ICHEC
Irish Centre for High-End Computing

PRACE Winter School 2012

Hybrid Programming on
Massively Parallel Architectures

CUDA Libraries and MPI+OpenMP+CUDA

Filippo Spiga, Computational Scientist

filippo.spiga@ichec.ie - http://filippospiga.me

sfi Science Foundation Ireland

Ireland's EU Structural Funds Programmes 2007 - 2013

Co-funded by the Irish Government and the European Union

EUROPEAN REGIONAL DEVELOPMENT FUND

AN RÍNN OIDEACHAIS AGUS EOLAÍOCHTA | DEPARTMENT OF EDUCATION AND SCIENCE

HEA
Higher Education Authority
An tÚdarás um Ard-Oideachas

Outline

- CUDA Libraries Ecosystem
  - use-case: compute 3D-FFT on multi-GPU
- Mixing CUDA and FORTRAN easily
- Consideration over pageable/non-pageable memory
  - use-case: PHIGEMM library
- Performance Measurement and Metrics

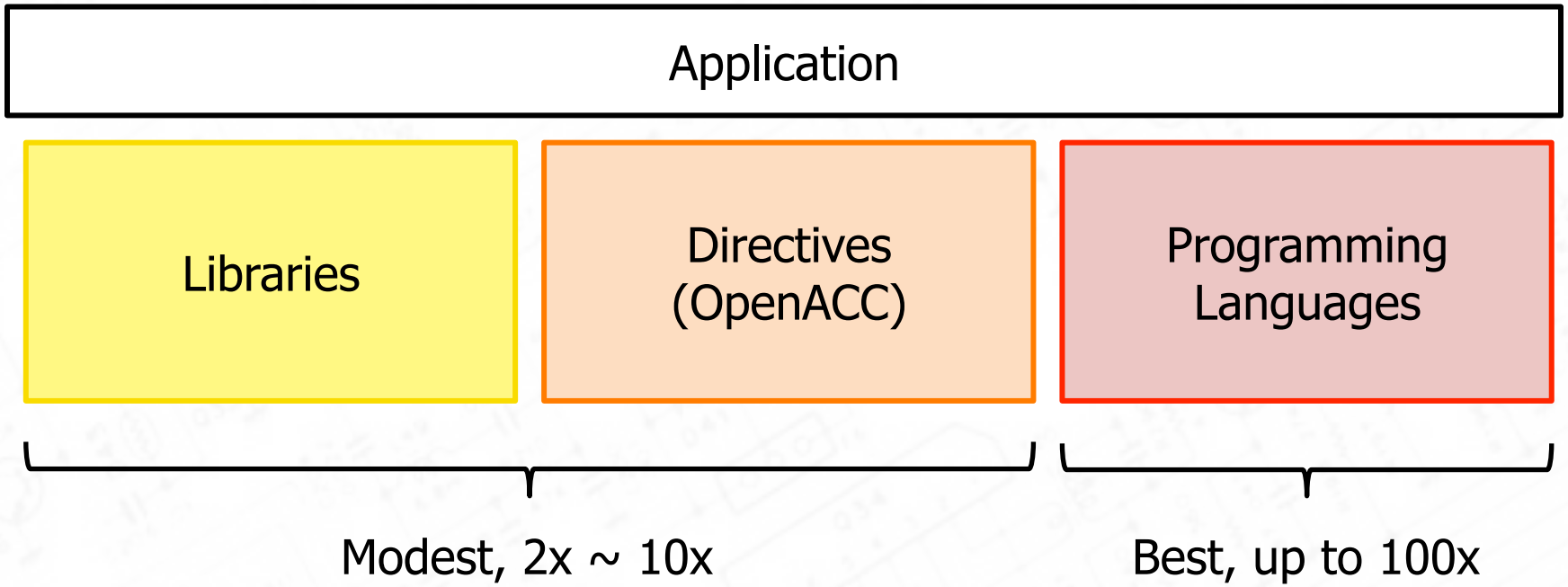
- Recap about CUDA+MPI
- MPI+OpenMP+OpenMP: why, when, how

- Use-case: GPU-accelerated Quantum ESPRESSO (PWSCF)
  - one-by-one kernel *performance-driver* analysis
  - consideration about overall performance, serial and parallel

Acknowledgments first!

- ICHEC, Irish Centre for High-End Computing
- Ivan Girotto, Computational Scientists - ICHEC
- CINECA, Consorzio Interuniversitario
- Marzia Rivi, HPC Specialist - CINECA
- M. Fatica, P. Wang and C. Woolley - NVIDIA Corporation
- Dhabaleswar K. (DK) Panda - Ohio State University
- Rob Farber, former ICHEC Research Visitor
- P. Giannozzi, L. Marsamos, C. Cavazzoni - Quantum ESPRESSO
- Philip Yang, former ICHEC Summer Intern

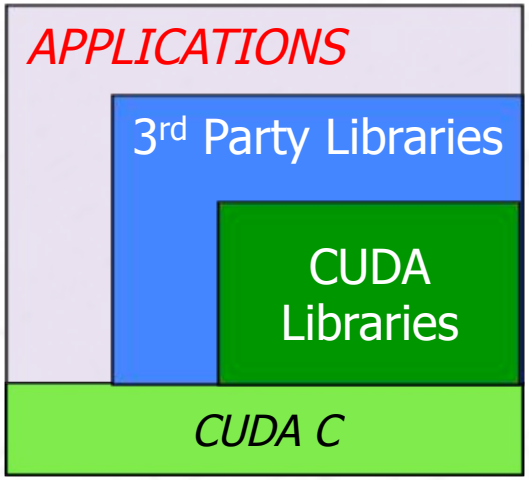
3 Ways to Accelerate on GPU



CUDA library ecosystem

NVIDIA CUDA Toolkit Libraries

CUBLAS
CUSPARSE
CUFFT
CURAND
NPP
Thrust
math.h
system calls



3rd Party Libraries

MAGMA
CULA
libJacket
CUSP
IMSL Library
NAG*
OpenVidia
OpenCurrent

CUFFT - Fast Fourier Transform library

- Algorithms based on Cooley-Tukey ($n = 2a \cdot 3b \cdot 5c \cdot 7d$) and Bluestein
- Simple interface similar to FFTW
- 1D, 2D and 3D transforms of complex and real data
- Row-major order (C-order) for 2D and 3D data
- Single precision (SP) and Double precision (DP) transforms
- In-place and out-of-place transforms
- Batch execution for doing multiple transforms
- Streamed asynchronous execution
- Non normalized output: IFFT(FFT(A))=len(A)*A
- Major constrains for better performance (single power, length multiplies)

CUFFT – Code Sample (C)

```
#define NX 256
#define NY 128
cufftHandle plan;
cufftComplex *idata, *odata;

cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);

/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX,NY, CUFFT_C2C);
/* Use the CUFFT plan, transform out of place. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
/* Inverse transform in place. */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(idata); cudaFree(odata);
```

CUBLAS – Basic Linear Algebra Library

- Self-contained at the API level, same name convention of BLAS
- Supports all the BLAS functions:
  - Level 1 (vector-vector, O(N)): AXPY, DOT
  - Level 2 (matrix-vector, $O(N^2)$):GEMV, TRSV
  - Level 3 (matrix-matrix, $O(N^3)$): GEMM, TRSM
- Following BLAS convention, column-major storage
- Error handling
- FORTRAN interfaces are **Thunking** and **non-Thunking** (default)
- Support streams and asynchronous calls
- Batched GEMM (small matrices)

CUBLAS - Thunking versus non-Thunking

Thunking:

