Rutinas y Librerías de Álgebra Lineal en Sistemas Many-Core (GPUs y MIC)

Luis P. García González

Unidad de Informática
Universidad Politécnica de Cartagena

UPCT

Diciembre de 2014

Introducción

- Procesadores Multicore, sistemas cc-NUMA, GPUs y Coprocesadores pueden ofrecer mejoras en el rendimiento de las librerías de Álgebra Lineal
  - Son necesarias técnicas de optimización del software para poder obtener beneficios de las posibilidades que ofrecen estos sistemas computacionales
  - Modelar el tiempo de ejecución de la rutina
  - Análisis experimental del comportamiento de la rutina con la variación de ciertos parámetros
- En esta presentación:
  - El comportamiento de rutinas BLAS y LAPACK en diferentes librerías de Álgebra Lineal y diferentes sistemas computacionales con GPUs y MICs

Índice

- NVIDIA GPUs
  - NVIDIA CUBLAS
  - NVIDIA XTCUBLAS
  - MAGMA GPU
- Coprocesadores Intel Xeon Phi (MIC)
  - Intel MKL
  - MAGMA MIC

Experiencias con NVIDIA GPUs. Introducción I

Características

- Hasta 2688 cores CUDA organizados en Streaming Multiprocessor (MP)
- Hasta 6 GBytes GDDR5
- Modelo de programación CUDA
- Funciones paralelas denominadas **kernels**
- Múltiples threads organizados en bloques de theads y grids de bloques

NVIDIA GPUs

Introducción II

Preparación del sistema

Instalación de NVIDIA CUDA Toolkit

- Software que proporciona un entorno de desarrollo en C/C++ y librerías:
  - Compilador CUDA de NVIDIA: NVCC
  - cuFFT: Librería Transformada Rápida de Fourier
  - cuBLAS: Librería BLAS
  - cuSPARSE: librería para Matrices Dispersas
  - cuRAND: Generador de Números Aleatorios
  - etc.

Librerías de Álgebra Lineal para GPUs:

- MAGMA: Matrix Algebra on GPU and Multicore Architectures
- CULA Tools
- IMSL Fortran Numerical Library

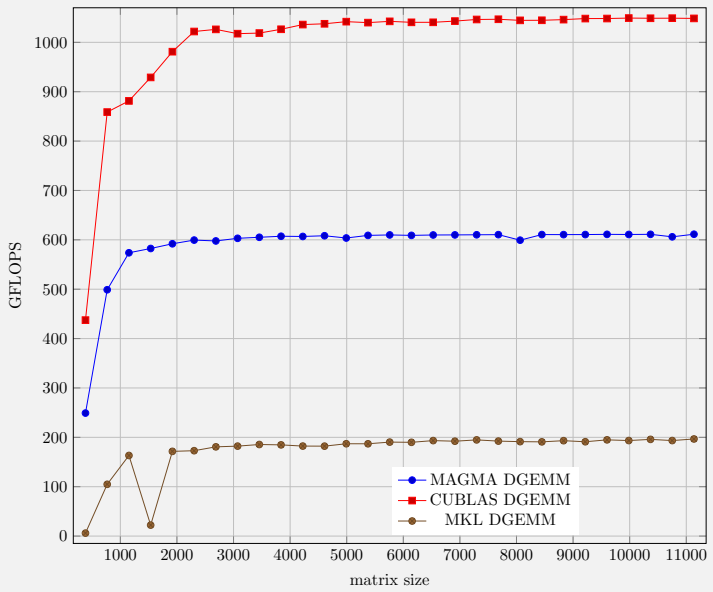
Ejemplo de llamada a rutina en CUBLAS

DGEMM

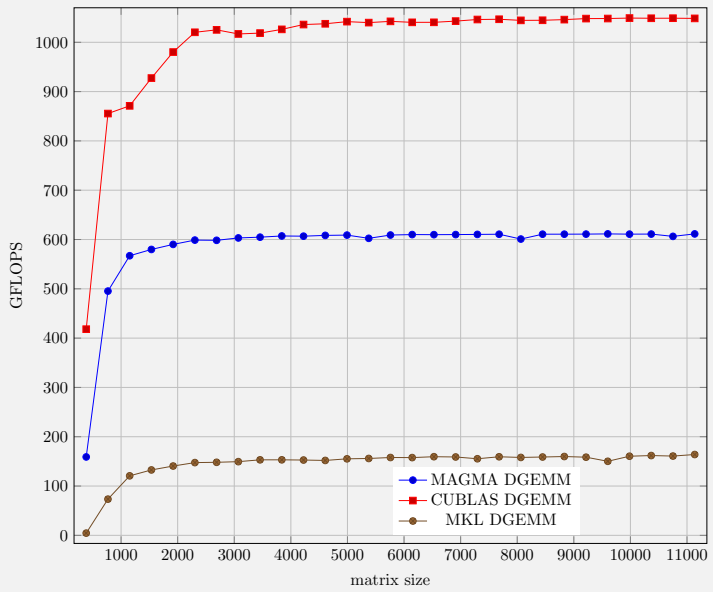
```
h_A = (double *)malloc(sizeof(double)*szeA);
h_B = (double *)malloc(sizeof(double)*szeB);
h_C = (double *)malloc(sizeof(double)*szeC);
...
cudaMalloc((void **) &d_A, sizeof(double)*ldda*K);
cudaMalloc((void **) &d_B, sizeof(double)*lddb*N);
cudaMalloc((void **) &d_C, sizeof(double)*lddc*N);
...
cublasSetMatrix(M, K, sizeof(double), h_A, lda, d_A, ldda);
cublasSetMatrix(K, N, sizeof(double), h_B, ldb, d_B, lddb);

cublasCreate(&handle);
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, M, N, K,
      &alpha, d_A, ldda, d_B, lddb, &beta,  d_C, lddc);

cublasGetMatrix(M, N, sizeof(double), d_C, lddc, h_C, ldc);
```

DGEMM: MAGMA, CUBLAS en GPU y MKL en CPU

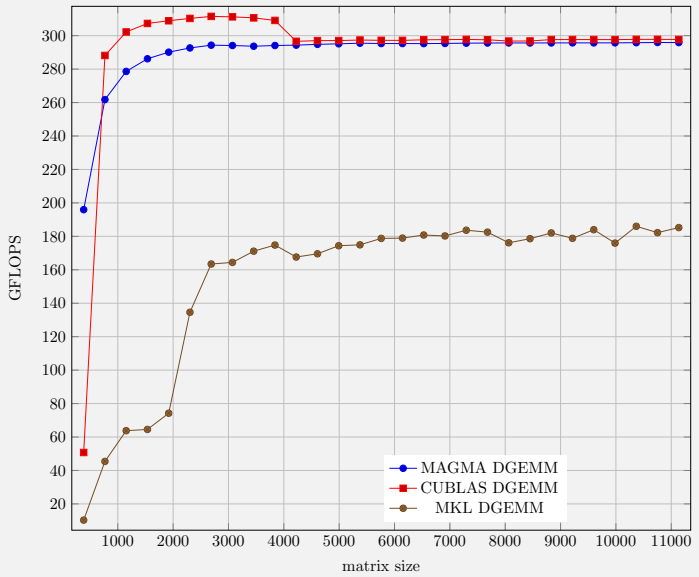
Tesla K20c, 706 MHz, 13 MP x 192 Cores (2496 Cores). Intel Xeon E5-2620 2.00 GHz (12 cores)



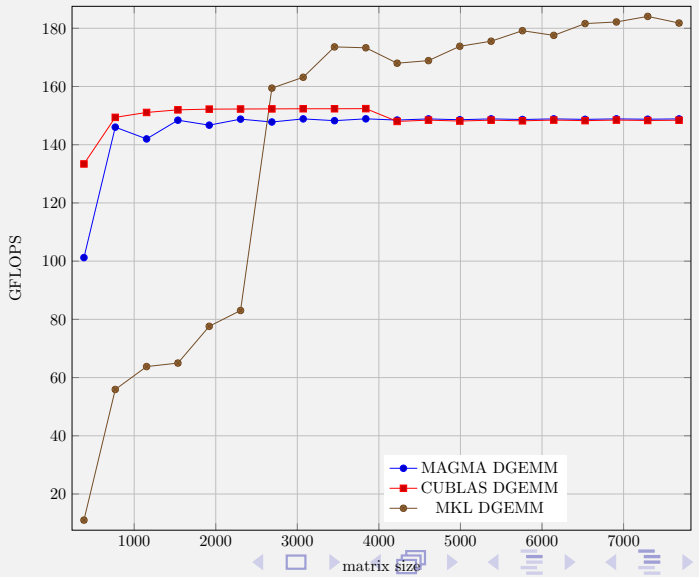
Tesla K20c, 706 MHz, 13 MP x 192 Cores (2496 Cores). Intel Xeon E7530 1.87 GHz (24 cores)



