Implementación de BLAS SSPR en GPGPU
Programación GPGPU - UNLP/UDC

Andrés More
more.andres@gmail.com

Resumen

El trabajo consiste en implementar diferentes versiones de la función SSPR de la librería BLAS. El objetivo principal es contrastar el uso de CPUs contra GPUs, tanto en rendimiento como complejidad de implementación.

1. SSPR

Esta función realiza una operación *rank-1 update* en una matriz simétrica con números de punto flotante de precisión simple, una representación gráfica puede verse en la Figura 1.

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$$

Figura 1: Cálculo de SSPR

Esta operación pertenece al nivel 2 de BLAS (Basic Linear Algebra Subprograms), ya que opera sobre una matriz y un vector. Aunque la matriz se define como un vector que contiene la parte superior (o inferior) triangular empacada secuencialmente por columnas.

En particular el elemento A_{ij} en el caso superior se encuentra dentro un vector AP según la ecuación 1. El vector AP tiene entonces un tamaño de $((n \times (n+1))/2)$.

$$AP(i + (j(j-1)/2)) = A_{ij} (\forall j >= i) \tag{1}$$

2. Algoritmos

En concreto se codificaron 4 versiones de la función SSPR, solo se implementó soporte para matrices superiores con desplazaje 1.

2.1. Versión secuencial en CPU

Basado fuertemente en el BLAS original `http://www.netlib.org/blas/sspr.f` y también en el implementado por la librería científica GNU (GSL) en `gsl-X/cblas/source_spr.h`.

```
for (i = 0; i < n; i++) {
   const float tmp = alpha * x[i];

   for (j = 0; j <= i; j++)
       ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
}
```

Se ve claramente que la cantidad de cómputo por cantidad de datos a transferir no es significante y por lo tanto una implementación GPU no va ser mucho más eficiente que en una CPU.

2.2. Versión en GPU con llamada a cuBLAS

Para utilizar cuBLAS basta una simple llamada como la siguiente.

```
status = cublasSspr(handle, mode, n, &alpha, cx, incx, cap);
    if (status != CUBLAS_STATUS_SUCCESS)
        err("cublasSspr: %d (%s)", status, cublas2str[status]);
```

Este código altamente optimizado puede encontrarse en el paquete *NVIDIA CUBLAS* 1.1 en los archivos `sspr.cu` y `sspr.h`. Para acceder al código hay que inscribirse como desarrollador oficial.

El kernel primero carga en memoria compartida $X[i]$ e $X[j]$, luego procesa varios elementos de la matriz por hilo reusando los mismos. Otro truco que utiliza es realizar operaciones de desplazaje de bits en lugar de división por 2 para calcular ubicaciones dentro de la matriz compactada.

Para utilizar las funciones de soporte recomendadas además de las específicas de CUDA hay que utilizar `cublasCreate()`, y muchas otras como `cublasSetVector()`, `cublasGetVector()`, `cublasDestroy()`; incluyendo manejo de errores con tipos de datos opacos como `cudaError_t`, `cublasHandle_t` y `cublasStatus_t`, `cublasFillMode_t`.

2.3. Versión directa en GPU

La versión directa en GPU utiliza un hilo por cada elemento del vector X computando en paralelo la matriz resultado.

```
__global__ void sspr_naive_kernel(int uplo, int n, float alpha,
                                  const float *x, int incx, float *ap)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        const float tmp = alpha * x[i];

        int j = 0;
        for (j = 0; j <= i; j++)
            ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
        }
}
```

Para ejecutar el kernel se utiliza una llamada similar al caso anterior.

```
sspr_naive_kernel<<< (n / capabilities.maxThreadsPerBlock),
                (capabilities.maxThreadsPerBlock) >>>
                (uplo, n, alpha, cx, incx, cap);
```

2.4. Versión en GPU con memoria compartida

La primer versión optimizada usa memoria compartida para disminuir el tiempo de acceso a parte del vector X. Todos los hilos de un mismo bloque cargar en memoria compartida un elemento. Luego al utilizar los elementos de X se comprueba si están en memoria compartida o global.

```
__global__ void sspr_optimized_kernel(int uplo, int n, float alpha,
                                      const float *x, int incx, float *ap)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        int tid = threadIdx.x;

        extern __shared__ float cache[];
        float *xi = (float *) cache;
        xi[tid] = x[i];

        __syncthreads();

        const float tmp = alpha * x[i];

        int j = 0;
        for (j = 0; j <= i; j++) {
            if (blockIdx.x * blockDim.x < j \&\&
                blockIdx.x * blockDim.x + 1 > j)
```

```
                ap[((i*(i+1))/ 2 + j)] += xi[j] * tmp;
            else
                ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
        }
    }
}
```