- Allows interfacing to existing applications without any changes → wrappers
- During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPGPU memory
- Intended for light testing, call overhead

Non-Thunking (default):

- Existing applications need to be modified slightly to allocate/deallocate data the in GPGPU memory space (using `CUBLAS_ALLOC` and `CUBLAS_FREE`) and to copy data between GPU and CPU memory spaces (using `CUBLAS_SET_VECTOR`, `CUBLAS_GET_VECTOR`, `CUBLAS_SET_MATRIX`, and `CUBLAS_GET_MATRIX`)
- Intended for production code, high flexibility

CUBLAS – Code Sample (FORTRAN)

```
PROGRAM example_sgemm

! Define 3 single precision matrices A, B, C
REAL, dimension(:,:), allocatable:: A(:,:), B(:,:), C(:,:)
INTEGER :: n = 100

ALLOCATE(A(n,n), B(n,n), C(n,n))

! Initialize A, B and C
...

! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
! memory allocation on device and data movement)
CALL cublas_SGEMM('n','n', n,n,n,1.,A,n,B,n,1.,C,n)

!  De-allocate A, B and C
...

END PROGRAM example_sgemm
```

Thunking

CUBLAS – Code Sample (FORTRAN)

```
PROGRAM example_sgemm
REAL, dimension(:,:), allocatable:: A(:,:),B(:,:),C(:,:)
INTEGER*8 :: devPtrA, devPtrB, devPtrC
INTEGER:: n=16, size_of_real=16
! Allocate
ALLOCATE(A(n,n),B(n,n),C(n,n))
CALL cublas_Alloc(n*n,size_of_real, devPtrA)
…
! Copy data to GPU
CALL cublas_Set_Matrix(n,n,size_of_real,A,n,devPtrA,n)
…
! Call SGEMM in CUBLAS library
CALL cublas_SGEMM('n','n', n,n,n,1.,devPtrA,n,devPtrB,n,1.,devPtrC,n)
! Copy data from GPU call
CALL cublas_Get_Matrix(n,n,size_of_real,devPtrC,n,C,n)
! Deallocate
CALL cublas_Free(devPtrA)
…
END PROGRAM example_sgemm
```

Non-Thunking

Thrust

- A template library for CUDA — Mimics the C++ STL
- Containers: manage memory on host and device:
  `thrust::host_vector<T>, thrust:device_vector<T>`
- Define ranges: `d_vec.begin()`, support Iterators
- Hides/Simplify data movement: know where data lives
- Algorithms acting on ranges and support general types and operators:
  - Element-wise operations*: for_each, transform, gather, scatter,...*
  - Reduction: *reduce, inner_product, reduce_by_key, ...*
  - Prefix-Sums: *inclusive_scan, inclusive_scan_by_key, ...*
  - Sorting: *sort, stable_sort, sort_by_key*

use-case: compute 3D-FFT on multi-GPU

Example inside the book "*CUDA Application Design and Development*" (Rob Farber)

GOAL:

- batch multiple 3D-FFT across multiple GPU in the same host/workstation
- Measure the time spent for computation and data transfer through profiling (*is there overlapping?*)

DESIGN:

- C++ & Templates
- Thrust for data manipulation (we do not focus on it)
- Multi-plan cuFFT wrapped in a separate C++ class

fft3Dtest.cu – Code Snippet (C++)

```
cudaStream_t streams[nGPU];
for(int sid=0; sid < nGPU; sid++) {
     cudaSetDevice(sid);
     if(cudaStreamCreate(&streams[sid]) != 0) {
          cerr << "Stream create failed!" << endl;
     }
}

cudaSetDevice(0);
if(cudaHostAlloc(&h_A, nelements*sizeof(REAL), cudaHostAllocPortable)
     != cudaSuccess) {
     cerr << "cudaHostAlloc failed!" << endl; exit(1);
}

#pragma parallel for
for(long i=0; i < nelements; i++)  h_A1[i] = h_A[i] = i%n2ft3d;

DistFFT3D<REAL> dfft3d(nGPU, h_A, nelements, dim, nPerCall,  streams);
```

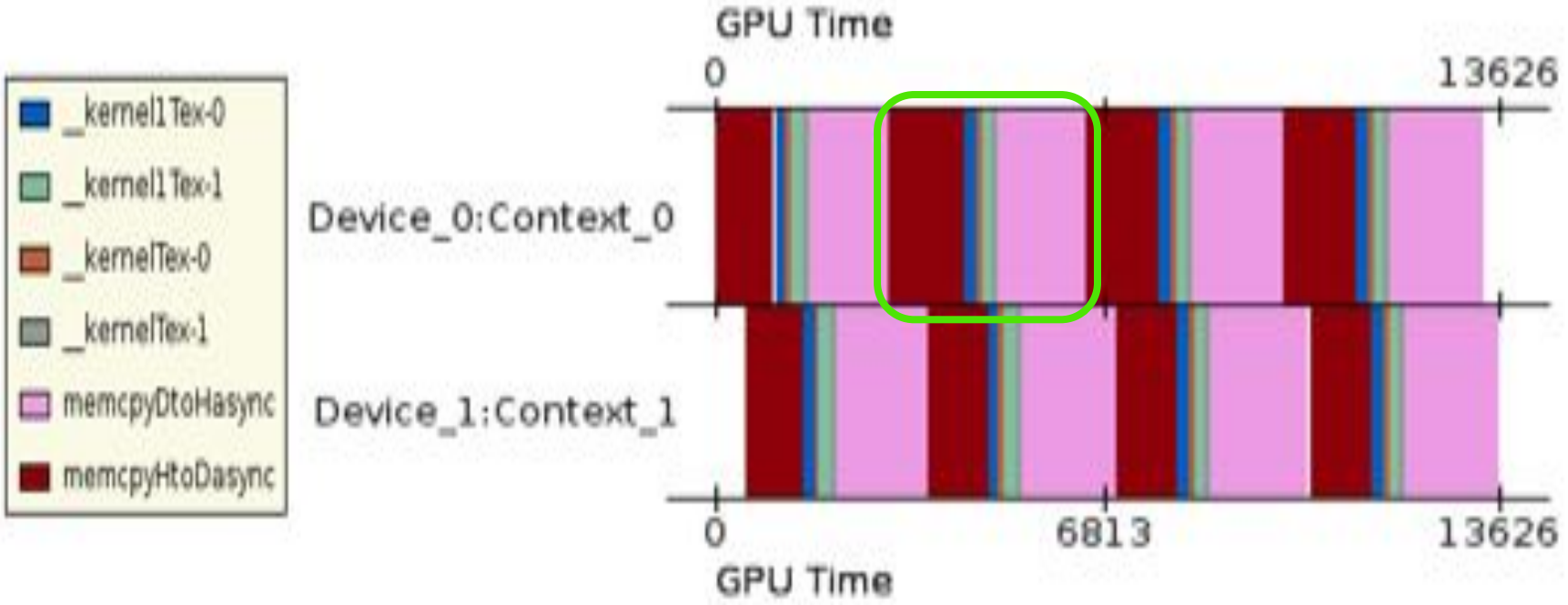
Courtesy of R. Farber

fft3Dtest.cu – Code Snippet (C++)

```
long h_offset=0;for(int i=0; i < totalFFT; i += nGPU*nPerCall) {
     for(int j=0; j < nGPU; j++) {
          cudaSetDevice(j);

          cudaMemcpyAsync(d_data[j], ((char*)h_data)+h_offset,
               bytesPerGPU, cudaMemcpyDefault,streams[j]);

          cforwardFFT_(fftPlanMany[j],d_data[j], d_data[j]);
          cinverseFFT_(fftPlanMany[j],d_data[j], d_data[j]);

          cudaMemcpyAsync(((char*)h_data)+h_offset, d_data[j],
               bytesPerGPU, cudaMemcpyDefault,streams[j]);
          h_offset += bytesPerGPU;
     }
}

cudaDeviceSynchronize();
cudaSetDevice(0);
```