Tesla C2075, 1147 MHz, 14 MP x 32 Cores (448 Cores). Intel Xeon E5-2620 2.00 GHz (12 cores)

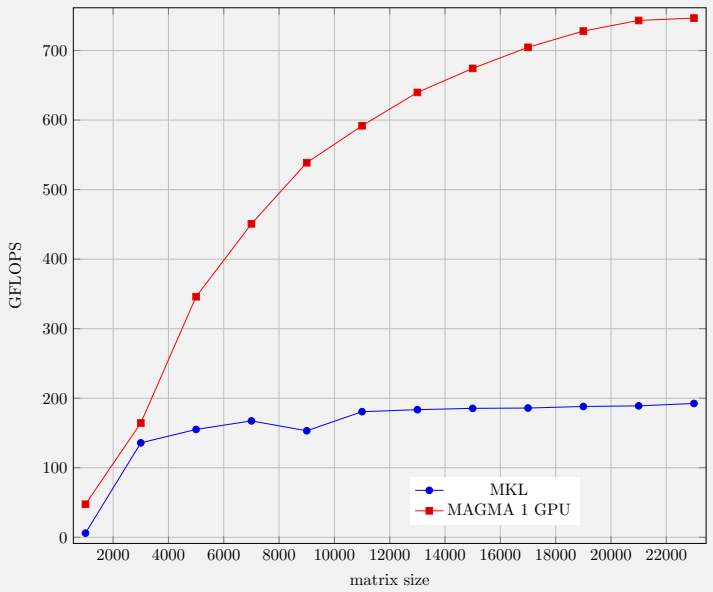


Geforce GTX 590, 1215.0 MHz, 16 MP x 32 Cores (512 Cores). Intel Xeon E5-2620 2.00 GHz (12 cores)

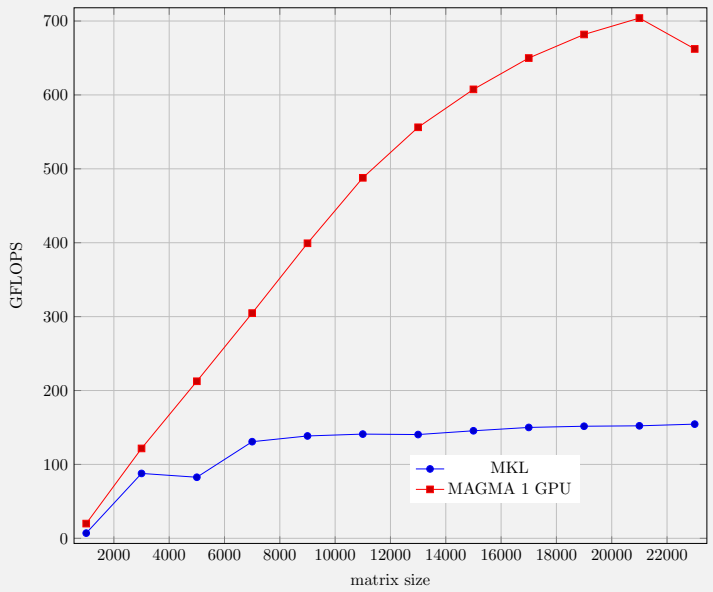


LU: MAGMA GPU+CPU y MKL en CPU

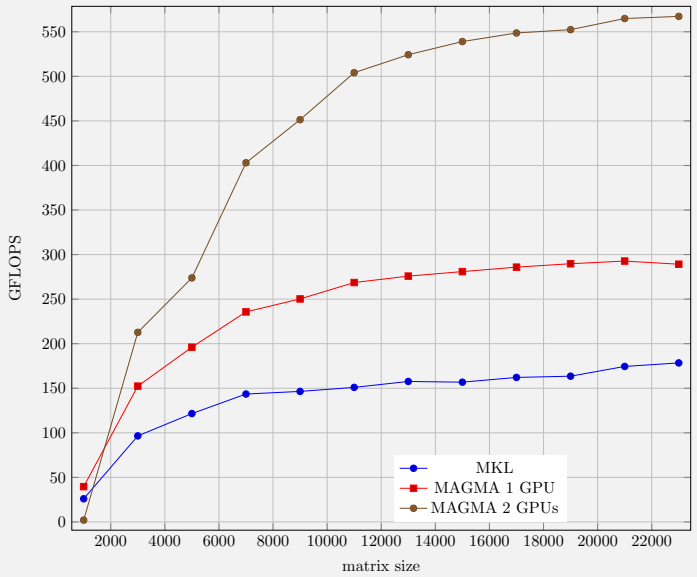
Tesla K20c, 706 MHz, 13 MP x 192 Cores (2496 Cores). Intel Xeon E5-2620 2.00 GHz (12 cores)



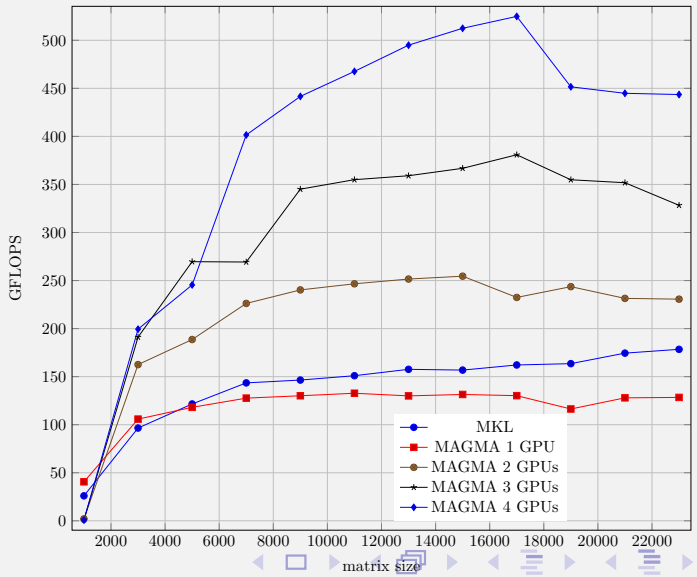
Tesla K20c, 706 MHz, 13 MP x 192 Cores (2496 Cores). Intel Xeon E7530 1.87 GHz (24 cores)



Tesla C2075, 1147 MHz, 14 MP x 32 Cores (448 Cores). Intel Xeon E5-2620 2.00 GHz (12 cores)



Geforce GTX 590, 1215.0 MHz, 16 MP x 32 Cores (512 Cores). Intel Xeon E5-2620 2.00 GHz (12 cores)



Ejemplo de llamada a rutina en MAGMA GPU

LU

```
cublasInit();

magma_malloc_cpu((void **) &h_A,  szeA*sizeof(double) );
magma_malloc_cpu((void **) &ipiv,  M*sizeof(magma_int_t));
magma_malloc((void **) &d_A,  (ldda*M)*sizeof(double) );
...
magma_dsetmatrix( M, M, h_A, lda, d_A, ldda );
magma_dgetrf_gpu( M, M, d_A, ldda, ipiv, &info);
magma_dgetmatrix( M, M, d_A, ldda, h_A, lda);
...

cublasShutdown();
```

Selección tamaño bloque en MAGMA GPU

Función de la GPU y del tamaño del problema, pero no de la CPU

LU

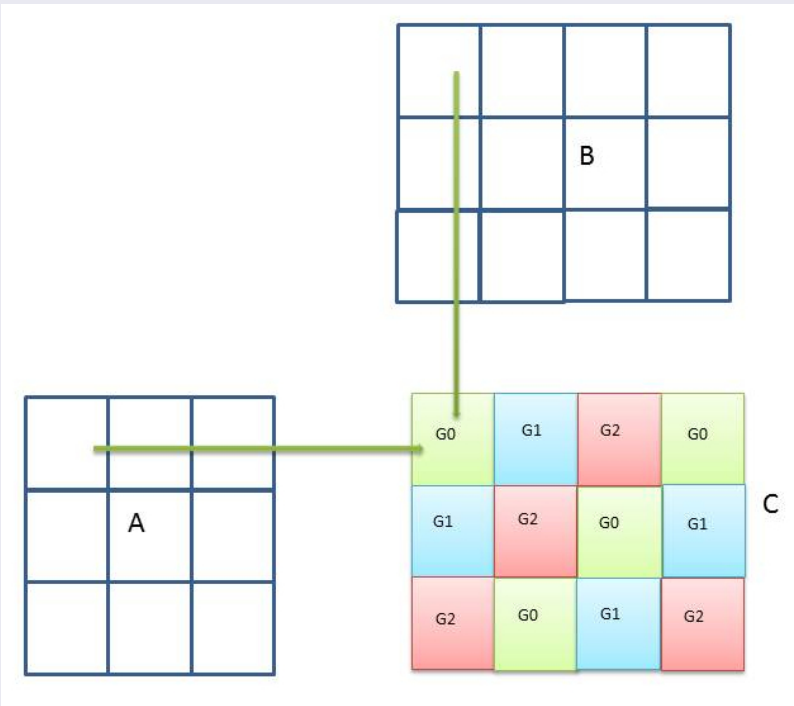
```
magma_int_t arch = magma_getdevice_arch();
    if ( arch >= 300 ) {          // 3.x Kepler
    if      (m <  3072) return 64;
    else if (m < 10240) return 128;
    else                return 256;
}
else {                            // 1.x and 2.x Fermi
    if      (m <  4096) return 64;
    else                return 128;
}
```

Nueva CUBLASXT

Descripción general

La API `cublasXt` proporciona un interfaz **multi-GPU** al host: la aplicación sólo reserva espacio para las matrices en el host. No hay restricción en el tamaño (siempre y cuando quepan en memoria del host). Sólo disponible para rutinas BLAS3.

