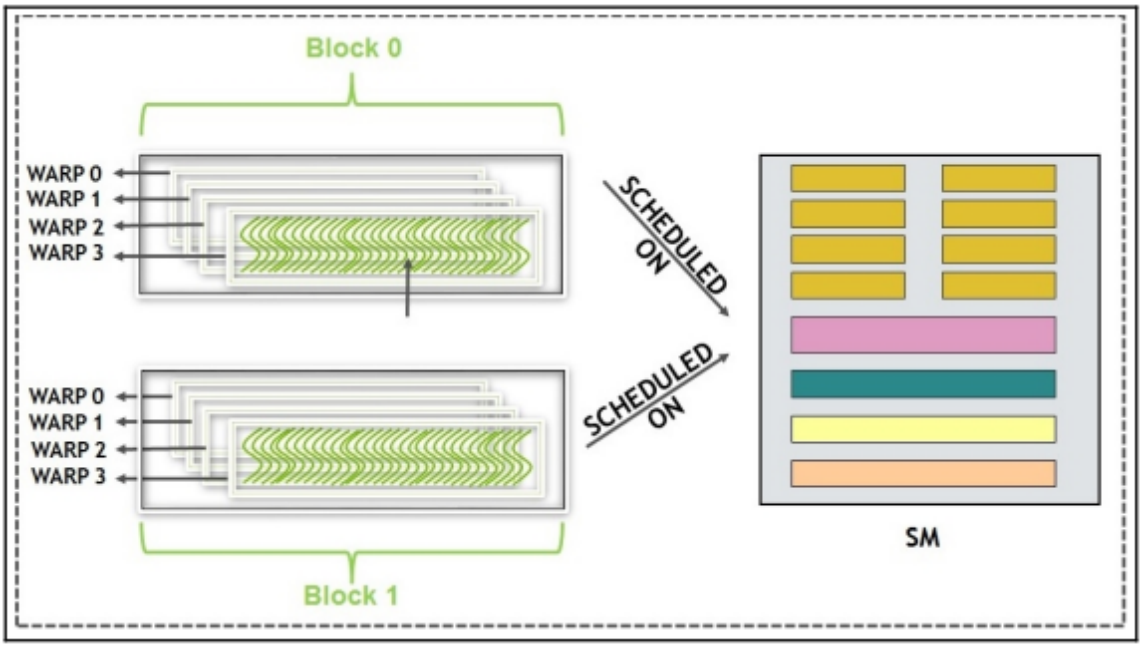
```
__global__ void device_add(int *a, int *b, int *c) {
  int index = threadIdx.x + blockIdx.x* blockDim.x;
  c[index] = a[index] + b[index]
}

int main() {
...
  //Alloc space for device copies of a, b ,c
  cudaMalloc((void**)&d_a, size);
  cudaMalloc((void**)&d_b, size);
  cudaMalloc((void**)&d_c, size);
...
  //Free space allocated for device copies
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
...
}
```

`cudaMalloc` asigna los datos en la memoria del dispositivo. Los apuntadores en los argumentos del kernel apuntan a la memoria del dispositivo. Se libera esta memoria utilizando la API `cudaFree` . Como se puede ver, todos los hilos en el bloque tienen acceso a esta memoria dentro del kernel.

Acceso contiguo o fusionado contra Acceso no contiguo a la memoria global

Para usar la memoria global de manera eficiente, es importante conocer el concepto de *warp* (conjunto de hilos). Un warp es una unidad de programación/ejecución de hilos en un SM. Una vez que el bloque ha sido asignado a un SM, se divide en una unidad de 32 hilos conocido como **warp**.

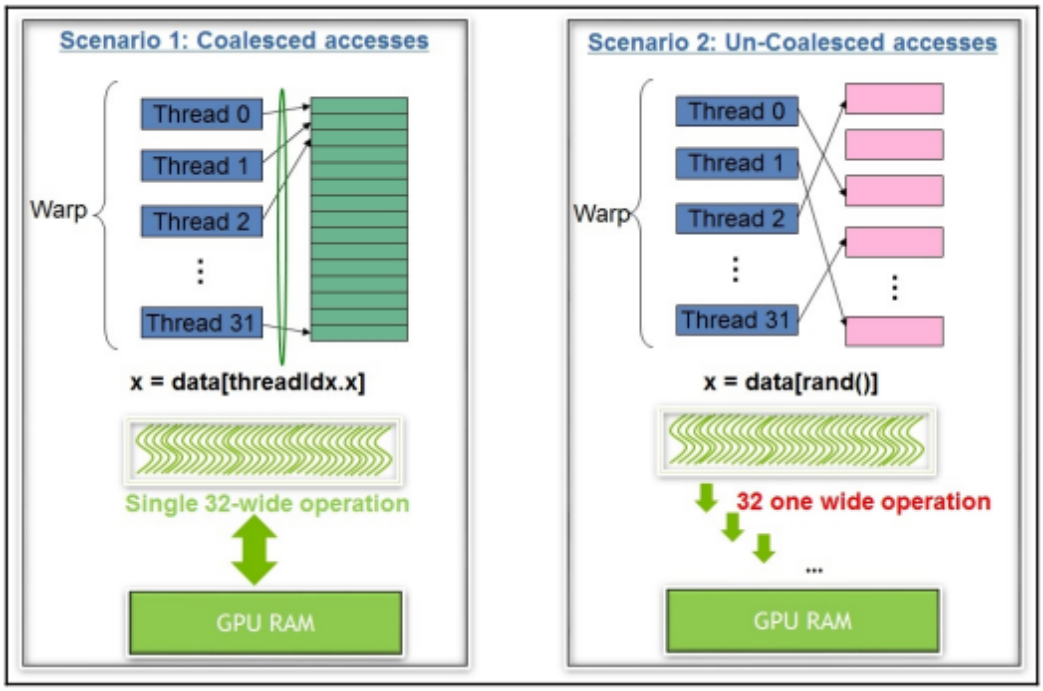
Para demostrar este concepto, observemos el siguiente ejemplo. Si se asignan dos bloques a un SM y cada bloque tiene 128 hilos, el número de warps en un bloque es de $128/32 = 4$ warps y el número total de warps en el SM es de $4 \times 2 = 8$. Obsérvese el siguiente diagrama:



De todos los warps disponibles, aquellos cuyos operandos estén listos para la siguiente instrucción serán elegibles para ejecutarse. Todos los hilos en el warp ejecutan la misma instrucción. CUDA sigue el modelo **Single Instruction, Multiple Thread**, esto es, todos los hilos en un warp obtienen y ejecutan la misma instrucción. Para utilizar los accesos a memoria global de manera óptima, el acceso debe ser contiguo. La diferencia entre contiguo y no contiguo es la siguiente:

- **Acceso contiguo a memoria global:** El acceso secuencial a memoria es adyacente.
- **Acceso no contiguo a memoria global:** El acceso secuencial a memoria no es adyacente.

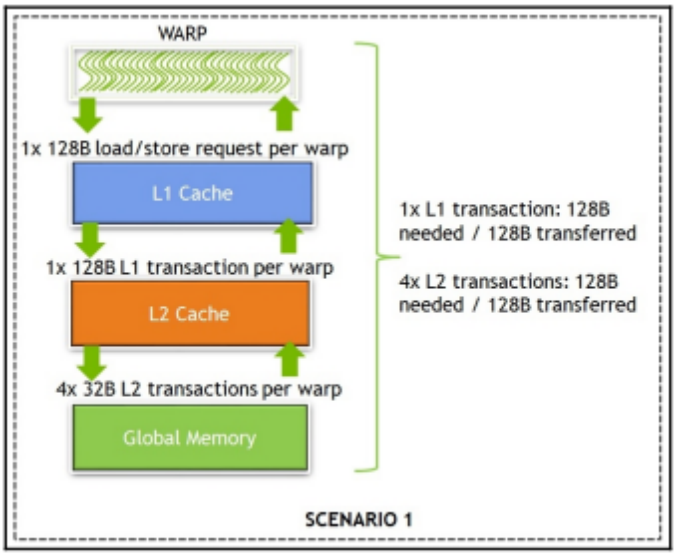
El siguiente diagrama muestra con más detalle este tipo de acceso:



Para entender mejor este concepto, es necesario entender como llegan los datos a través de cache lines.

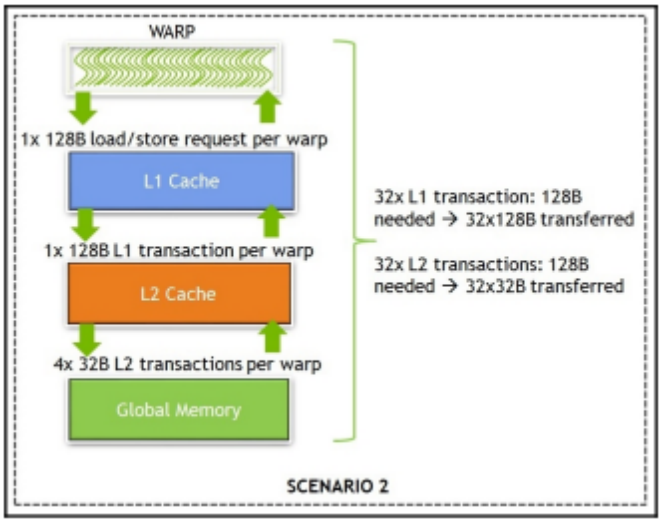
Escenario 1: Warp solicita 32 grupos alineados de 4 bits consecutivos.

La dirección cae en una línea de caché y una operación con anchura 32. El bus se utliza al 100%, esto es, estamos utilizando todos los datos solicitados desde la memoria global, colocándola en un caché sin desperdiciar ancho de banda.



Escenario 2: Warp solicita 32 grupos de 4 bits esparcidos

Aunque el warp necesita 128 bytes, hay 32 operaciones de anchura 1 siendo ejecutadas, resultando en 32×128 bytes moviéndose a través del bus. La utilización del bus es menor al 1%, resultando un desperdicio de ancho de banda.