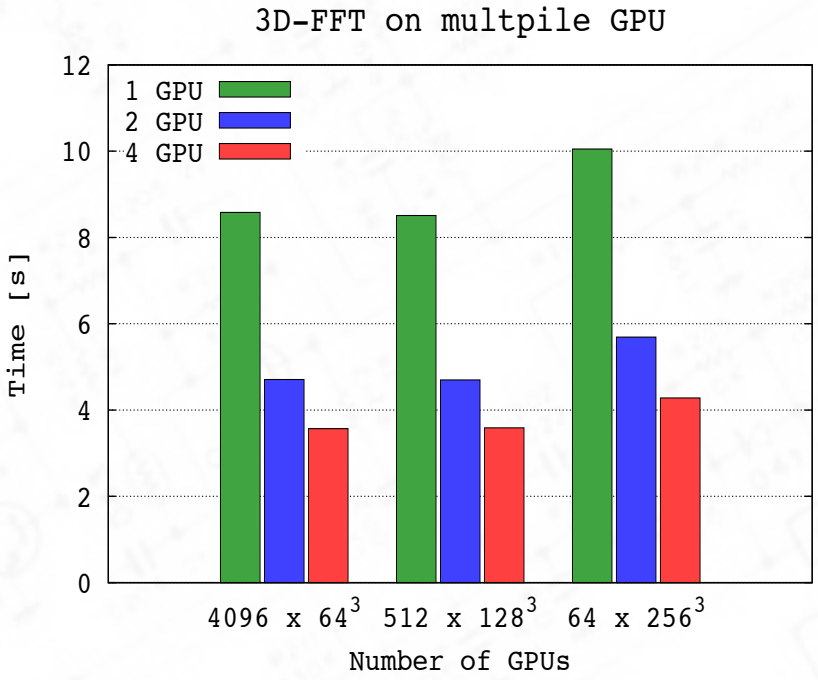
Courtesy of R. Farber

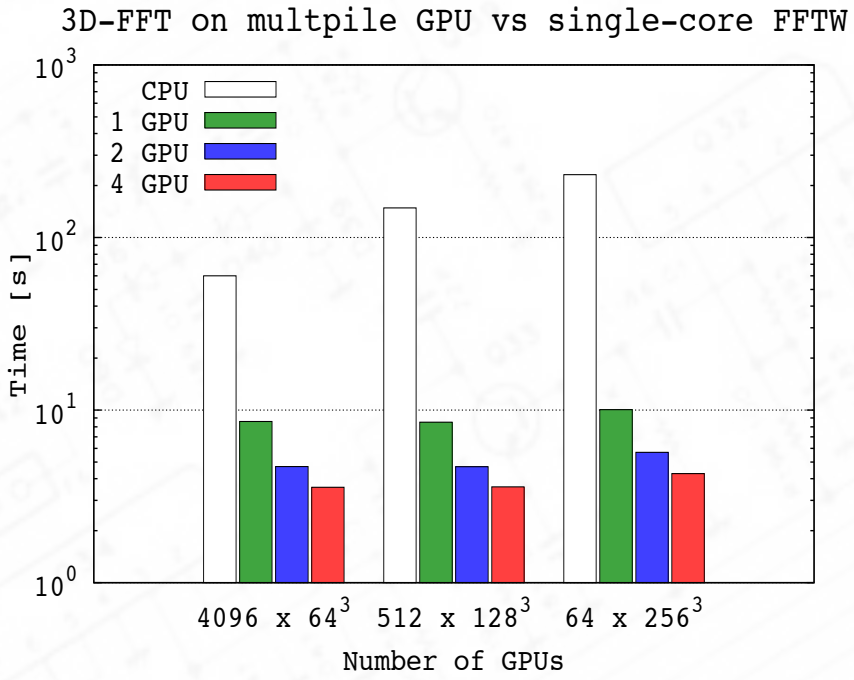
fft3Dtest.cu - Profiling



Courtesy of R. Farber

fft3Dtest.cu - Profiling





Courtesy of R. Farber

Mixing CUDA and FORTRAN

Q: Can I call CUDA library routines directly in a FORTRAN program?

A: Yes but keep in mind the data order convention!

Q: Can I wrote a CUDA kernel in FORTRAN?

A: No… but CUDA Fortran exists (PGI)

Q: Can I call CUDA runtime APIs in FORTRAN?

A: Yes and No… but CUDA Fortran exists (PGI)

Q: So, how I can perform a simple pinned allocation of in FORTRAN?

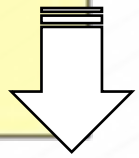
A: …

Pinned allocation - Code Sample (FORTRAN)

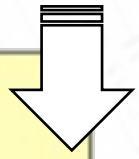
```
MODULE cuda_mem_alloc
     INTERFACE
     ! cudaMallocHost
     INTEGER (C_INT) function cudaHostAlloc(buffer, size, flag) &
               bind(C,name="cudaHostAlloc")
          MODULE iso_c_binding
          IMPLITICT NONE
          TYPE(C_PTR)  :: buffer
          INTEGER(C_SIZE_T), VALUE :: size
          INTEGER(C_SIZE_T), VALUE :: flag
     END FUNCTION cudaHostAlloc
     ! cudaFreeHost
     INTEGER (C_INT) function cudaFreeHost(buffer)  bind(C,name="cudaFreeHost")
          MODULE iso_c_binding
          IMPLITICT NONE
          TYPE(C_PTR), VALUE :: buffer
     END FUNCTION cudaFreeHost
     END INTERFACE
END MODULE cuda_mem_alloc
```

Pinned allocation - Code Sample (FORTRAN)

```
#define N 1000
#define M 1000

USE iso_c_binding
USE cuda_mem_alloc

INTEGER, PARAMETER   ::   singlePrecision = kind(0.0)
INTEGER, PARAMETER   ::   fp_kind_single       = singlePrecision

INTEGER, PARAMETER   ::   doublePrecision      = kind(0.0d0)
INTEGER, PARAMETER   ::   fp_kind_double       = doublePrecision

INTEGER :: res
INTEGER (C_SIZE_T), PARAMETER :: test_flag = 0
INTEGER (C_SIZE_T) :: allocation_size
```



Pinned allocation - Code Sample (FORTRAN)



```
REAL(fp_kind_single), dimension(:,:), pointer :: h_A ! REAL
REAL(fp_kind_double), dimension(:,:), pointer :: h_B ! REAL DOUBLE PRECISION
REAL(fp_kind_single), dimension(:,:), pointer :: h_C ! COMPLEX
REAL(fp_kind_double), dimension(:,:), pointer :: h_D ! COMPLEX DOUBLE PRECISION

TYPE(C_PTR) :: cptr_A, cptr_B, cptr_C, cptr_D

allocation_size = M*N*SIZEOF(fp_kind_single)
res = cudaHostAlloc (cptr_A, allocation_size, test_flag )
CALL c_f_pointer ( cptr_A, h_A, (/ M, N/) )

allocation_size = M*N*SIZEOF(fp_kind_double)*2
res = cudaHostAlloc (cptr_D, allocation_size, test_flag )
CALL c_f_pointer ( cptr_D, h_D, (/ M, N/)

! Deallocate properly
res = cudaFreeHost( cptr_A )
res = cudaFreeHost( cptr_D )
```