Esquema cíclico por bloques



Explicación

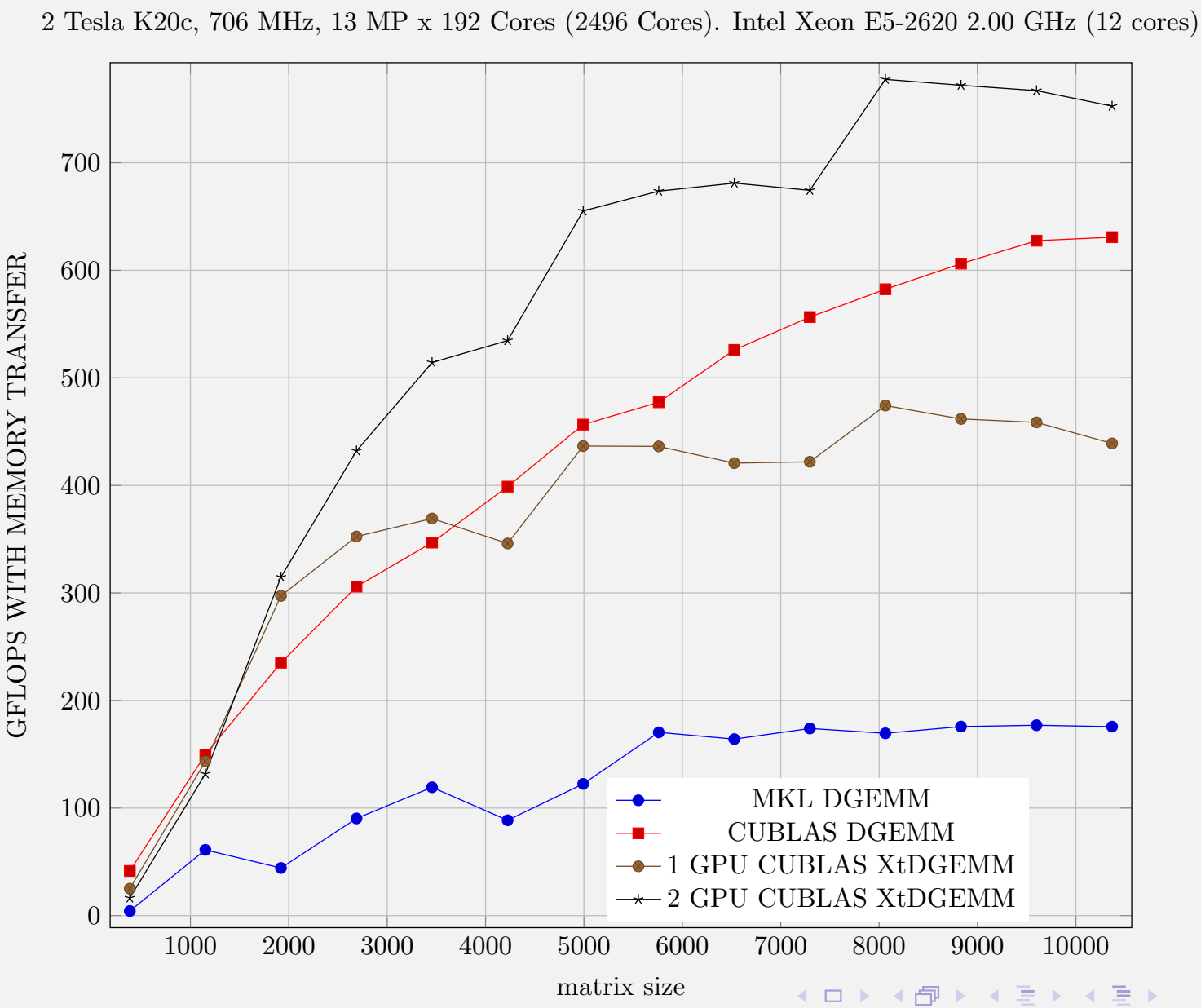
- Las matrices se dividen en bloques cuadrados de tamaño **NB**x**NB**.
- Por cada GPU se crea un **thread de CPU** encargado de las transferencias de memoria necesarias y de lanzar las operaciones cuBLAS requeridas.

Ejemplo de llamada a rutina en XT CUBLAS

XTDGEMM

```
h_A = (double *)malloc(sizeof(double)*szeA);
h_B = (double *)malloc(sizeof(double)*szeB);
h_C = (double *)malloc(sizeof(double)*szeC);
...
// cudaMalloc((void **) &d_A, sizeof(double)*ldda*K);
// cudaMalloc((void **) &d_B, sizeof(double)*lddb*N);
// cudaMalloc((void **) &d_C, sizeof(double)*lddc*N);
...
// cublasSetMatrix(M, K, sizeof(double), h_A, lda, d_A, ldda);
// cublasSetMatrix(K, N, sizeof(double), h_B, ldb, d_B, lddb);

cublasXtCreate(&handle);
cublasXtDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, M, N, K,
      &alpha, h_A, lda, h_B, ldb, &beta,  h_C, ldc);

// cublasGetMatrix(M, N, sizeof(double), d_C, lddc, h_C, ldc);
```

DGEMM: MKL en CPU, CUBLAS y XTCUBLAS



Influencia del tamaño de bloque en XTCUBLAS

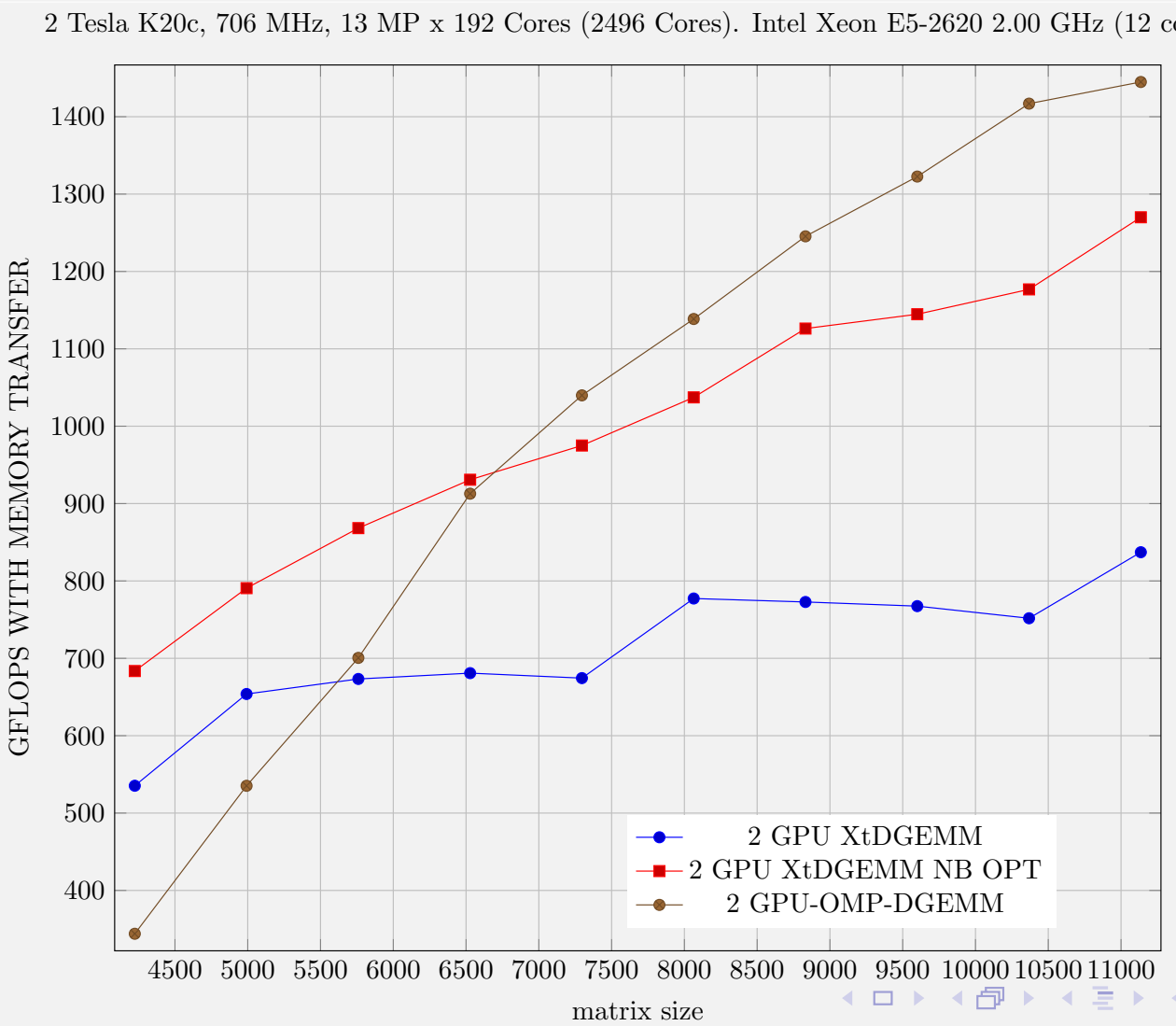
- Prefijado a **1024**
- Se selecciona con la rutina `cublasXtSetBlockDim(cublasXtHandle_t handle, int blockDim)`

