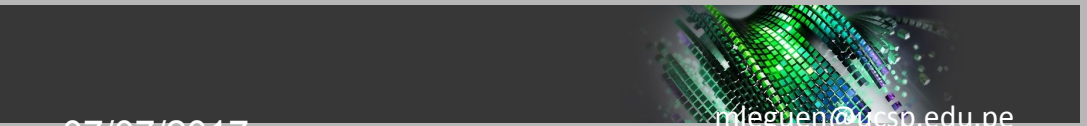
Universidad Católica San Pablo

CUDA **Multiples GPU**

Marc-Antoine Le Guen



mleguen@ucsp.edu.pe

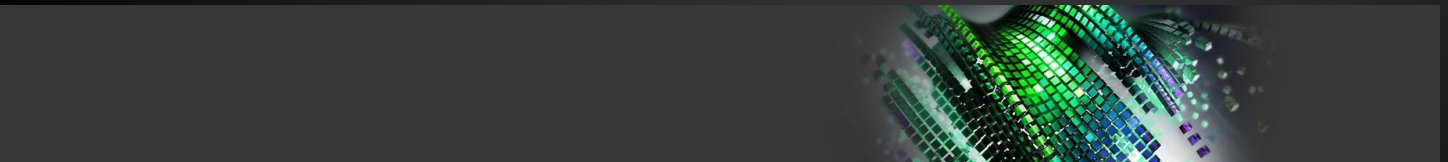
Múltiples GPU

- Muchos sistemas contienen varios GPUs
- Algunas tarjetas gráficas contienen 2 GPUs
- Se trata de aprovechar al máximo el sistema

Múltiples GPU - Producto Escalar

- Estructura de datos:
  - int deviceID //id de GPU
  - int size
  - float *a // input vector
  - float *b // input vector
  - float returnValue // final result

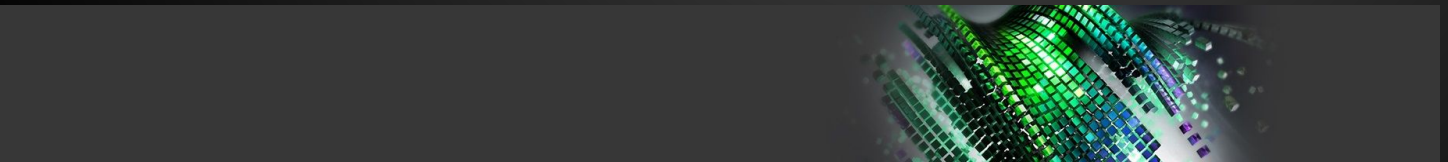
```
struct DataStruct {
    int deviceID;
    int size;
    float *a;
    float *b;
    float returnValue;
};
```



Múltiples GPU - Producto Escalar

- Cuanto GPU disponibles

```
int deviceCount;
cudaGetDeviceCount( &deviceCount );
if (deviceCount < 2) {
        printf( "We need at least two devices, but only found %d\n", deviceCount );
        return 0;
}
```



Múltiples GPU - Producto Escalar

- Como manejar dos GPUs
  - Utilizar la estructura de datos definida anteriormente
    - N/2 la mitad de los datos para cada uno
    - puntero + N/2 para el segundo GPU <- offset
  - Un thread del CPU por GPU

```
DataStruct data[2];

data[0].deviceID = 0;
data[0].size = N/2;
data[0].a = a;
data[0].b = b;

data[1].deviceID = 1;
data[1].size = N/2;
data[1].a = a + N/2;
data[1].b = b + N/2;
```

Múltiples GPU - Producto Escalar

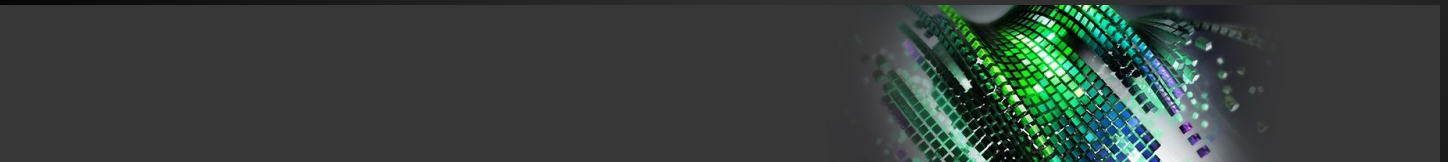
- Como manejar dos GPUs
  - Un thread del CPU por GPU
    - void routine (void* pvoidData)
    - Sincronizar el thread
    - Recuperar el resultado a dentro de la estructura de datos

```
Thread   thread = start_thread( routine, &(data[0]) ); //thread 1 -> device 0
routine( &(data[1]) ); // thread 0 -> device 1
end_thread( thread );  // syncro

free( a );
free( b );
printf( "Value calculated:  %f\n", data[0].returnValue + data[1].returnValue );
```