Consideration over PINNED memory

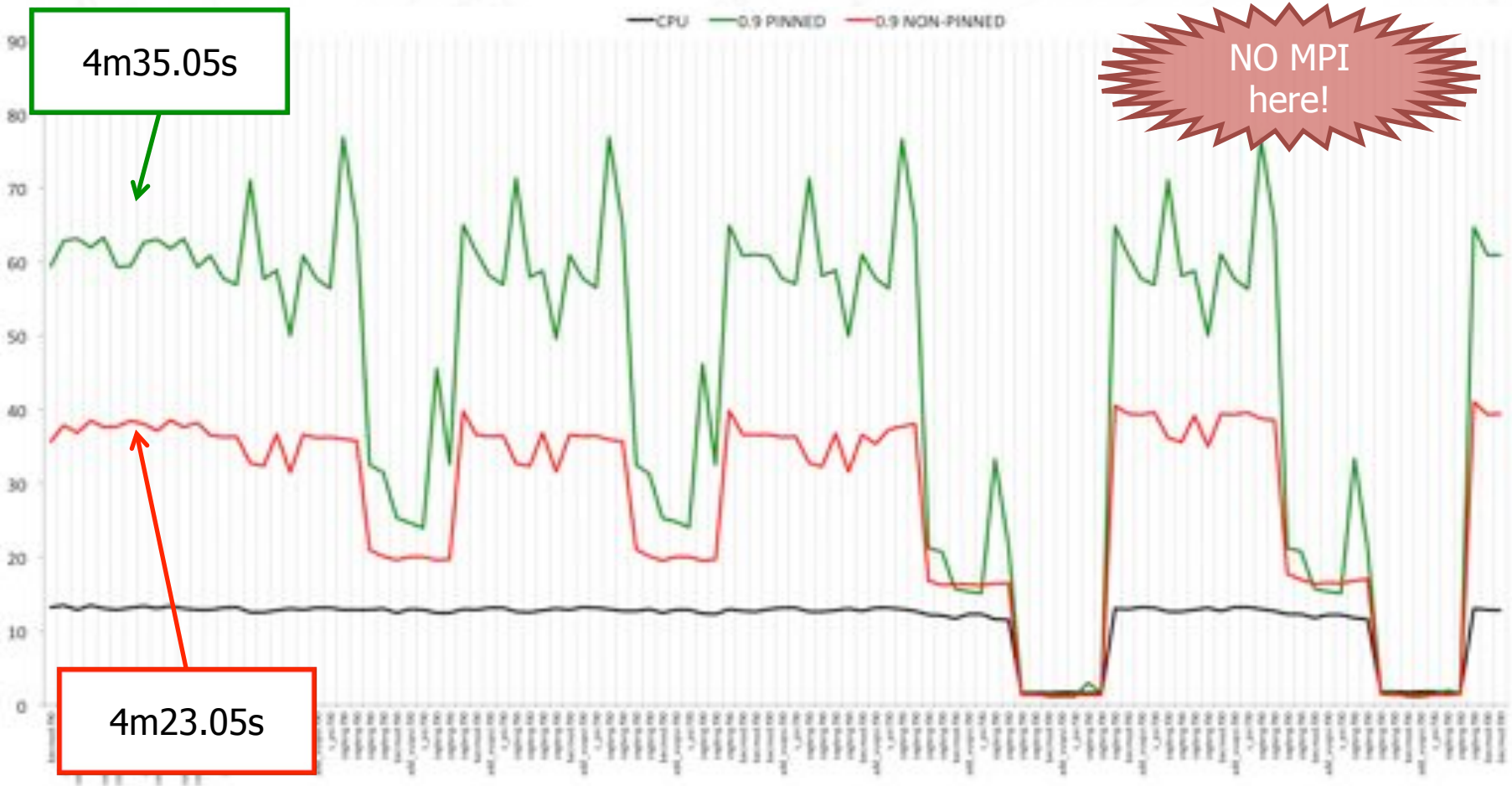
PINNED (or non-pageable) memory is a memory allocated on the host that…

- allows asynchronous data transfers (no CPU needed, DMA works alone);
- allows overlapping COMPUTATION/COMPUTATION on the same stream;
- since it is page-locked, the SO cannot move it;
- if you do not de-allocate it, never disappears*;
- if you allocate it too much (how much?), the overall performance of the system is affected.

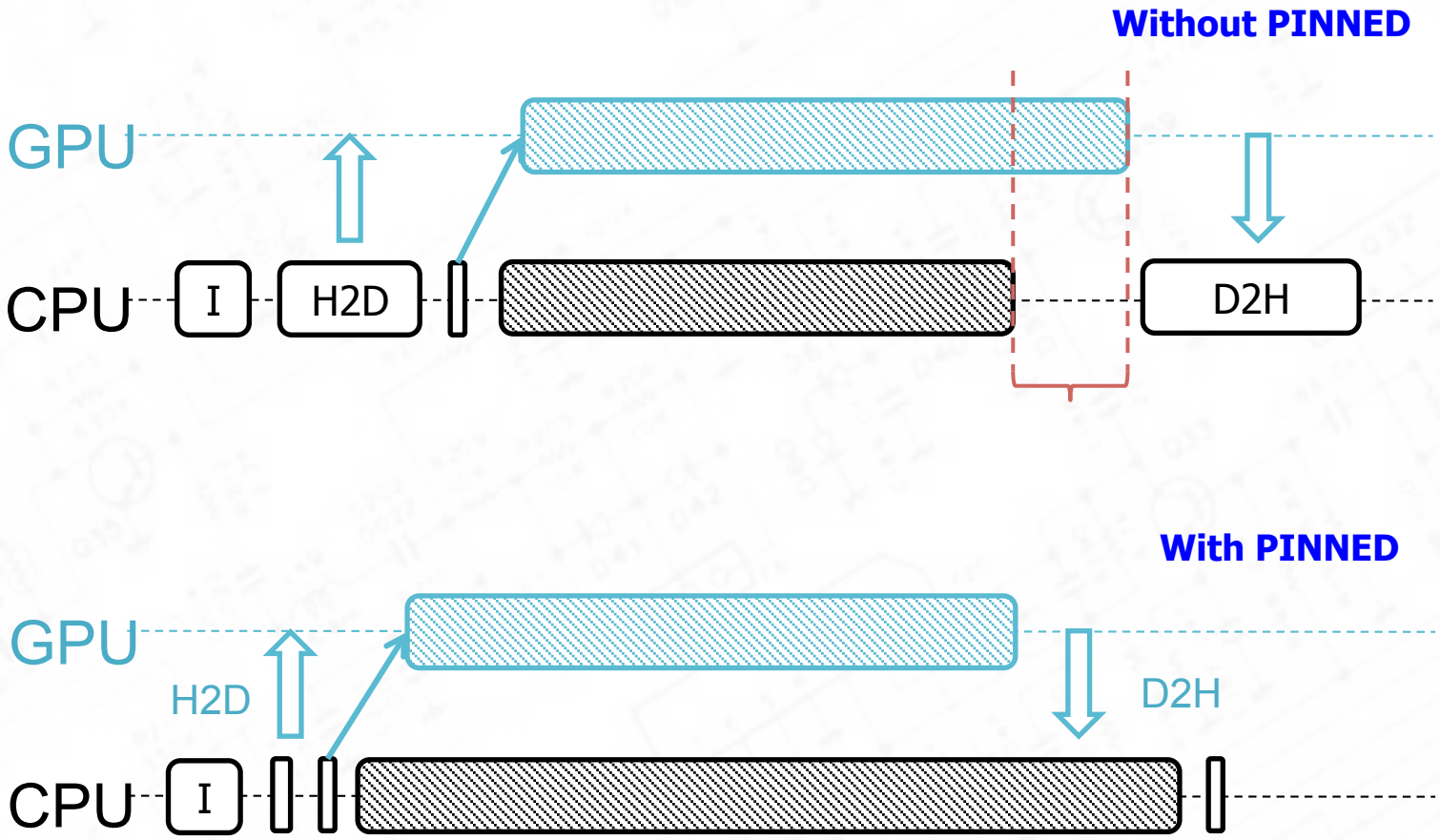
Note: you can probe if a host memory is pinned or not (`cudaHostGetFlags`)…