Como pudimos observar en los diagramas anterior, es importante el cómo los hilos acceden a los datos de la memoria global. Para usar la memoria global de manera óptima es importante mejorar la continuidad de los accesos Existen varias estrategias para hacerlo; una de ellas es cambiar la disposición de los datos.

Análisis del Rendimiento de la Memoria

Es importante para un desarrollador de aplicaciones entender el rendimiento de memoria de una aplicación. Esto se puede definir de dos maneras:

- **Desde el punto de vista de la aplicación**: Cuenta los bytes solicitados por la aplicación.
- **Desde el punto de vista del hardware:** Cuenta los bytes que se movieron por el hardware.

Ambos números son completamente diferentes. Esto se debe a distintas razones, incluyendo acceso no contiguo, conflictos de banco de memoria compartidos, entre otros. Los dos aspectos que debemos utilizar para analizar desde el punto de vista de la memoria son:

- **Patrón de Direcciones:** Determinar el patrón de acceso en código es bastante difícil, por lo que usar herramientas como “profilers” se vuelve muy importante. Las métricas mostradas, como la eficiencia de memoria global y transacciones L1/L2 por acceso tienen que ser revisadas con cuidado.
- **El número de accesos concurrentes:** Debido a que el GPU oculta la latencia, se vuelve importante saturar el ancho de banda de memoria. Pero, determinar el número de accesos concurrentes normalmente no es suficiente. También, el rendimiento desde un punto de vista de Hardware es mucho más diferente que el valor teórico.

La sección anterior nos mostró ejemplos de cómo usar la memoria global y utilizarla de manera óptima. Algunas veces, el acceso contiguo desde memoria global es difícil (por ejemplo, en caso de mallas sin estructura, donde celdas vecinas puede que no residan una junto a la otra en memoria). Para resolver este problema o reducir su impacto en el rendimiento, necesitamos utilizar otro tipo de memoria: **memoria compartida.**

Memoria Compartida

La memoria compartida tiene un rol vital en la jerarquía de memoria de CUDA, conocida como **Cache manejada por el usuario**. Esto brinda un mecanismo a los usuarios para leer/escribir datos de manera contigua desde la memoria global y guardarla en memoria, la cual actuará como una caché pero puede ser controlada por el usuario.

La memoria compartida solo es visible para hilos en el mismo bloque. Todos los hilos en un bloque ven la misma versión de una variable compartida. Tiene beneficios similares al caché de un CPU, pero, mientras que una caché de CPU no puede ser manejada de manera explícita, la memoria compartida sí. La memoria compartida tiene una latencia menor a la memoria global y un ancho de banda mayor.

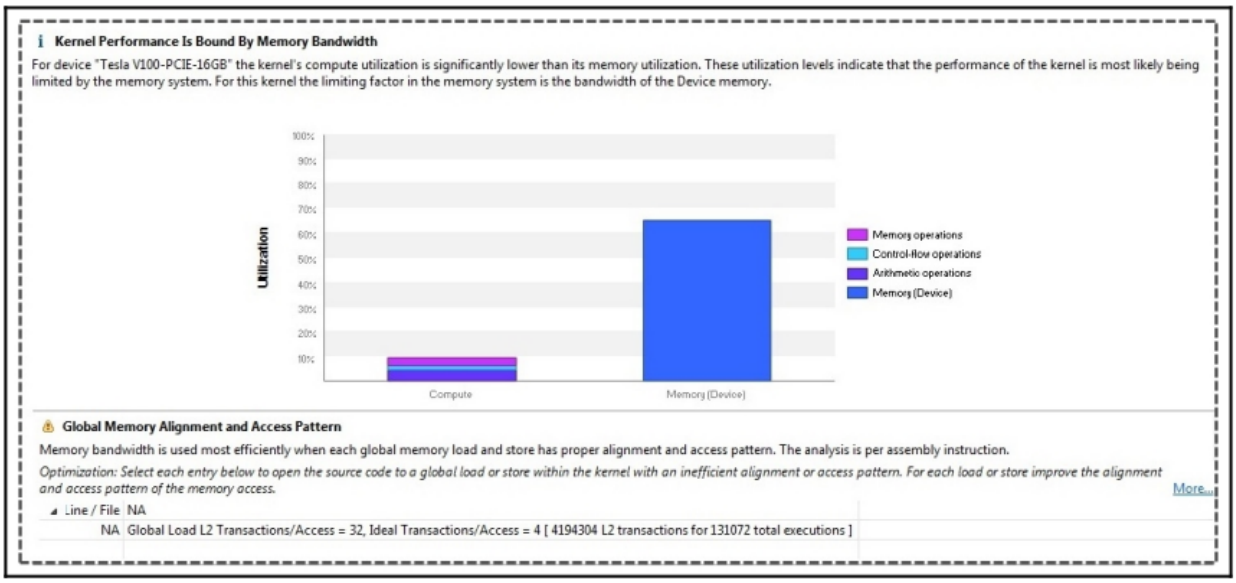
El punto clave para la utilización de la memoria compartida es que los hilos dentro de un mismo bloque pueden compartir el acceso a memoria. Los programadores de CUDA pueden usar variables compartidas para mantener los datos que se reutilizan varias veces durante la ejecución de un kernel.