3. Resultados

El sistema utilizado es una *HP Mobile Workstation EliteBook 8530w* contando con un CPU *Intel T9600* y una tarjeta gráfica *Quadro FX 770m*.

```
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Duo CPU T9600 @ 2.80GHz
stepping        : 10
cpu MHz         : 2793
cache size      : 6144 KB
fpu             : yes
cpuid level     : 13
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
                  ss ht tm pbe pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave osxsave lahf_lm
TLB size        : 0 4K pages
```

Se utilizó *CUDA Toolkit 4.2*, publicado en Febrero de 2012. Como dispositivo se utilizó una version mobile de una *Quadro FX 770M*, con una cantidad promedio de 462 MB de memoria global disponible para cálculo.

```
capabilities.name = Quadro FX 770M
capabilities.totalGlobalMem = 512.00 MB
capabilities.sharedMemPerBlock = 16.00 KB
capabilities.regsPerBlock = 8192
capabilities.warpSize = 32
capabilities.memPitch = 2097152.00 KB
capabilities.maxThreadsPerBlock = 512
capabilities.maxThreadsDim = 512 512 64
capabilities.maxGridSize = 65535 65535 1
capabilities.totalConstMem = 64.00 KB
capabilities.major = 1
capabilities.minor = 1
capabilities.clockRate = 1220.70 MHz
capabilities.textureAlignment = 256
capabilities.deviceOverlap = 1
capabilities.multiProcessorCount = 4
cudaMemGetInfo.free = 462 MB
```

En la figura 2 se grafican los tiempos obtenidos en segundos con diferentes tamaños de matriz. Los tiempos son el promedio de 32 ejecuciones de cada

método. Para validar las implementaciones se redujo la matriz a una suma global.

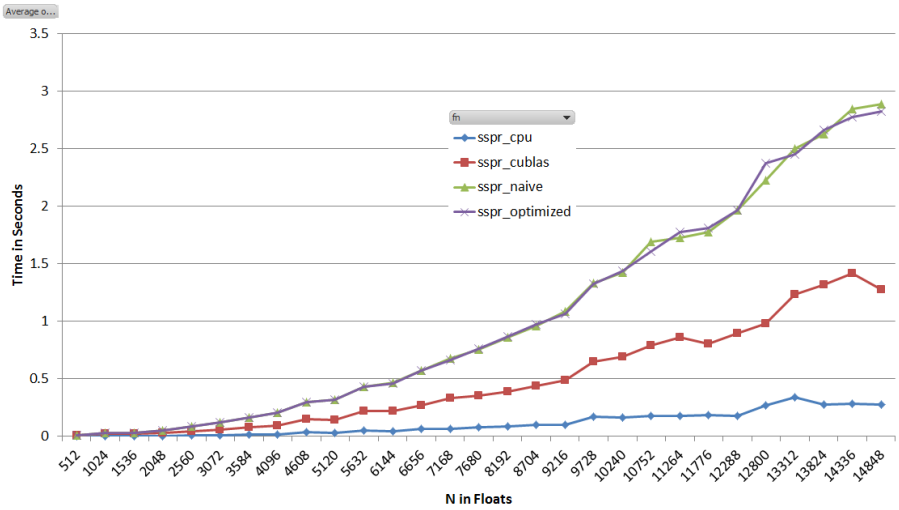

Figura 2: Comparación del Tiempo de Ejecución

Se denota claramente que la optimización realizada no impacta positivamente, por lo que la estrategia que aplica cuBLAS es superior. Sin embargo la versión serial en CPU es aún más eficiente.

4. Análisis de Rendimiento

Se realizo un análisis de rendimiento como para justificar las optimizaciones y entender el rendimiento de cada método.

4.1. Uso de Registros

Primero se ejecutó el compilador *nvcc* con información extra como para controlar la cantidad de registros. La cantidad de registros por thread es mínima.

```
ptxas info: Compiling entry function 'sspr_naive_kernel' for 'sm_13'
ptxas info: Used 7 registers, 48+16 bytes smem, 4 bytes cmem[1]
ptxas info: Compiling entry function 'sspr_optimized_kernel' for 'sm_13'
ptxas info: Used 10 registers, 48+16 bytes smem, 4 bytes cmem[1]
```

Se encontró el problema que esta información no se muestra cuando se especifica `-arch=compute_11` pero sí con `-arch=sm_13`.

4.2. NVIDIA Visual Profiler

En la figura 3 se muestra una captura de la herramienta *nvvp* sobre el código desarrollado.