Note: if you code involves MPI communication is better to exploit GPU Direct features and avoid multiple pinned allocations at host level required to exchange "data on host memory" between the GPU and the IB driver…

Consideration over PINNED memory

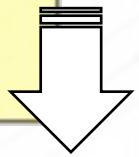


use-case: PHIGEMM



phiGEMM – Code Snippet (C)

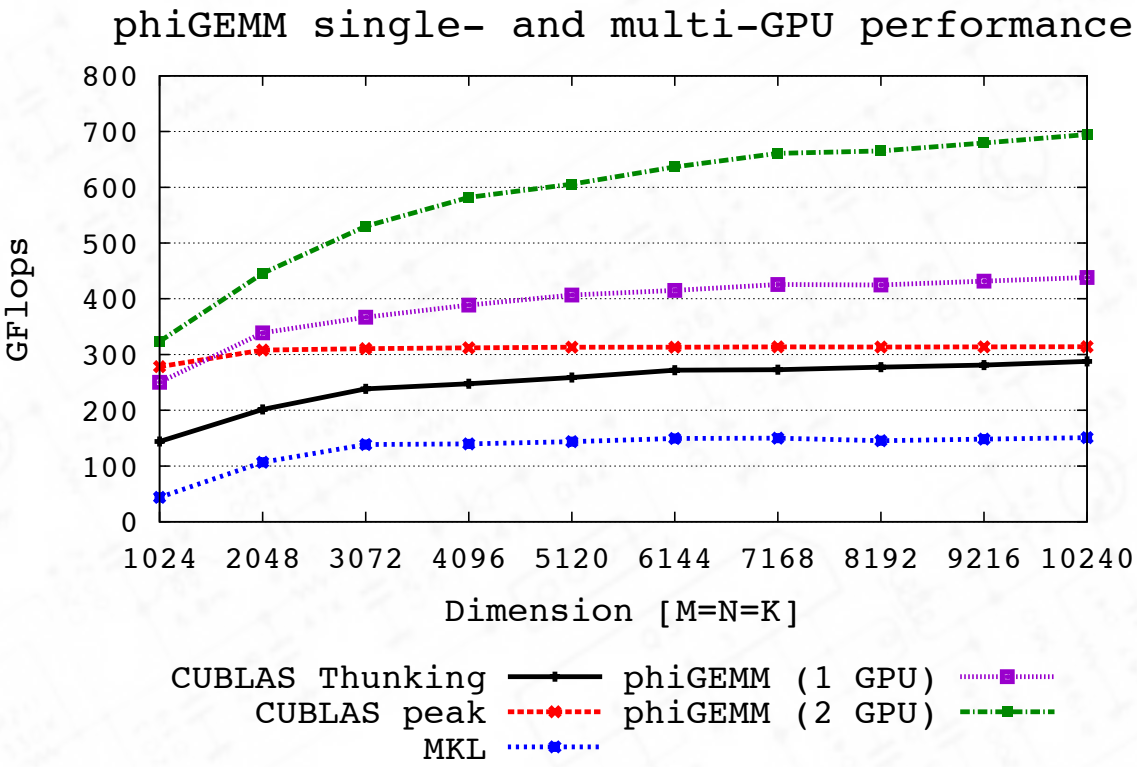
```
for (iDev = 0; iDev < phiGemmNumDevices * NSTREAMS; iDev++) {

     cudaSetDevice(deviceIds[iDev % phiGemmNumDevices]);

     for (j = 0; j < 6; j++)
          devPtrA[iDev]=(double *)(dev_scratch[iDev]);

     if ( is_transa ) {
          status = cublasSetMatrixAsync (k_h2d[iDev], m_h2d[iDev],
                         sizeof(double), A+shiftA, *lda, devPtrA[iDev],
                         k_gpu[iDev], phiStreams[iDev]);
          shiftA += m_h2d[iDev] * (*lda);
     } else {
          status = cublasSetMatrixAsync (m_h2d[iDev], k_h2d[iDev],
                         sizeof(double), A+shiftA, *lda, devPtrA[iDev],
                         m_gpu[iDev], phiStreams[iDev]);
          shiftA += m_h2d[iDev];
     }
…
```



PHIGEMM – Code Snippet (C)

```
      cublasGemm (phiHandles[ iDev ], cu_transa, cu_transb,
            m_gpu[iDev], n_gpu[iDev], k_gpu[iDev],
            alpha, devPtrA[iDev], gpu_lda, devPtrB[iDev], gpu_ldb,
            beta, devPtrC[iDev], m_gpu[iDev]);

      status = cublasGetMatrixAsync (m_h2d[iDev], n_h2d[iDev],
            sizeof(double), devPtrC[iDev], m_gpu[iDev], C+shiftC,
            *ldc, phiStreams[iDev]);

      if (status != CUBLAS_STATUS_SUCCESS) {
            fprintf (…);
      }
      …
}

gemm_mkl(transa, transb, &m_cpu, &n_cpu, &k_cpu, alpha, A+a_offset,
            lda, B+b_offset, ldb, beta, C+c_offset, ldc);
```

use-case: PHIGEMM



cuBLAS API: new *versus* Legacy

- the handle to the CUBLAS library context is initialized using the `cublasCreate(...)` function and it is explicitly passed to every subsequent library function call.
  - more control over the library setup when using multiple host threads and multiple GPUs.

- the scalars α and β can be passed by reference on the host or the device, instead of only being allowed to be passed by value on the host.
  - library functions can be executed asynchronously using streams even when α and β are generated by a previous kernel;
  - same for the scalar value output.

What change? (`cublas.h` → `cublas_v2.h`)

\+ new parameter in the call (handler)

Performance Measurement and Metrics

- Objective: find out the limiting factor in kernel performance
  - Memory bandwidth bound
  - Instruction throughput bound (compute)
  - Latency bound
  - Combination of the above
- Address the limiters in order of importance
  - Determine how close to the HW limits the resource is being used
  - Apply optimizations (*not in this presentation*)
- Inspection/Investigation
  - Automatically by the Visual Profile
  - Manually (*just few hints*)
- Typically an iterative process