Como ejemplo, veremos cómo realizar la operación de **Matriz Transpuesta**.

El siguiente código muestra cómo se haría de manera ingenua la transposición de una matriz utilizando memoria global.

```
__global__ void matrix_transpose_naive(int *input, int *output) {
  int indexX = threadIdx.x + blockIdx.x * blockDim.x;
  int indexY = threadIdx.y + blockIdx.y * blockDim.y;
  int index  = indexY * N + indexX;
  int tIndex = indexX * N + indexY;
  output[index] = input[tIndex];
}
```

Esta implementación usando memoria global resulta en un acceso no contiguo, ya sea al leer la matriz o escribiendo en la matriz. Si analizamos la ejecución del programa utilizando el profiler de nvidia obtenemos la siguiente salida:



La salida muestra que el acceso no contiguo a la memoria global limita el rendimiento de la aplicación, por lo que se tiene que trabajar en ello para mejorar el rendimiento. Una manera de resolver esto es utilizar memoria con un gran ancho de banda y baja latencia, como la memoria global. El truco está en leer y escribir desde la memoria global de manera contigua.

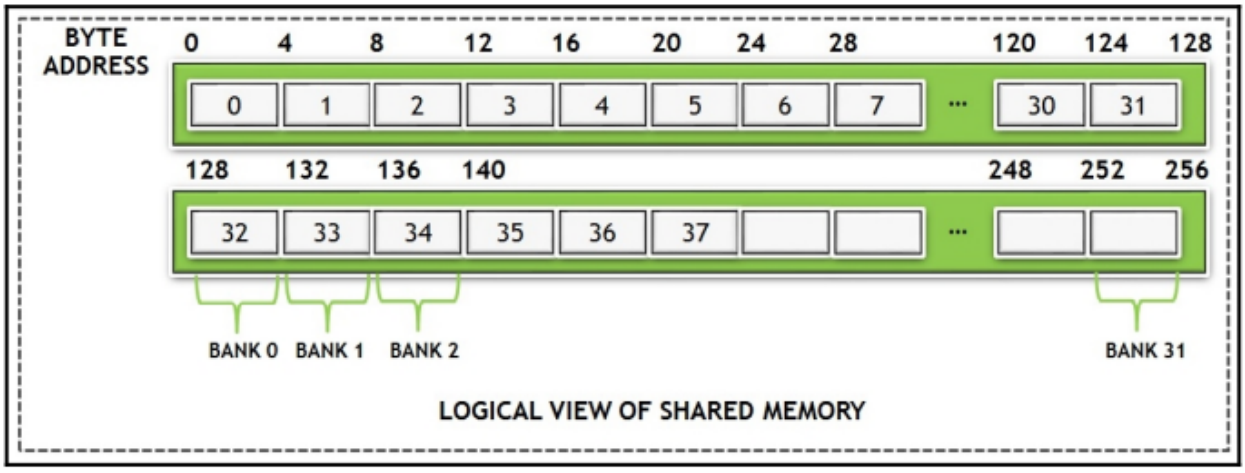
```
__global void matrix_transpose_shared(int *input, int *output) {
  __shared__ int sharedMemory [BLOCK_SIZE][BLOCK_SIZE];
  //global index
  int indexX = threadIdx.x + blockIdx.x * blockDim.x;
  int indexY = threadIdx.y + blockIdx.y * blockDim.y;

  //transposed global memory index
  int tindexX = threadIdx.x + blockIdx.y * blockDIm.x;
  int tindexY = threadIdx.y + blockIdx.x * blockDIm.y;

  //local index
  int localIndexX = threadIdx.x;
  int localIndexY = threadIdx.y;
  int index = indexY * N + indexX;
  int transposedINdex = tindexY * N + tindexX;

  //Matriz transpuesta en memoria compartida
  //La memroia global se lee en manera contigua
  sharedMemory[localIndexX][localIndexY] = input[index]
  __syncthreads();
  //La salida escrita en memoria global de manera contigua
  output[transposedIndex] = sharedMemory[localIndexY][localIndexX];
}
```

Este código muestra cómo se implementa la matriz transpuesta usando memoria compartida. La memoria global se lee de manera fusionada, mientras que la transposición sucede en memoria compartida.