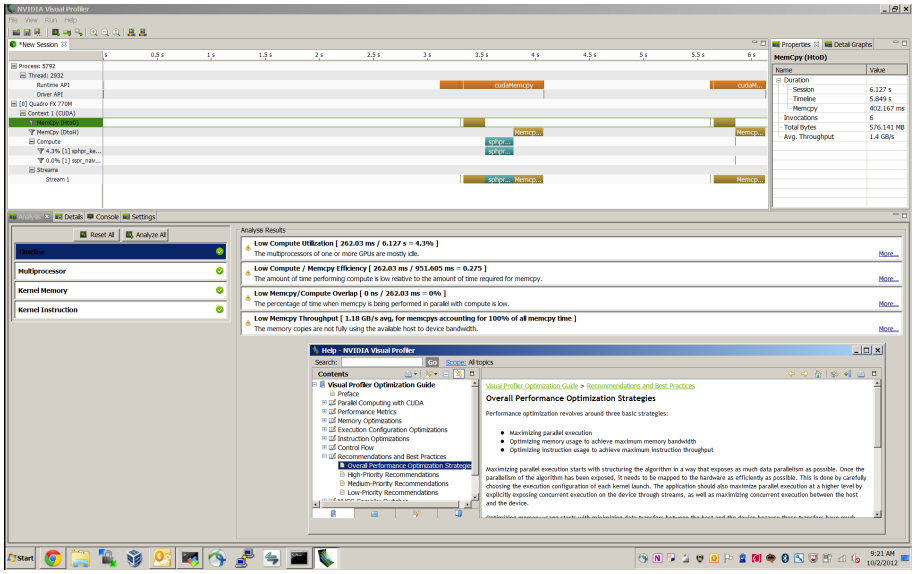

Figura 3: Análisis de Rendimiento

La herramienta identifica que:

- no hay solapamiento de transferencia de datos y cómputo
- no se utiliza todo el ancho de banda durante la transferencia
- no se utiliza completamente la capacidad de cómputo

Estos problemas se deben mayormente a la baja cantidad de memoria disponible en el dispositivo, y al hecho de que debido a la representación compactada de matrices no se realiza suficiente cantidad de cómputo por byte transmitido como para justificar el uso de GPU como acelerador.

5. Aceleraciones

Para calcular las aceleraciones logradas se eligió un tamaño de entrada de casi la totalidad de la memoria libre de la GPU. Las aceleraciones más interesantes son mostradas en la tabla 1.

```
INPUT
-----
SSPR_N = 14848 floats (packed 110238976 floats)
```

```
SSPR_ALPHA = 3.141593
memory = 420 MB
cudaMemGetInfo.free = 462 MB
```

Cuadro 1: Aceleración para N=14848

Método 1	**Método 2**	**Aceleración**
cublas (1.4995 seg)	cpu (0.389625 seg)	3.85x
naive (3.090625 seg)	cpu (0.389625 seg)	7.93x
optimized (2.97325 seg)	cpu (0.389625 seg)	7.63x
naive (3.090625 seg)	cublas (1.4995 seg)	2.06x
optimized (2.97325 seg)	cublas (1.4995 seg)	1.98x
optimized (2.97325 seg)	naive (3.090625 seg)	0.95x

6. Conclusión

6.1. Rendimiento

Un estudio relacionado fue realizado por Microsoft Research, comparando el rendimiento de BLAS nivel 2 en FPGA, CPU y GPU. La figura 4 muestra los resultados que convalidan los obtenidos durante este trabajo. Es decir, la cantidad de cómputo no es significante como para justificar la transferencia de datos.

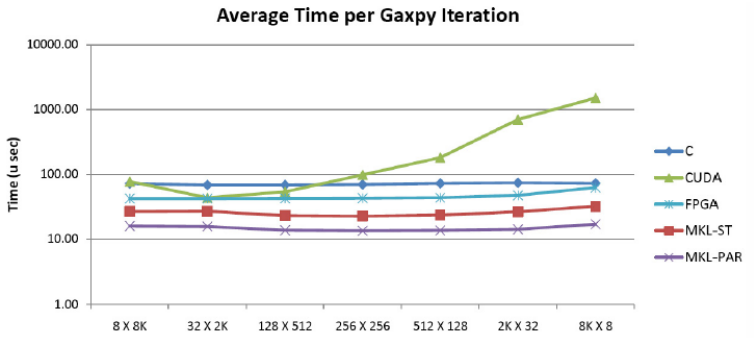

Figura 4: Microsoft Research: BLAS Comparison on FPGA, CPU and GPU

6.2. cuBLAS

El encuentro con la librería cuBLAS no fue muy grato. No hay funciones que se encarguen del ciclo completo de inicialización, transferencia de datos, cálculo y transferencia de resultados. Se necesitan aproximadamente 60 líneas de código para poder ejecutar una llamada BLAS.

cuBLAS no incluye una función para traducir códigos de error como `strerr()` o `cudaGetErrorString()`. La documentación es confusa, por ejemplo dice que no hay

que usar `cudaAlloc()` porque esta deprecada, pero la referencia a `cublas_sspr()` dice que hay que usarla.

Para este trabajo hubo que realizar una migración del código de ejemplo *Linux* a *Windows*, durante el proceso se identificó y resolvió el problema de que `gettimeofday()` no está disponible.