Resultados con DGEMM

NB	4224	5760	7296	8832	10368	11136
768	486.98	546.22	585.18	610.99	637.09	642.92
1024	535.31	673.32	674.42	772.81	751.77	837.12
1536	661.07	**868.26**	938.00	1032.65	1077.09	1023.83
2048	658.25	867.30	975.04	1056.46	1018.25	1146.27
2304	**683.57**	857.42	889.12	**1126.21**	1226.5	1243.94
2560	649.76	814.22	**990.51**	1055.60	1144.9	1243.37
2816	615.67	799.93	966.67	960.67	**1176.82**	**1270.02**
3328	542.33	812.37	931.37	1073.27	980.38	1111.05
$\frac{N}{NB}$	1.8	3.8	3.6	3.8	3.7	4.0

DGEMM XTBLAS. INFLUENCIA *NB* y ALGORITMO

- Las prestaciones dependen del tamaño de bloque y del algoritmo.



Nueva NVBLAS

Descripción general

La librería NVBLAS, en aplicaciones que realizan llamadas a BLAS para CPU, es capaz de encaminar dinámicamente esas llamadas a una o más GPUs (usando la API de XTCUBLAS).

Se requiere volver a enlazar

```
icc call_dgemm.o  -o call_dgemm -lnvblas -lmkl_intel_lp64 \
-lmkl_intel_thread -lmkl_core -lpthread -lm -openmp
```

Ejemplo código

```
h_A = (double *)malloc(sizeof(double)*szeA);
h_B = (double *)malloc(sizeof(double)*szeB);
h_C = (double *)malloc(sizeof(double)*szeC);
...
dgemm_("No Transpose", "No Transpose", &M, &N, &K,
        &alpha, h_A, &lda, h_B, &ldb, &beta, h_C, &ldc);
```

Nueva NVBLAS. Configuración

Archivo `nvblas.conf`

```
# Specify which output log file (default is stderr)
NVBLAS_LOGFILE  nvblas.log

#Put here the CPU BLAS fallback Library of your choice
#NVBLAS_CPU_BLAS_LIB  libopenblas.so
NVBLAS_CPU_BLAS_LIB  libmkl_rt.so

# List of GPU devices Id to participate to the computation
# By default if no GPU are listed, only
# device 0 will be used
#NVBLAS_GPU_LIST 0 2 4
#NVBLAS_GPU_LIST ALL
NVBLAS_GPU_LIST ALL0

# Tile Dimension
NVBLAS_TILE_DIM 2048
```

Nueva NVBLAS. Configuración

Archivo `nvblas.conf`

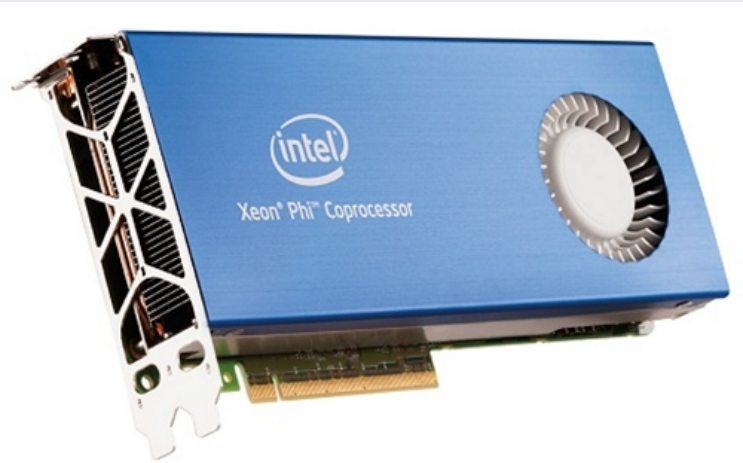
```
# Autopin Memory
NVBLAS_AUTOPIN_MEM_ENABLED

#List of BLAS routines that are prevented from running on GPU
# The current list of BLAS routines supported by NVBLAS are
# GEMM, SYRK, HERK, TRSM, TRMM, SYMM, HEMM, SYR2K, HER2K

#NVBLAS_GPU_DISABLED_DGEMM
#NVBLAS_GPU_DISABLED_ZGEMM

# Computation can be optionally hybridized between CPU and GPU.
# The option NVBLAS_CPU_RATIO_<BLAS_ROUTINE> give the ratio [0,1]

#NVBLAS_CPU_RATIO_DGEMM 0.07
```

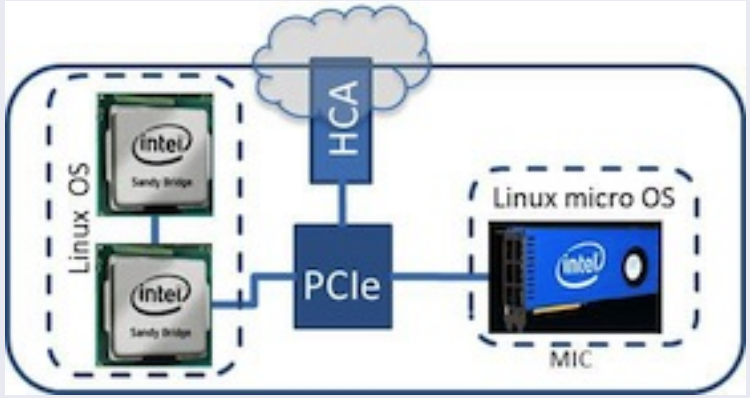
Experiencias con Intel Many Integrated Core (MIC). Introducción I

Características

- Hasta 61 cores (1.0/1.3 GHz), basados en x86 (Pentium) direccionamiento de 64 bits
- Cores con una VPU con 32 registros de 512 bits (SIMD)
- Cada core soporta 4 threads por hardware
- Interconexión: bus en anillo bidireccional
- Hasta 6 GBytes GDDR5

Intel Xeon Phi





Introducción II

Preparación del sistema

- Instalación del Intel MPSS (Manycore Platform Software Stack)
  - Software que permite a nivel de usuario y del sistema que los programas se ejecuten y se comuniquen con el coprocesador Intel Xeon Phi
- Instalación de los Compiladores Offload: Intel C/C++ y Fortran 2013
  - Compiladores capaces de generar código que se ejecutan sólo en el host o sólo en el coprocesador, o parejas de binarios que se ejecutarán tanto en el host como en el coprocesador y se comunicarán entre ellos

Posibilidades de programación paralela

- Las mismas que las disponibles en el sistema host:
  - Intel Threading Building Blocks
  - Intel Cilk Plus
  - pthreads
  - OpenMP
  - MPI

Introducción III

- El coprocesador contiene memoria flash y SMC para almacenar la BIOS, el bootloader, el firmware y un pequeño kernel embebido para el coprocesador.
- Adicionalmente el coprocesador ejecuta un microkernel de Linux.
- El kernel embebido se carga desde la flash cuando el coprocesador se inicializa.
- El microkernel de Linux se carga desde el host cuando el coprocesador se arranca.

```
$ micctrl --status
mic0: online (mode: linux image: /lib/firmware/mic/uos.img)
```

- El coprocesador no dispone de acceso a un sistema de archivos permanente. Todos los sistemas de archivos serán RAM (residente en la memoria del coprocesador) o sistemas de archivos en red (montado vía NFS desde el host).