Performance Measurement and Metrics

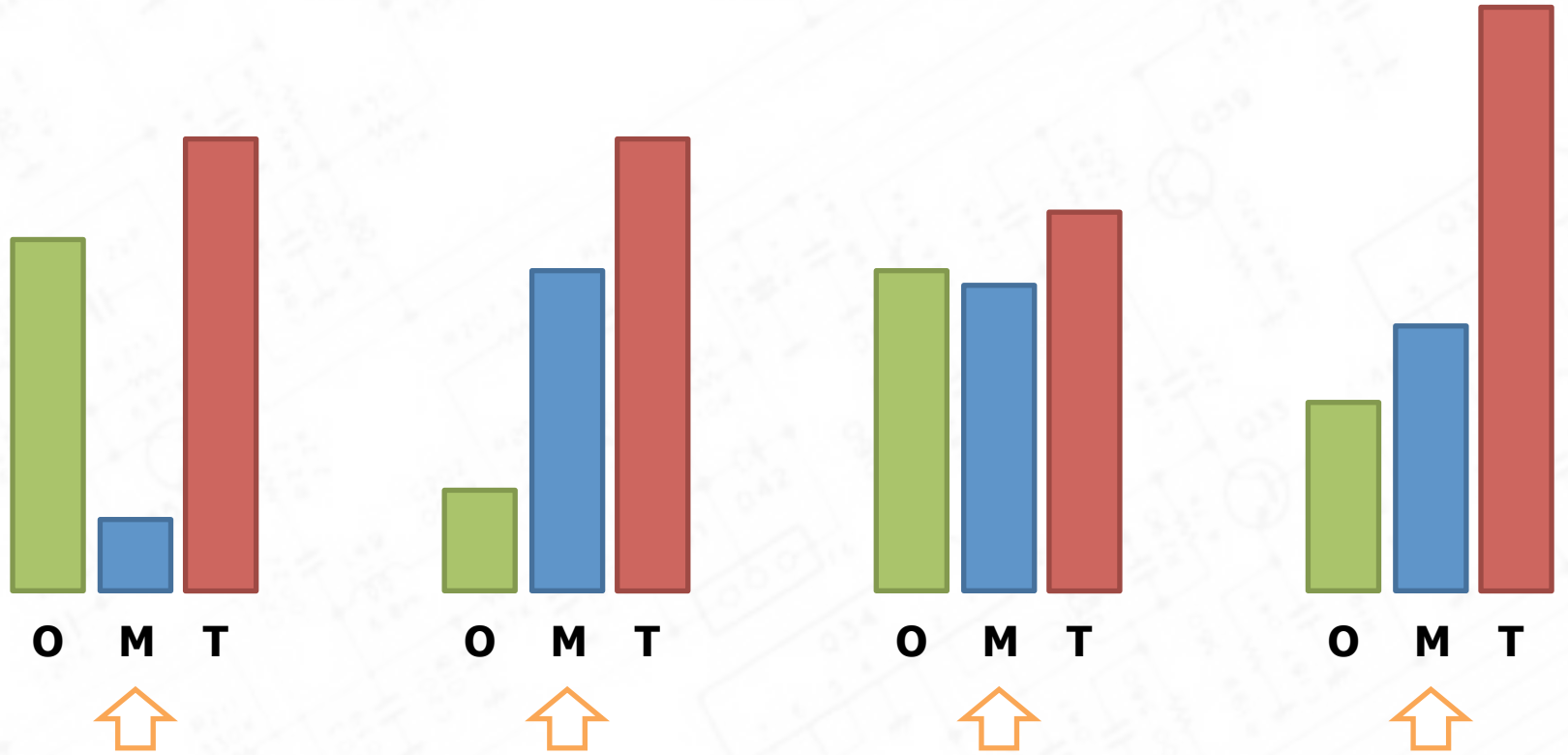
Visual Profiler…

- reports data by SM (with exception of L2 and DRAM counters)
- needs more executions to collect everything (so… it is a bit slow)
  - REMEMBER: blocks and warps are scheduled at run-time!
- to compute derivated statistics automatically you need it (and X)

Manually inspection…

- Re-code your kernel to obtain 3 versions…
  - a *base kernel* which performs the desired overall operation (T);
  - a *memory kernel* which has the same device memory access patterns as the base kernel but no math operations (M);
  - a *math kernel* which performs the math operations of the base kernel without accessing global memory (O).
- Run and evaluate the *gputime* (be smart → CUDA event)

Performance Measurement and Metrics



Compute-bound

Memory-bound

Balanced
(limited latency)

Latency issues

Performance Measurement and Metrics

```
__global__  base(float *a, float *b) {
      int i;
      i = (blockIdx.x-1)*blockDim.x + threadIdx.x;
      a[i] = sin(b[i]);
}


__global__ memory(float *a, float *b) {
      int i;
      i = (blockIdx.-1)*blockDim.x + threadIdx.x;
      a[i] = b[i];
}


__global__ math(float *a, float b, int flag) {
      float v;
      int i;
      i = (blockIdx.-1)*blockDim.x + threadIdx.x
      v = sin(b);
      if (v*flag == 1) a[i] = v;
}
```

Smart trick to avoid compiler interference at assembly level

Courtesy of M. Fatica

Performance Measurement and Metrics

Output of the command-line profiler (export CUDA_PROFILE=1)

```
method=[ base ] gputime=[ 871.680 ] cputime=[ 5.000 ] occupancy=[ 1.000 ]
method=[ memory ] gputime=[ 620.064 ] cputime=[ 5.000 ] occupancy=[ 1.000 ]
method=[ math ] gputime=[ 661.792 ] cputime=[ 5.000 ] occupancy=[ 1.000 ]
```

→ Kernel balanced, latency under control

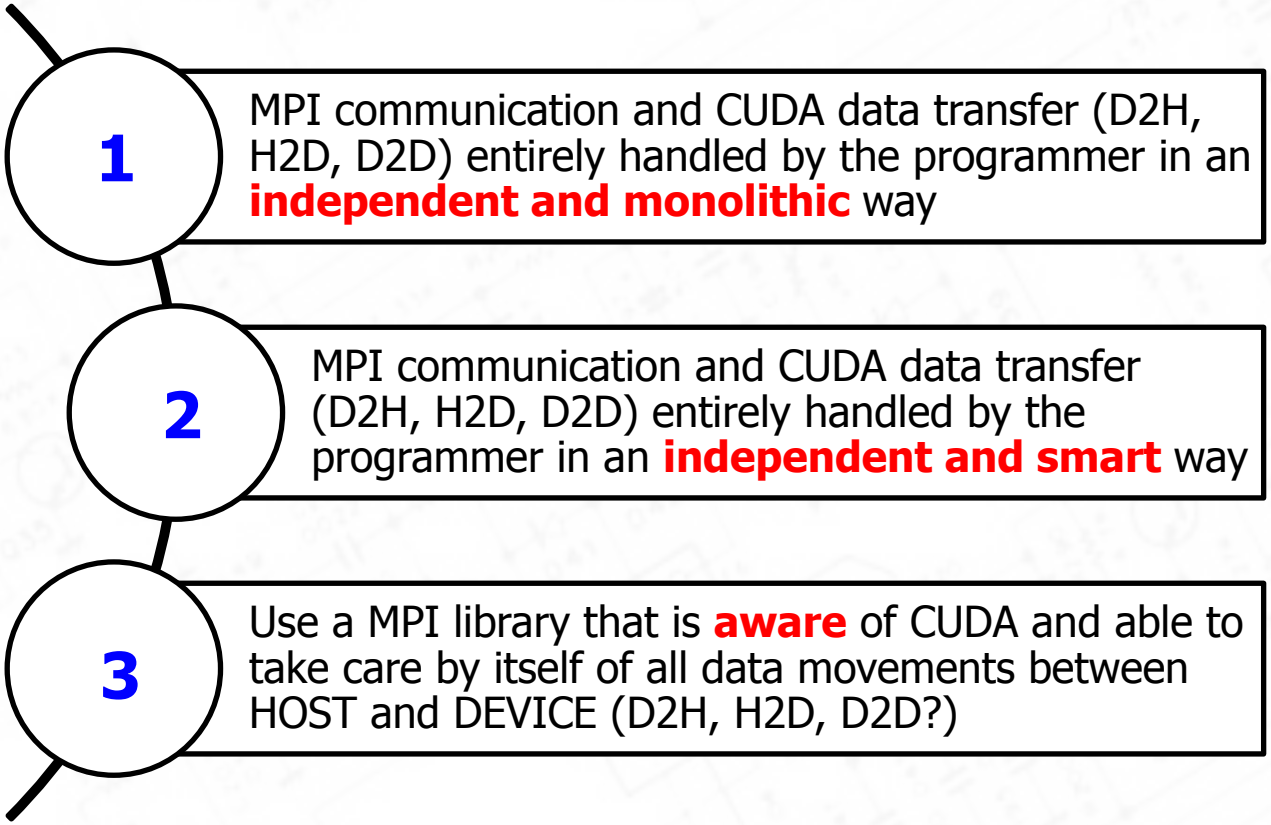
Can we do better? Yes! sin() → __sin()

```
method=[ base ] gputime=[ 626.944 ] cputime=[ 5.000 ] occupancy=[ 1.000 ]
method=[ memory ] gputime=[ 619.328 ] cputime=[ 4.000 ] occupancy=[ 1.000 ]
method=[ math ] gputime=[ 163.136 ] cputime=[ 5.000 ] occupancy=[ 1.000 ]
```