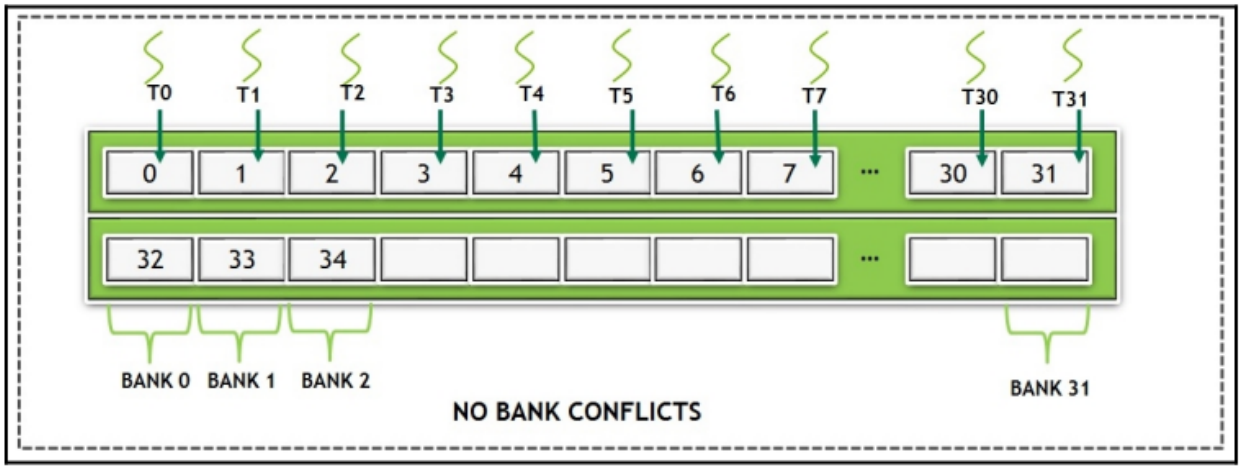
Conflicto de Bancos y su efecto en memoria compartida

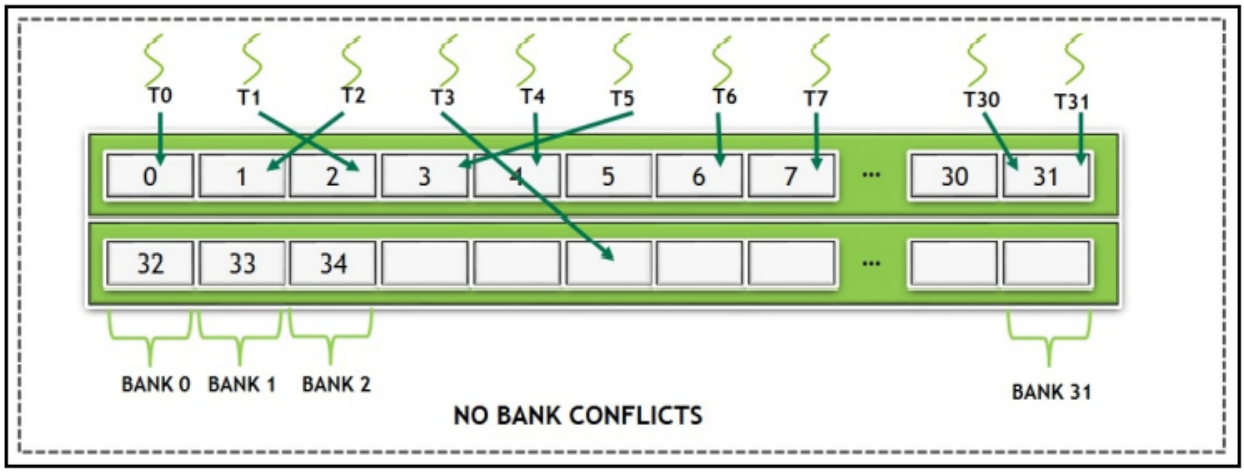
A pesar de haber obtenido una mejora en rendimiento, no significa que estamos usando la memoria compartida de manera efectiva. Al analizar la salida del profiler podemos ver que existen conflictos del patrón de acceso a memoria compartida, lo que es señal de conflicto de bancos.

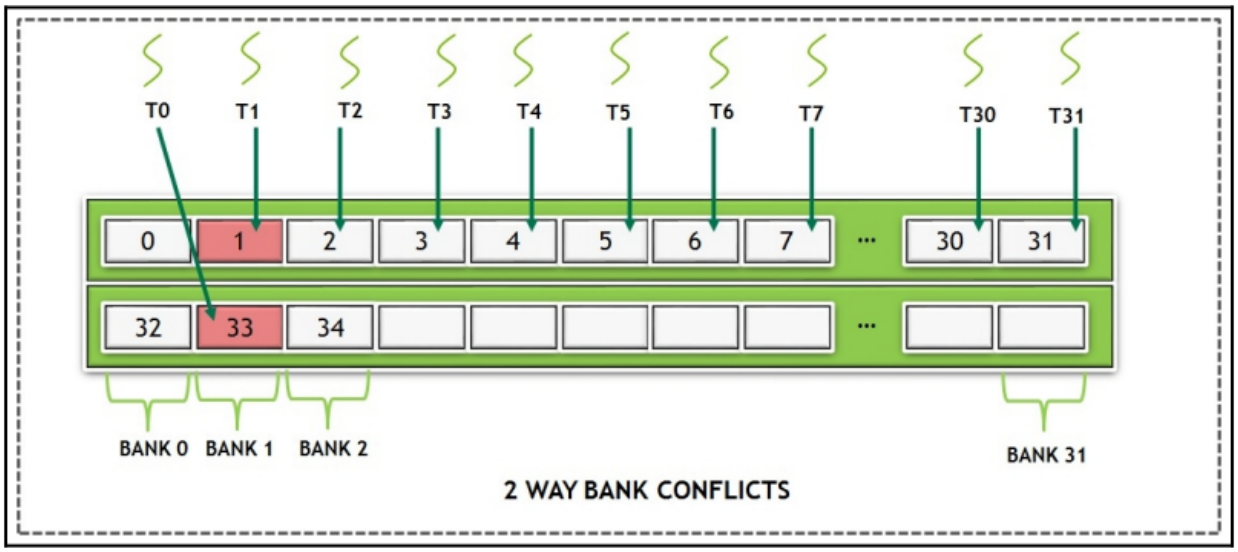
Para entender este problema, es necesario entender el concepto de *bancos*. La memoria compartida se organiza en bancos para obtener mayor ancho de banda. Cada banco puede atender una dirección por ciclo. La memoria puede atender tantos accesos simultáneos como la cantidad de bancos que tenga. Cuando se almacena un arreglo en memoria compartida, los grupos de 4bytes adyacentes van a bancos sucesivos.