Introducción IV

- Con código offload no se necesita que el usuario esté dado de alta en el coprocesador. Se utiliza el usuario genérico `micuser`
- Con código navito (aquel que comienza la ejecución en el coprocesador) o MPI se requiere que el usuario pueda acceder al coprocesador. `ssh mic0`
- Durante la instalación del Intel Many Core Platform Software Stack (Intel MPSS). Se copia `/etc/passwd` del host al `/etc/passwd` en el coprocesador y los contenidos de `$HOME./ssh`.

Ejemplo código offload

Reducción secuencial

```
double reduction(double *data, int size)
{
        double ret = 0.0;
        #pragma offload target(mic) in(data:length(size))
        for (int i = 0; i < size; i++) {
             ret += data[i];
        }
       return ret;
}
```

Las claúsulas **in**, **out** e **inout** indican la dirección de los datos que serán transferidos entre el host y el coprocesador.

Ejemplo código offload OpenMP

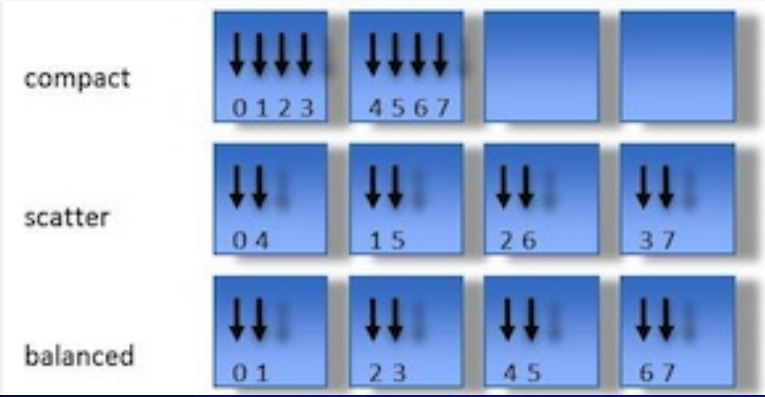
Reducción OpenMP

```
double omp_reduction(double *data, int size)
{
    double ret = 0.0;
    #pragma offload target(mic) in(data:length(size))
    {
        #pragma omp parallel for reduction(+:ret)
        for (int i = 0; i < size; i++) {
            ret += data[i];
        }
    }
    return ret;
}
```

Compilación y ejecución de código nativo

Ejemplo

```
$ icc -mmic -openmp hello_omp.c -o hello_omp.mic
$ scp hello_omp.mic mic0:/tmp
$ scp /opt/intel/composerxe/lib/mic/libiomp5.so mic0:/tmp
$ ssh mic0
$ export LD_LIBRARY_PATH=/tmp
$ export OMP_NUM_THREADS=8
$ export KMP_AFFINITY="verbose,balanced"
$ /tmp/hello_omp.mic
```



Compilación y ejecución de código MPI

Modos de programación MPI

- **Modo sólo coprocesador**: modo de ejecución nativo. El rango de procesos MPI reside únicamente en el coproceador. La aplicación puede ser lanzada desde el host o el coprocesador
- **Modo simétrico**: en este modo el rango de procesos MPI reside en el host y en el coprocesador
- **Modo MPI offload**: El rango de procesos MPI reside únicamente en el host. El rango de procesos MPI utiliza las capacidades offload de los compiladores de Intel para la descarga de trabajo al coprocesador

Ejemplo compilación y ejecución MPI

Copiar las librerías MPI y del Compilador a los coprocesadores

```
# scp /opt/intel/impi/4.1.1.036/mic/bin/* mic0:/bin/
mpiexec                       100% 1118KB   1.1MB/s   00:00
mpiexec.hydra                 100% 1118KB   1.1MB/s   00:00
pmi_proxy                     100%  926KB 926.3KB/s   00:00
...
# scp /opt/intel/impi/4.1.1.036/mic/lib/* mic0:/lib64/
libmpi.a                      100% 8174KB   4.0MB/s   00:02
libmpigf.a                    100%  688KB 687.5KB/s   00:00
libmpigf.so                   100%  321KB 320.6KB/s   00:00
...
# scp /opt/intel/lib/mic/* mic0:/lib64/
libimf.a                      100% 3790KB   3.7MB/s   00:01
libimf.so                     100% 2573KB   2.5MB/s   00:00
libiomp5.so                   100% 1058KB   1.0MB/s   00:00
...
```

Ejemplo compilación y ejecución MPI

Creación ejecutables host y coprocesadores

```
$ mpiicc -mmic hello_mpi.c -o hello_mpi.mic
$ mpiicc hello_mpi.c -o hello_mpi
```

Copia del ejecutable a los coprocesadores

```
$ scp hello_mpi.mic mic0:/tmp/hello_mpi.mic
$ scp hello_mpi.mic mic1:/tmp/hello_mpi.mic
```

Ejemplo compilación y ejecución MPI

Ejecución en host y coprocesadores

```
$ export I_MPI_MIC=enable
$ mpirun -n 4 -host prometeo ./hello_mpi \
: -n 3 -host mic0 /tmp/hello_mpi.mic  \
: -n 4 -host mic1 /tmp/hello_mpi.mic

Hello world: rank 0 of 11 running on prometeo.sait.upct.es
...
Hello World: rank 3 of 11 running on prometeo.sait.upct.es
Hello World: rank 4 of 11 running on prometeo-mic0.sait.upct.es
Hello World: rank 5 of 11 running on prometeo-mic0.sait.upct.es
Hello World: rank 6 of 11 running on prometeo-mic0.sait.upct.es
Hello World: rank 7 of 11 running on prometeo-mic1.sait.upct.es
...
Hello World: rank 10 of 11 running on prometeo-mic1.sait.upct.es
```

Librerías de álgebra lineal: Intel MKL

Modos de utilización en Xeon Phi

- Ejecución Nativa
- Offload asistido por el compilador
- Offload automático

Modo de ejecución nativo

Nos es necesario modificar el código

```
$ icc -O3 -mkl -mmic call_dgemm.c -o call_dgemm.mic
$ scp call_dgemm.mic mic0:/tmp
$ scp /opt/intel/mkl/lib/mic/libmkl_*.so mic0:/tmp
$ ssh mic0
$ cd /tmp
$ export KMP_AFFINITY=balanced
$ export OMP_NUM_THREADS=228
$ export LD_LIBRARY_PATH=/tmp
$ ./call_dgemm.mic 3000 3000 3000
```

Librerías de álgebra lineal: Intel MKL

Modo de ejecución offload asistido por el compilador

La ejecución de código MKL en el coprocesador se controla por directivas.

```
#pragma offload target(mic:0) \
  in(transa, transb, m, n, k, lda, ldb, ldc, d_one,
    d_zero) \
  in(a:length(szeA)) \
  in(b:length(szeB)) \
  in(c:length(szeC)) \
  out(c:length(szeC) alloc_if(0))
  {
        dgemm(&transa, &transb, &m, &n, &k,
                &d_one, a, &lda,
                b, &ldb, &d_zero, c, &ldc);
  }
```

Librerías de álgebra lineal: Intel MKL

Compilación MKL con offload asistido por el compilador

```
$ icc -O3 -openmp -mkl \
  -offload-option,mic,ld, "-L$MKLROOT/lib/mic -Wl,\
  --start-group -lmkl_intel_lp64 -lmkl_intel_thread \
  -lmkl_core -Wl,--end-group" call_dgemm.c -o call_dgemm
```

Ejecución

```
$ export MIC_ENV_PREFIX=MIC
$ export MIC_KMP_AFFINITY=balanced
$ export MIC_OMP_NUM_THREADS=224
$ export MIC_USE_2MB_BUFFERS=64K
$ ./call_dgemm 3000 3000 3000
```

Librerías de álgebra lineal: Intel MKL

Modo de ejecución offload automático

Nos es necesario modificar el código. MKL decide cuando **offload** código al coprocesador y la división óptima de trabajo entre el host y el coprocesador. Para rutinas BLAS se puede especificar la división con `mkl_mic_set_Workdivision(MKL_TARGET_MIC, 0, 0.5)`.

```
$ icc -O3 -mkl call_dgemm.c -o call_dgemm
$ export MKL_MIC_ENABLE=1
$ export OFFLOAD_DEVICES=0,1
$ export MKL_MIC_MAX_MEMORY=2GB
$ export MIC_ENV_PREFIX=MIC
$ export MIC_OMP_NUM_THREADS=224
$ export MIC_KMP_AFFINITY=balanced
$ export OFFLOAD_REPORT=2
$ ./call_dgemm 3000 3000 3000
```