Múltiples GPU - Producto Escalar

- void routine (void* pvoidData)
  - Definir el device que usaremos cudaSetDevice
    - data->deviceID

```
void* routine( void *pvoidData ) {
     DataStruct  *data = (DataStruct*)pvoidData;
     cudaSetDevice( data->deviceID );

.......
```



Múltiples GPU - Producto Escalar

- void routine (void* pvoidData)

```
void* routine( void *pvoidData ) {
        DataStruct   *data = (DataStruct*)pvoidData;
        cudaSetDevice( data->deviceID );

        int     size = data->size;
        float   *a, *b, c, *partial_c;
        float   *dev_a, *dev_b, *dev_partial_c;

        // allocate memory on the CPU side
        a = data->a;
        b = data->b;
        partial_c = (float*)malloc( blocksPerGrid*sizeof(float);

        // allocate the memory on the GPU
        cudaMalloc( (void**)&dev_a, size*sizeof(float) );
        cudaMalloc( (void**)&dev_b, size*sizeof(float) );
        cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) );

        // copy the arrays 'a' and 'b' to the GPU
        cudaMemcpy( dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice  );
        cudaMemcpy( dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice  );
```

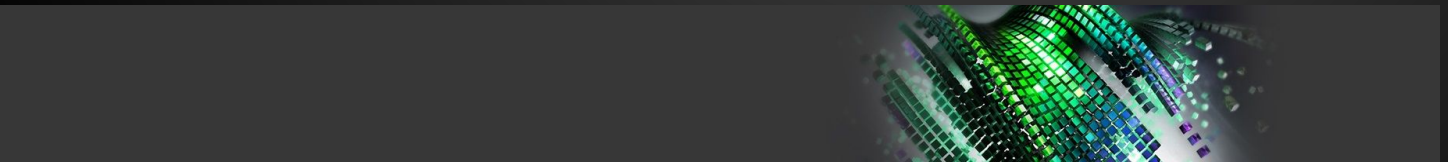
Múltiples GPU - Producto Escalar

- void routine (void* pvoidData)

```
    dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b, dev_partial_c );
    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost );
    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
       c += partial_c[i];
    }
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_partial_c );

    // free memory on the CPU side
    free( partial_c );

    data->returnValue = c;
    return 0;
}
```

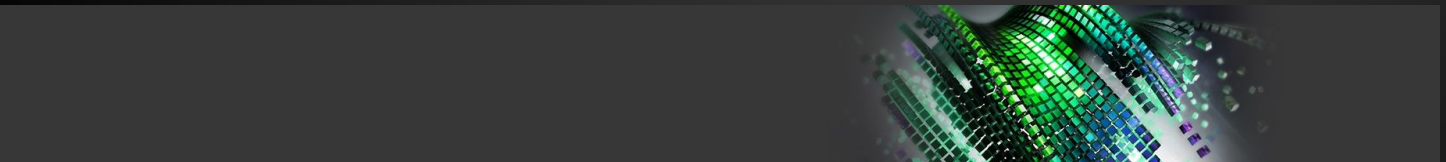
Múltiples GPU

- Definir una estructura de datos
- Definir una rutina : trabajo que cada GPU tendrá que hacer
- Lanzar un thread del CPU para cada GPU que trabajará en paralelo
- Y porque no usamos zero-copy memory ?



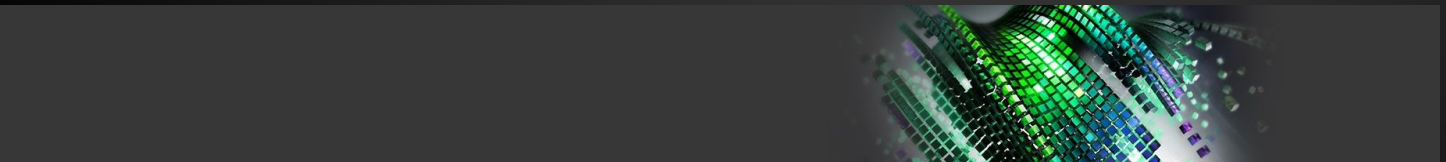
Múltiples GPU - Portable Pinned memory

- La memoria pinned aparece como *pinned* a un solo thread del CPU (el que hizo la asignación)
  - El otro thread verá esta memoria como una memoria estándar
  - CudaMemCpy para este thread se dará como una memoria estándar -> 50% más lenta.
  - Los streams no funcionaran porque necesitan pinned memory



Múltiples GPU - Portable Pinned memory

- Felizmente hay una solución simple a este problema
  - declarar la memoria como ***portable***
    - cudaHostAlloc
      - Flag : cudaHostAllocPortable utilizable con otros flags como cudaHostAllocWriteCombined