Múltiples accesos simultáneos con hilos dentro de un warp hacia un banco resulta en conflicto de bancos. En otras palabras, el conflicto ocurre cuando dentro de un warp dos o más hilos acceden a diferentes palabras de 4 bytes en el mismo banco. Lógicamente, esto es cundo dos o más hilos acceden a distintos renglones del mismo banco.


T0 T1 T2 T3 T4 T5 T6 T7 T30 T31
0 1 2 3 4 5 6 7 ... 30 31
32 33 34 ...
BANK 0 BANK 1 BANK 2 BANK 31
NO BANK CONFLICTS


T0 T1 T2 T3 T4 T5 T6 T7 T30 T31
0 1 2 3 4 5 6 7 ... 30 31
32 33 34 ...
BANK 0 BANK 1 BANK 2 BANK 31
NO BANK CONFLICTS


T0 T1 T2 T3 T4 T5 T6 T7 T30 T31
0 1 2 3 4 5 6 7 ... 30 31
32 33 34 ...
BANK 0 BANK 1 BANK 2 BANK 31
2 WAY BANK CONFLICTS

En el ejemplo d la matriz transpuesta, se observó un conflicto de banco de 32-way. PAra resolver esto, utilizamos una técnica conocida como “padding”; esto es, agregar un relleno a la memoria global, es decir, una columna adicional, para que los hilos accedan a diferentes bancos de memoria, y por consecuencia, un mejor desempeño.

```
__global void matrix_transpose_shared(int *input, int *output) {
  __shared__ int sharedMemory [BLOCK_SIZE][BLOCK_SIZE + 1];
  //global index
  int indexX = threadIdx.x + blockIdx.x * blockDim.x;
  int indexY = threadIdx.y + blockIdx.y * blockDim.y;

  //transposed global memory index
  int tindexX = threadIdx.x + blockIdx.y * blockDIm.x;
  int tindexY = threadIdx.y + blockIdx.x * blockDIm.y;

  //local index
  int localIndexX = threadIdx.x;
  int localIndexY = threadIdx.y;
  int index = indexY * N + indexX;
  int transposedINdex = tindexY * N + tindexX;

  //Matriz transpuesta en memoria compartida
  //La memroia global se lee en manera contigua
  sharedMemory[localIndexX][localIndexY] = input[index]
  __syncthreads();
  //La salida escrita en memoria global de manera contigua
  output[transposedIndex] = sharedMemory[localIndexY][localIndexX];
}
```

Read-only data/cache

Esta memoria es útil para almacenar datos de solo lectura que no cambiará durante la ejecución del kernel. La caché está optimizada para este propósito, basándose en la arquitectura del kernel.

Los Datos de Solo Lectura son visibles para todos los hilos en la malla en un GPU. Los datos se marcan como “Solo Lectura” para el GPU; por el otro lado, el CPU puede tanto escribir como leer estos datos.