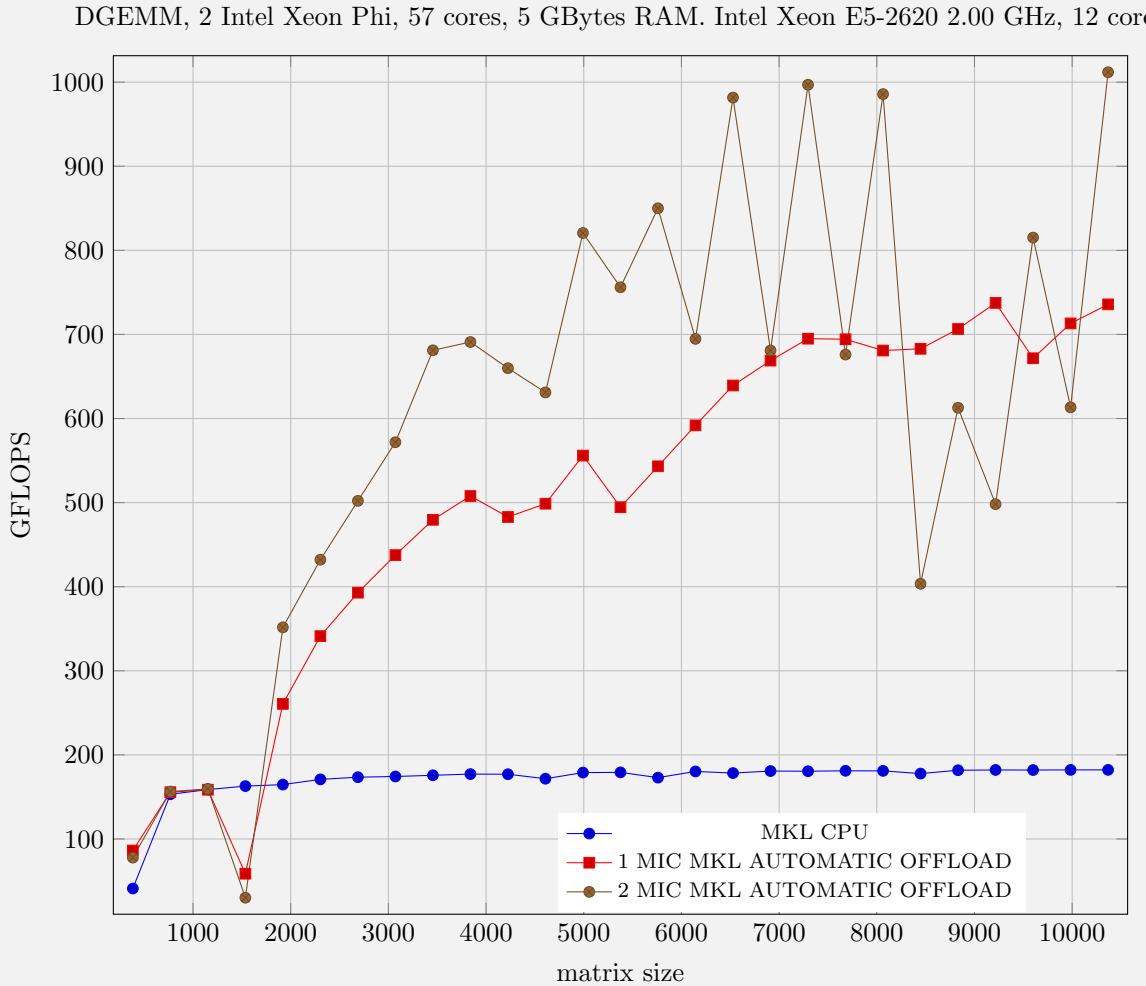
Librerías de álgebra lineal: Intel MKL

Salida modo de ejecución offload automático

```
$ ./call_dgemm 3000 3000 3000
[MKL] [MIC --] [AO Function] DGEMM
[MKL] [MIC --] [AO DGEMM Workdivision] 0.34 0.66

[MKL] [MIC 00] [AO DGEMM CPU Time] 4.116438 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time] 0.239395 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 123840000 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 51840000 bytes
 3000  3000  3000 4.11707 segundos
```

MKL DGEMM en Intel Phi

- Ejecución en modo OFFLOAD Automático con 1 y 2 Intel Phi



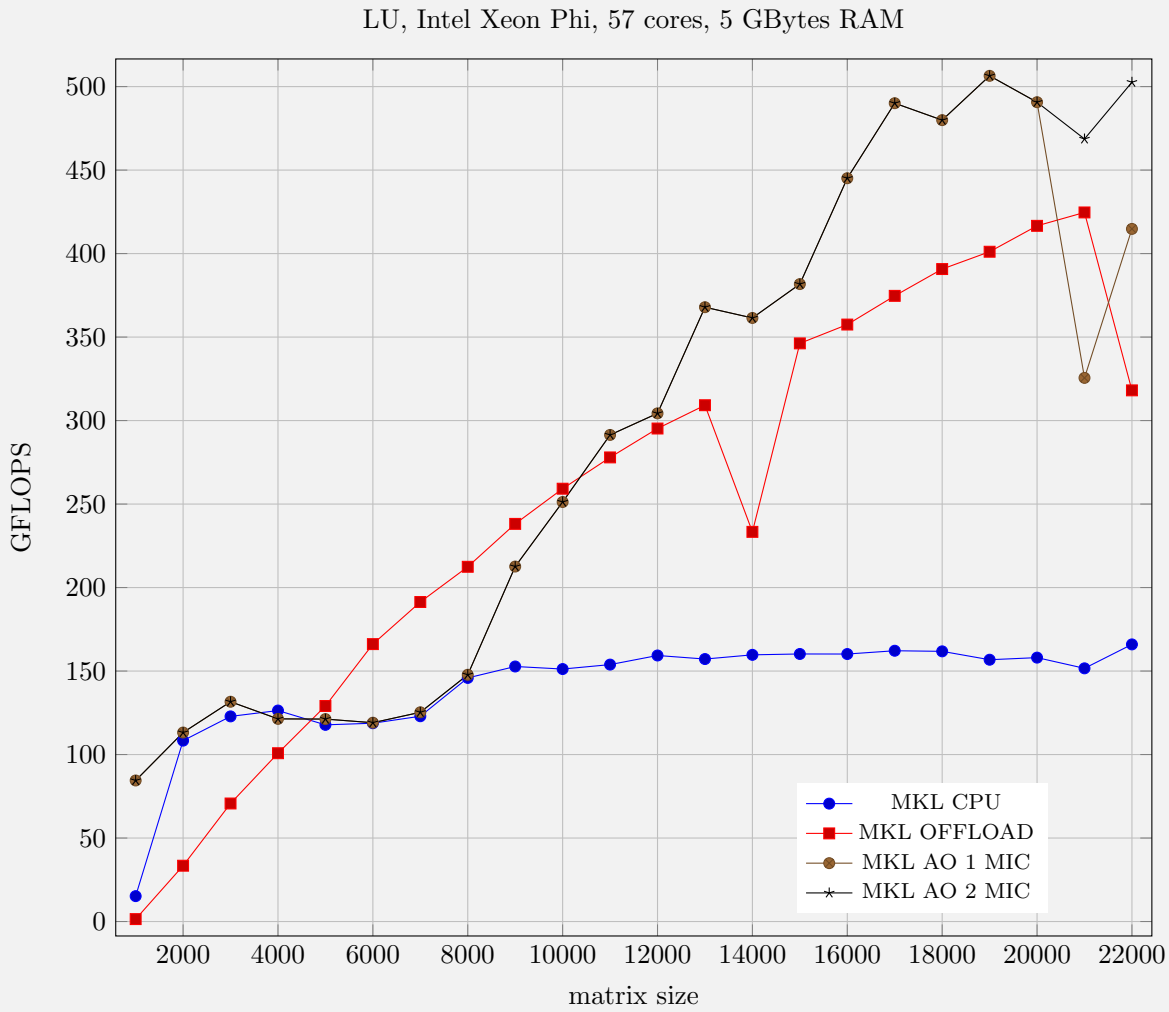
MKL DGEMM en Intel Phi

Intel AO WORKSIZE DIVISION

	DIVISION				
size	1 MIC		2 MIC		
1152					
1536	0.53	0.47	0.45	0.27	0.27
2304	0.37	0.63	0.26	0.37	0.37
3072	0.33	0.67	0.22	0.39	0.39
3840	0.31	0.69	0.20	0.40	0.40
4608	0.32	0.68	0.22	0.39	0.39
5376	0.29	0.71	0.19	0.41	0.41
6144	0.27	0.73	0.17	0.42	0.42
6912	0.25	0.75	0.15	0.42	0.42
7680	0.24	0.76	0.14	0.43	0.43
8448	0.24	0.76	0.14	0.43	0.43
9216	0.23	0.77	0.13	0.43	0.43
10368	0.23	0.77	0.12	0.44	0.44

MKL LU en Intel Phi

- Ejecución en modo OFFLOAD y AO con 1 y 2 Intel Phi



Librerías de álgebra lineal: MAGMA

Open Source Software `http://icl.cs.utk.edu/magma` : J. Dongarra et al. Universidad of Tennessee, Universidad de California, Universidad de Colorado, INRIA/Francia y KAUST/Arabia Saudí.