Múltiples GPU - Portable Pinned memory - Producto Escalar

- Verificar la compatibilidad del sistema

```
int main( void ) {
      int deviceCount;
      cudaGetDeviceCount( &deviceCount );
      if (deviceCount < 2) {
          printf( "We need at least two compute 1.0 or greater "
                "devices, but only found %d\n", deviceCount );
          return 0;
      }

      cudaDeviceProp  prop;
      for (int i=0; i<2; i++) {
          cudaGetDeviceProperties( &prop, i );
          if (prop.canMapHostMemory != 1) {
              printf( "Device %d can not map memory.\n", i );
              return 0;
          }
      }
```

Múltiples GPU - Portable Pinned memory - Producto Escalar

- cudaHostAllocPortable
  - Un requerimiento es usar cudaSetDevice para avisar

```
float *a, *b;
cudaSetDevice( 0 );
cudaSetDeviceFlags( cudaDeviceMapHost );
cudaHostAlloc( (void**)&a, N*sizeof(float), cudaHostAllocWriteCombined | cudaHostAllocPortable |
cudaHostAllocMapped );
cudaHostAlloc( (void**)&b, N*sizeof(float), cudaHostAllocWriteCombined | cudaHostAllocPortable |
cudaHostAllocMapped );

// fill in the host memory with data
for (int i=0; i<N; i++) {
a[i] = i;
b[i] = i*2;
}
```

Múltiples GPU - Portable Pinned memory - Producto Escalar

- Inicializar la estructura de datos de la misma forma que antes

```
// fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    // prepare for multithread
    DataStruct  data[2];
    data[0].deviceID = 0;
    data[0].offset = 0;
    data[0].size = N/2;
    data[0].a = a;
    data[0].b = b;

    data[1].deviceID = 1;
    data[1].offset = N/2;
    data[1].size = N/2;
    data[1].a = a;
    data[1].b = b;
```

Múltiples GPU - Portable Pinned memory - Producto Escalar

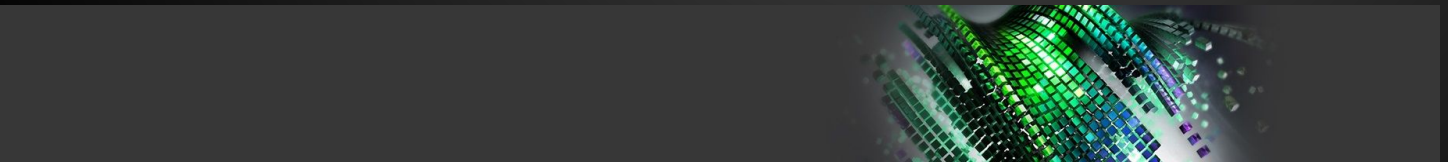
- Lanzar los threads con las rutinas y recuperar el resultado tampoco cambia aquí

```
	Thread   thread = start_thread( routine, &(data[1]);
	routine( &(data[0]);
	end_thread( thread );


	// free memory on the CPU side
	cudaFreeHost( a );
	cudaFreeHost( b );

	printf( "Value calculated:  %f\n",
		data[0].returnValue + data[1].returnValue );

	return 0;
}
```

Múltiples GPU - Portable Pinned memory - Producto Escalar

- No podemos llamar dos veces cudaSetDevice

```
void* routine( void *pvoidData ) {
    DataStruct  *data = (DataStruct*)pvoidData;
      if (data->deviceID != 0) {
        cudaSetDevice( data->deviceID );
        cudaSetDeviceFlags( cudaDeviceMapHost );
      }
```



Múltiples GPU - Portable Pinned memory - Producto Escalar

- Recuperar los punteros para el GPU
- aplicar el offset a los datos

```
int     size = data->size;
  float   *a, *b, c, *partial_c;
  float   *dev_a, *dev_b, *dev_partial_c;

  // allocate memory on the CPU side
  a = data->a;
  b = data->b;
  partial_c = (float*)malloc( blocksPerGrid*sizeof(float);

  // allocate the memory on the GPU
  cudaHostGetDevicePointer( &dev_a, a, 0 );
  cudaHostGetDevicePointer( &dev_b, b, 0 );
  cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) );

  // offset 'a' and 'b' to where this GPU is gets it data
  dev_a += data->offset;
  dev_b += data->offset;
```

Múltiples GPU - Portable Pinned memory - Producto Escalar

- Ejecutamos el kernel y recuperamos el resultado

```
    dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b, dev_partial_c );
    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost );

    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

    cudaFree( dev_partial_c );
    // free memory on the CPU side
    free( partial_c );

    data->returnValue = c;
    return 0;
}
```