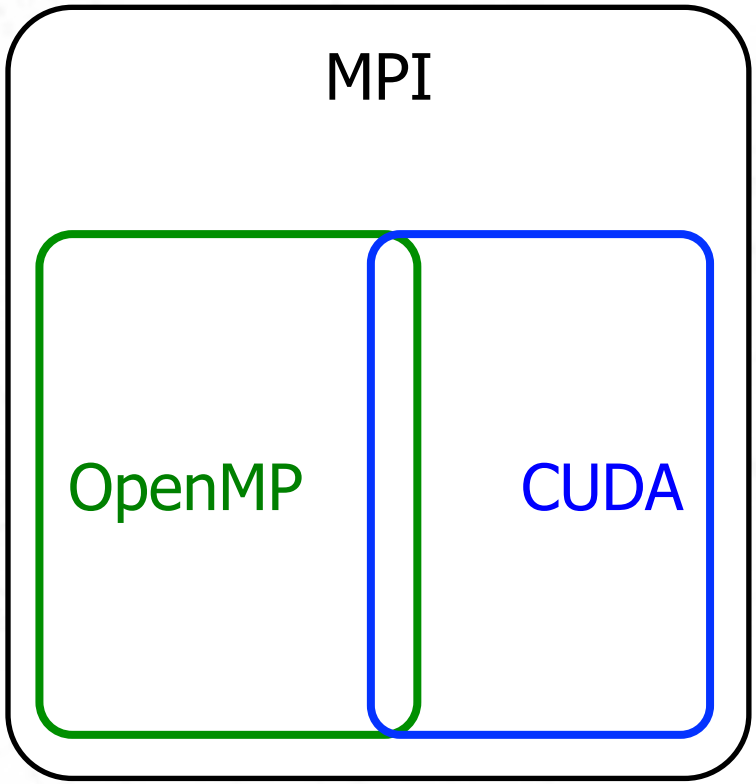
→ Kernel memory-bound

Courtesy of M. Fatica

MPI + CUDA, short summary



MPI + OpenMP + CUDA = what?



- Many HPC applications are adopting MPI+OpenMP nowadays
- We still need acceleration

Get shorter the time-to-solution

Take the positive of all models

- MPI: domain decomposition
- OpenMP: external big loops
- CUDA: computational-intensive inner loops and string FP throughput

Know what you have…

Let's forget about CUDA 3.x and "old" GPU devices…

Facts:

- OpenMP and CUDA target local computations
- MPI moves data (where is the data, HOST or DEVICE, is not *conceptually* so important) … and nothing else.
- All modern compilers are OpenMP compatible (al least for OpenMP v2.5)
- With CUDA 4.x it is now easier to manage multiple devices and to asynchronously off-load computation
- Current hybrid HW "forces" people to think about complex computations like a set of tasks with dependency that can be dispatch to the best computational device/unit available*

… and maximize it!

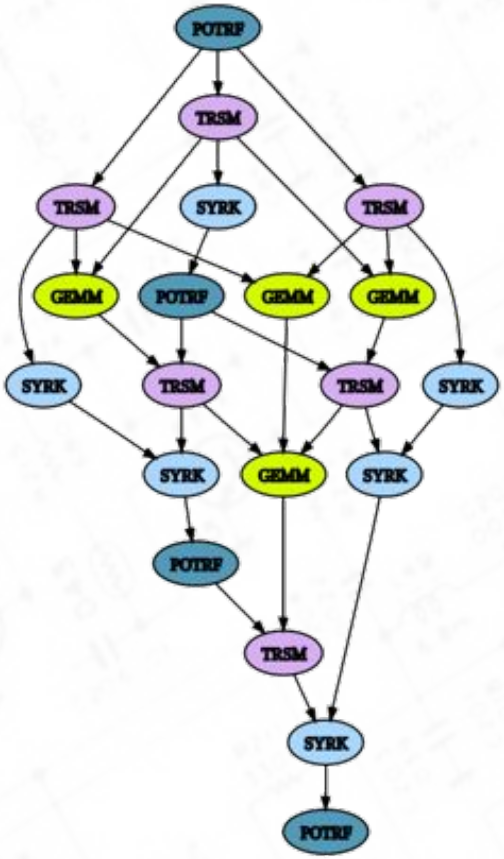
How? Keep in mind that…

- We live in a **hybrid multi-core** world
  - Cores per socket >= 4
  - Socket per node >= 2
  - GPU (or other accelerator) per node >=2

Challenging? **Yes!**
But not impossible.

- Current MPI+OpenMP codes (95%?) usually implement *master-only* or *funneled* hybrid strategies
  - When a MPI process communicates to another MPI process, other compute units are idle → overlapping?
- GPU computation is not-driven by the CPU
  - When GPU computes, CPU cores are free (and vice-versa) → overlapping?
- Algorithms implement task that might be independent each other
  - Let's the proper compute unit (CPU or GPU) do what is better for it

Example: PLASMA, DPLASMA & MAGMA



MAGMA uses **HYBRIDIZATION** methodology based on

- Representing linear algebra algorithms as collections of TASKS and DATA DEPENDENCIES among them
- Properly SCHEDULING the tasks' execution over the multicore and the GPU hardware components

What does HYBRIDIZATION means?

- Panels (Level 2 BLAS) are factored on CPU using LAPACK
- Trailing matrix updates (Level 3 BLAS) are done on the GPU using "look-ahead"

QUANTUM ESPRESSO in a nutshell

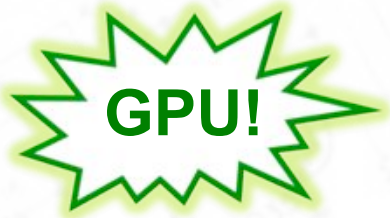
- QUANTUM ESPRESSO is an integrated suite of computer codes for electronic-structure calculations and materials modeling at the nanoscale.
  - Density-Functional Theory, plane waves, pseudo-potential, …
- It can run in parallel on several HPC architectures
  - Linux clusters, CRAY supercomputers, IBM Power & BlueGene, NEC, …
- It supports MPI and OpenMP for a high scalable parallel implementation
- It systematically uses standardized mathematical libraries
  - BLAS, LAPACK, FFTW, …
- Two main packages: PWSCF and CP

Time-consuming steps in PWSCF

- Calculation of Charge Density
  - FFT + matrix-matrix multiplication
- Calculation of Potential
  - FFT + operations on real-space grid
- Davidson Iterative Diagonalization (SCF)
  - FFT + eigenvalues/eigenvectors problem + matrix-matrix multiplication

Basically most CPU time spent in linear-algebra operations, implemented in BLAS and LAPACK libraries, and in FFT!