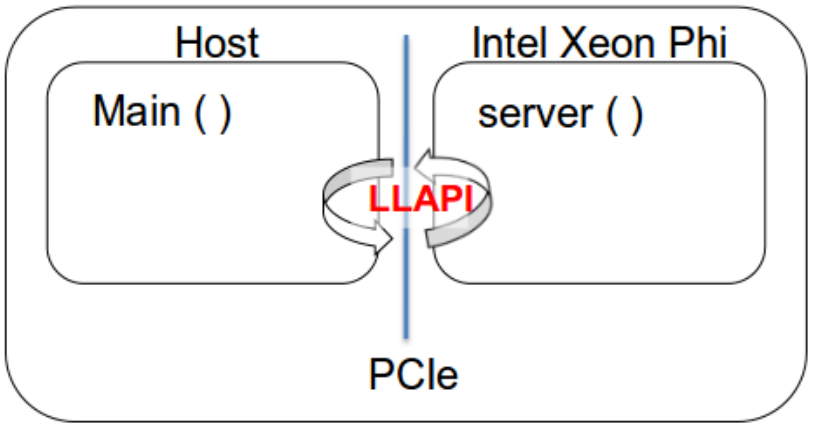
Consideraciones

- No hay una API de alto nivel similar a CUDA/OpenCL que facilite el uso del coprocesador desde el host. (OpenCL 1.2 está ya disponible en el Intel SDK 2013).
- Hay un conjunto de directivas para **offload** de código, pero son de muy alto nivel y en principio poco apropiadas para el desarrollo de una librería numérica de Altas Prestaciones.
- Uso de LLAPI (Low Level API) y SCIF para Intel Xeon Phi.

Librerías de álgebra lineal: MAGMA

Modelo de programación

- En el coprocesador Intel Xeon Phi, MAGMA se ejecuta como un "servidor"
- Las comunicaciones entre el host y el coprocesador están implementadas utilizando LLAPI y SCIF
- Mismo interface que en MAGMA GPU



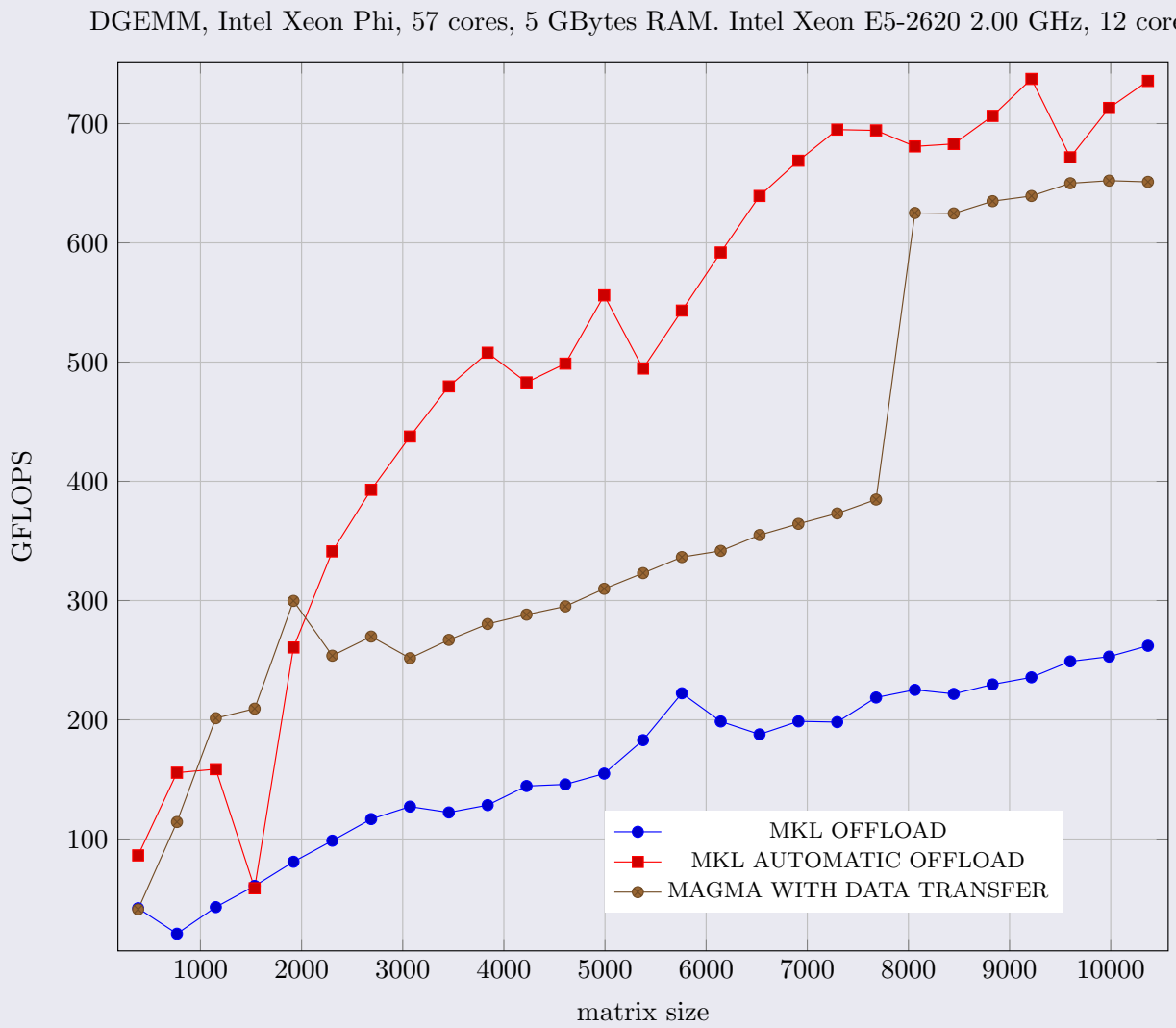
Ejemplo de llamada a rutina en MAGMA MIC

LU

```
magma_init();
err = magma_get_devices( &device, 1, &num );
err = magma_queue_create( device, &queue );

magma_malloc_host( h_A,  double, szeA );
magma_malloc_host( ipiv,  magma_int_t, M );
magma_malloc( d_A, double, ldda*M );
...
magma_dsetmatrix( M, M, h_A, 0, lda, d_A, 0, ldda, queue );

ret = magma_dgetrf_mic( M, M, d_A, 0, ldda, ipiv, &info, queue );

magma_dgetmatrix( M, M, d_A, 0, ldda, h_A, 0, lda, queue );
...
magma_queue_destroy( queue );
magma_finalize();
```