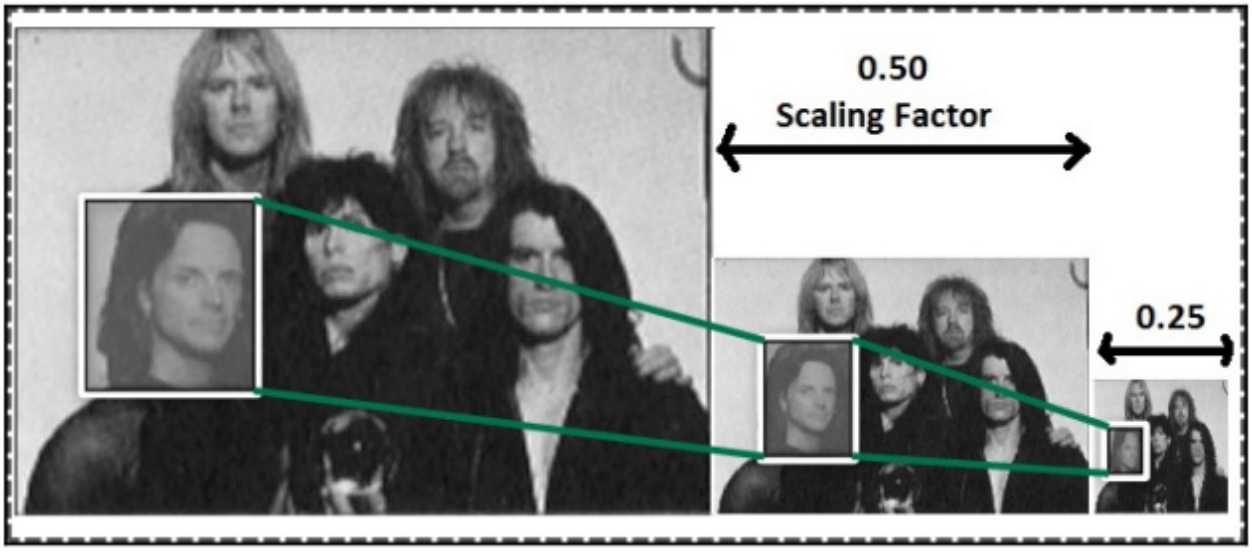
Tradicionalmente este caché se le conoce como **texture cache**. Los usuarios pueden usar la API de texture, o, con las nuevas versiones de CUDA, los apuntadores marcados como `const __restrict__` también son marcados como datos de solo lectura.

Los datos de solo lectura son ideales cuando un algoritmo necesita que todo el warp lea la misa direccion/datos. La texture cache está optimizada para localidades en 2D y 3D. La memoria Texture ha demostrado ser útiles en aplicaciones que necesiten acceso aleatorio a la memoria.

La memoria texture soporta interpolación bilineal y trilineal, lo cual es útil para algoritmos de procesamiento de imágenes como escalar una imagen.

Computer Vision - Escalamiento de una imagen usando memoria Texture

Usaremos el escalamiento de una imagen como ejemplo del uso de memoria texture.



El escalamiento de imagen requiere la interpolación de un pixel de imagen en 2 dimensiones. Testure provee ambas funcionalidades, lo cual, si se accede a memoria golbal de manera directa, resultaría en un acceso no fusionado.

Existen cuatro pasos necesarios para usar la memoria texture:

1. Declarar la memoria Texture.
2. Ligar la memoria texture a una referencia.
3. Leer la memoria texture usando una referencia texture en el kernel de CUDA.
4. Desligar la memoria texture de la referencia.

Los siguientes códigos demuestran cómo realizar los cuatro pasos para usar la memoria texture:

1. Declarar la memoria texture

```
texture<unsigned char, 2, cudaReadModelElementTypre> tex;
```

Creamos una descripción de canal, que será usado mientras ligamos a la textura.

```
cudaArray* cu_array;
cudaChannelFormatKind kind = cudaChannelFormatKindUnsigned;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(8, 0, 0, 0, kind);
```

2. Especificamos los parámetros al objeto texture

```
struct cudaTextureDesc texDesc;
memset(&texDesc,0,sizeof(texDesc));
//Establecemos la memoria a cero
texDesc.addressMode[0] = cudaAddressModeClamp;
//Establecemos la dimensión x adressmode a Clamp
texDesc.addressMode[1] = cudaAddressModeClamp;
//Establecemos la dimensión y addressmode a clamp
texDesc.filterMode = cudaFilterModePoint;
//El modo de filtro establecido a Point
texDesc.readMode = cudaReadModeElementType;
//Leyendo el tipo de elemento y no interpolado
texDesc.normalizedCoords = 0;
```

3. Leer la memoria texture desde la referencia texture en el kernel de CUDA

```
imageScaledData[index] = tex2d<unsignedchar>(
            texObj,
            (float)(tidX*scale_factor),
            (float)(tidY*scale_factor)
        );
```

4. Destruir el objeto texture

```
cudaDestroyTextureObject(texObj);
```

Los aspectos importantes en la memoria texture, que son configurados por el desarrollador, son los siguientes:

- **Dimensión del Texture:** Esto define si la texture es de 1, 2 o 3 dimensiones. La profundidad, ancho y altura también son definidos para cada dimensión.
- **Texture type:** Esto define el tamaño en términos de si es un integer básico, o un texel de punto flotante.
- **Modo de lectura del Texture:** Define cómo son leídos los elementos. Pueden ser leídos en `NormalizedFloat` , que espera los índices en un rango de [0.0 1.0] y [-1.0 1.0], o en modo `ModeElement`.
- **Modo de direccionamiento del texture:** Algo especial de la memoria texture es cómo accederá a las direcciones fuera del rango. Aunque suene raro, es algo muy común en algoritmos que trabajan con imágenes. Las Textures brindan la opción de tratar los índices fuera de rango como sujetadas (clamped), envueltas (wrapped) o reflejadas (mirrored).
- **Modo de filtrado del Texture:** Esto define cómo se calcula el valor regresado cuando se obtiene el texture. Se soportan dos modos de filtrado: `cudaFilterModePoint` y `cudaFilterModeLinear` . Cuando se establece de manera lineal, la interpolación es posible. En `ModePoint` no realiza una interpolación pero regresa un texel de la coordenada más cercana.