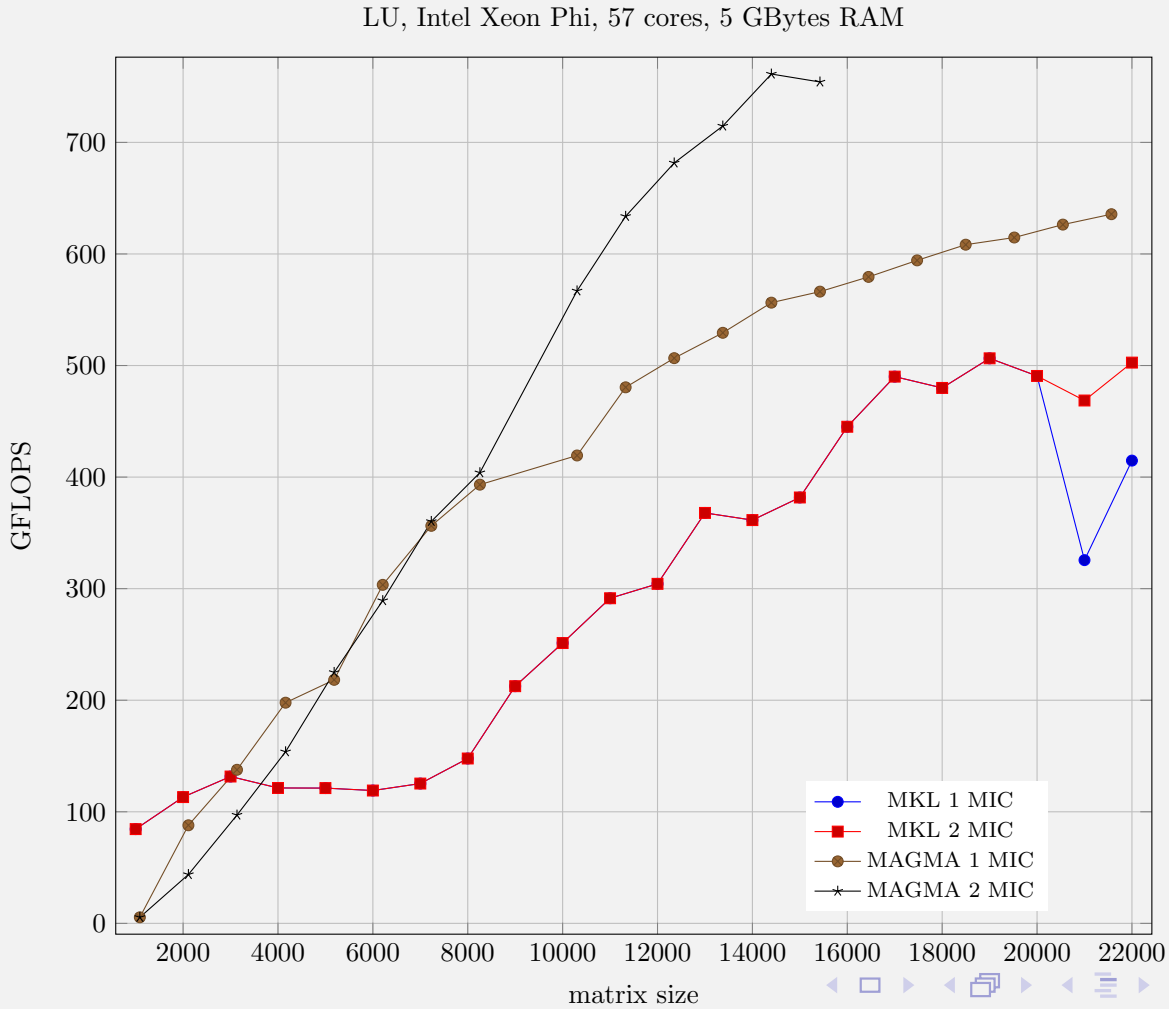
MAGMA y MKL Intel Phi

Comparativa DGEMM Intel MKL y MAGMA



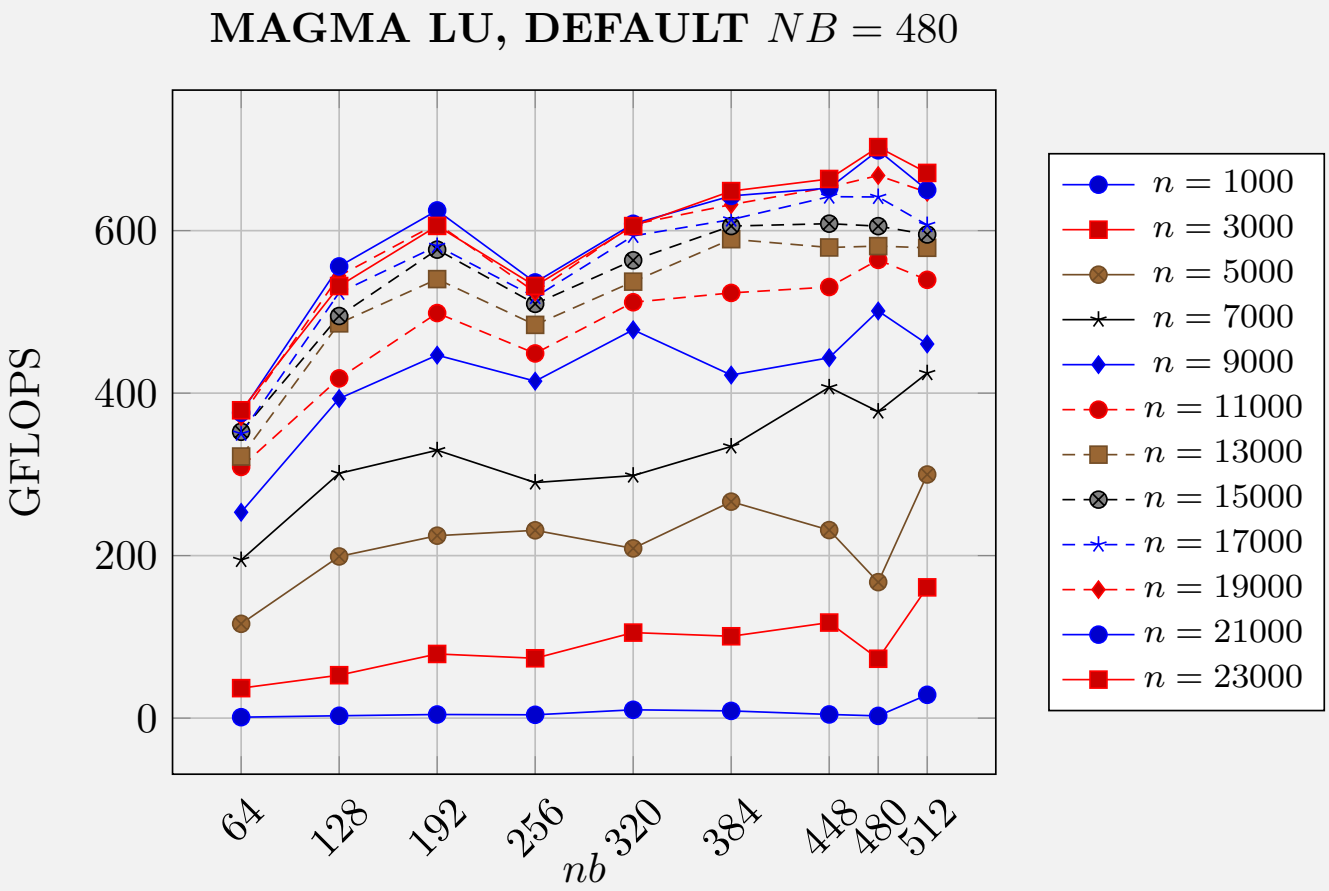
LU MAGMA y MKL Intel Phi

- Parámetros ajustables. **MKL**: número de threads, división del trabajo en rutinas AO. **MAGMA**: tamaño de bloque y número de threads



MIC LU MAGMA. INFLUENCIA *NB*

- En la versión de MAGMA instalada el valor por omisión para el *NB* es siempre de 480.



Ejecución de trabajos en GPU o MIC

```
#!/bin/bash
#SBATCH --job-name="SLURM TEST JOB"
#SBATCH --nodes=1
#SBATCH --partition=manycore
#SBATCH --gres=mic:2
#SBATCH --time=01:00:00
#SBATCH --output=$HOME/slurm-OUT.log
#SBATCH --error=$HOME/slurm-ERR.log
#======START=============================================
# Load your required module files here
module load intel/2013
#module load magma/mic/1.3.0

export MKL_MIC_ENABLE=1
export OFFLOAD_DEVICES=0,1
export MKL_MIC_MAX_MEMORY=2GB
export MIC_ENV_PREFIX=MIC
export MIC_OMP_NUM_THREADS=224
export MIC_KMP_AFFINITY=balanced

cd $HOME/code/mic/mkl

./ao_dgetrf 1000 22000 1000 10
#======END===============================================

$ sbatch simple-mic-offload.sh
```

Ejecución de trabajos en MIC en modo nativo

- Reservar nodos con Xeon Phis. Inicializar Xeon Phi
- Conectarse al nodo reservado
- Iniciar sesión (ssh) en el Xeon Phi

```
$ salloc -N1 --gres=mic:2
$ /etc/slurm/init_mic.sh
$ scontrol show job [JOBID]
$ ssh nombre-nodo
$ scp dgemm_gpu.mic mic0:/tmp
$ ssh mic0
$ /tmp/dgemm_gpu.mic

$ salloc -N1 --gres=mic:2
$ /etc/slurm/init_mic.sh
$ mpirun -np 1 hello_word.mic : -np 1 hello_word
```