La implementación de la memoria texture en `CUDA.jl` sigue siendo experimental.

Registros en GPU

Una de las principales diferencias entre las arquitecturas del CPU y del GPU es la abundancia de registros en el GPU comparado con el CPU. Estos registros ayudan a

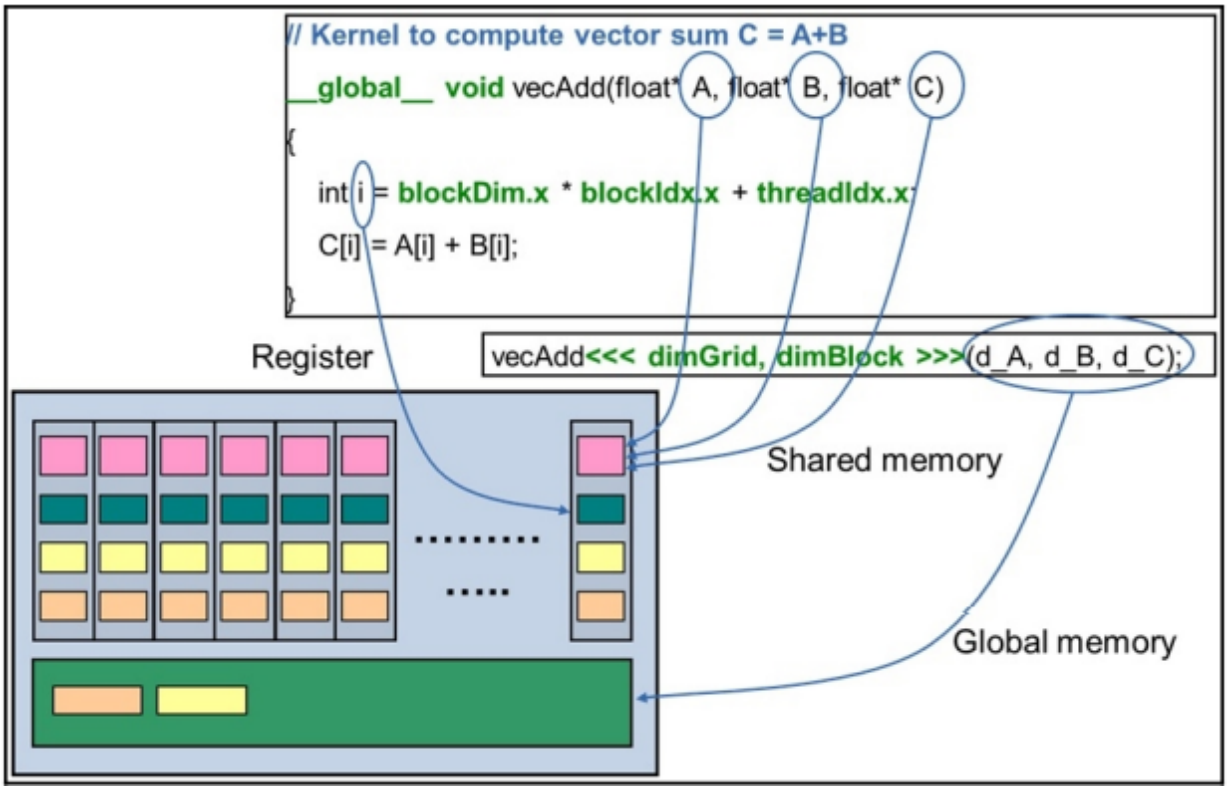
los hilos a mantener la mayoría de sus datos, reduceindo la latencia en el cambio de contexto.

Los registros tienen un alcance de un solo hilo. Una copia privada de la variable es creada para todos los hilos lanzados en el grid. Cada hilo tiene acceso a su copia privada de la variable.

Las variables locales que se declaran como parte del kernel son almacenadas en los registros. Valores intermedios también son almacenados en registros.

Cada SM tiene un número fijo de registros. Durante la compilación, se trata de encontrar el mejor número de registros por hilo. Cuando no alcanza la cantidad de registros, los datos se mueven a memoria local, que puede ser un caché L1/L2 o incluso más abajo en la jerarquía de memoria, como memoria global. A esto se le conoce como **derrame de registros** (register spills).

El número de registros por hilo juega un papel importante en cuántos bloques e hilos pueden estar activos en un SM. En general, se recomienda no declarar demasiadas variables locales innecesarias. Si los registros están restringiendo el número de hilos que pueden ser agendados en un SM, el desarrollador debería considerar reestructurar su código dividiendo el kernel en dos - o más si es posible.



Las variables declaradas dentro del kernel `vecAdd` se almacenan en registros de memoria. Los argumentos que se pasan al kernel apuntan a memoria global, pero la variable en sí se almacena en registros de memoria compartida ubicados en el GPU. El diagrama anterior muestra la jerarquía de memorias de la arquitectura CUDA y la ubicación de distintos tipos de variables.