Parallelization levels in PWscf



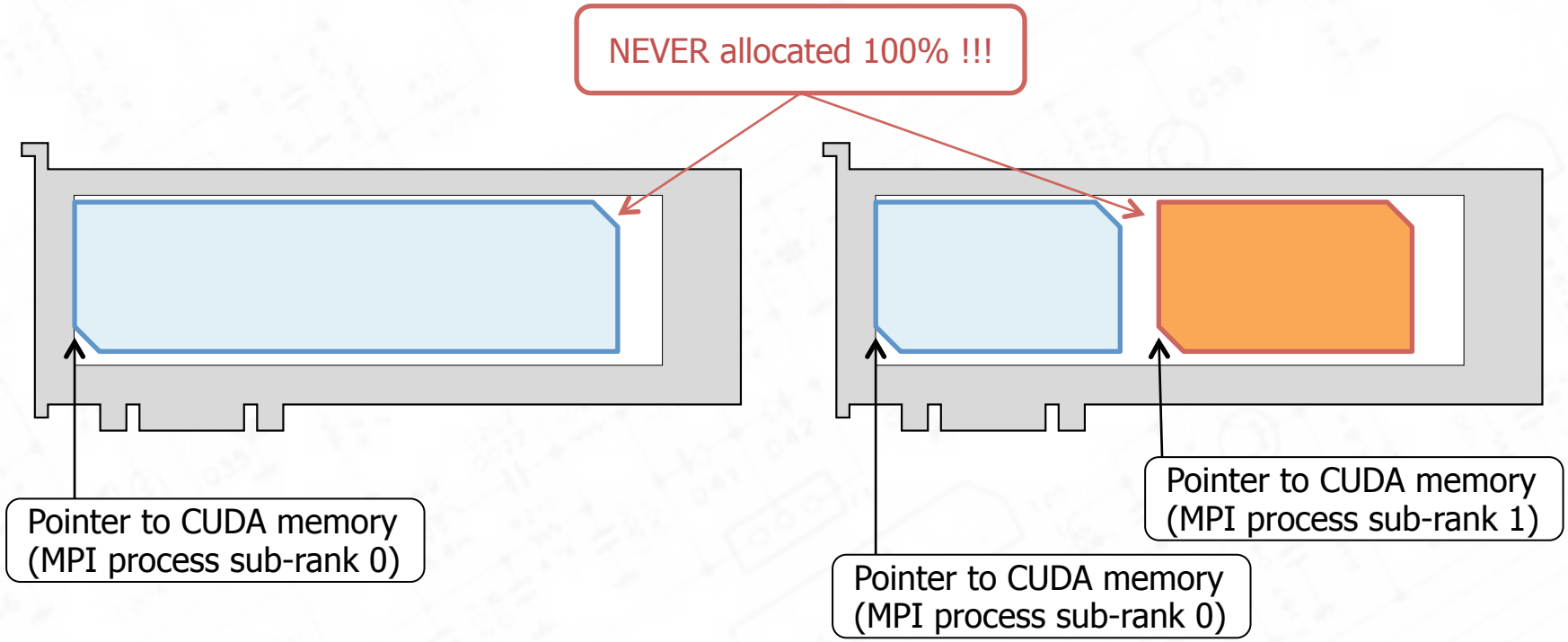
Images	• Only for Nudged Elastic Band (NEB) calculations
K-points	• Distribution over k-points (if more than one) • Scalability over k-points (if more than one) • No memory scaling
Plane-waves	• Distribution of wave-function coefficients • Distribution of real-grid points • Good memory scale, good overall scalability, LB
Linear algebra & task groups	• Iterative diagonalization (fully-parallel or serial) • Smart grouping of 3DFFTs to reduce *compulsory* MPI communications
Multi-threaded kernels	• OpenMP handled *explicitly* or *implicitly* • Extend the scaling on multi-core machines with "limited" memory

GPU Developments

- MPI-GPU binding & GPU memory management
- NEWD → CUDA NEWD (multiple kernels combined)
- ADDUSDENS → CUDA ADDUSDENS
- VLOC_PSI → CUDA VLOC_PSI (CUDA kernels + CUFFT)
- BLAS 3 *GEMM → PHIGEMM library
- (serial) LAPACK → MAGMA library

VLOC_PSI acts over distributed data
NEWD/ADDUSDENS act over local data

MPI-GPU binding & GPU memory management

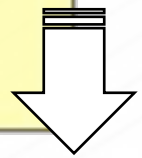


NOTE 1: PWscf keeps track of how much GPU memory is available

NOTE 2: any process can access the memory allocated to another process in the card!!!

MPI-GPU binding – Code Snippet (C)

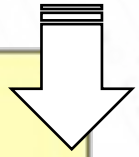
```
/* What is my name? */
MPI_Get_processor_name(lNodeName, &lNodeNameLength);

/* Collecting all names... */
lNodeNameRbuf = (char*) malloc(lSize * MPI_MAX_PROCESSOR_NAME * sizeof(char));
MPI_Allgather(lNodeName, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, lNodeNameRbuf,
                    MPI_MAX_PROCESSOR_NAME, MPI_CHAR, MPI_COMM_WORLD);

/* lRanksThisNode is a list of the global ranks running on this node */
lRanksThisNode = (int*) malloc(lSize * sizeof(int));

for(i = 0; i < lSize; i++) {
     if(strncmp(lNodeName, (lNodeNameRbuf + i * MPI_MAX_PROCESSOR_NAME),
               MPI_MAX_PROCESSOR_NAME) == 0) {
          lRanksThisNode[lNumRanksThisNode] = i;
          lNumRanksThisNode++;
     }
}
```



MPI-GPU binding – Code Snippet (C)

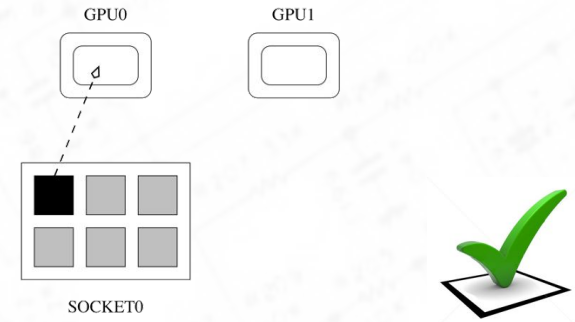
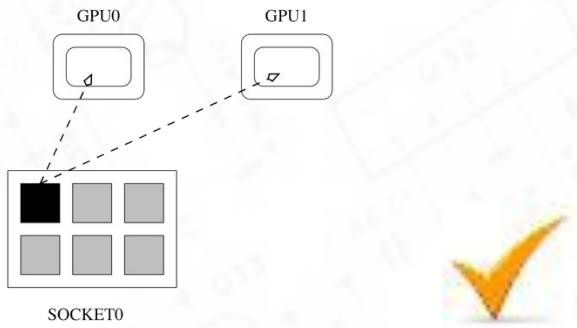
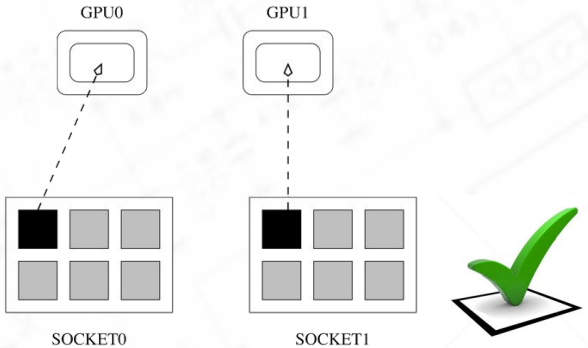
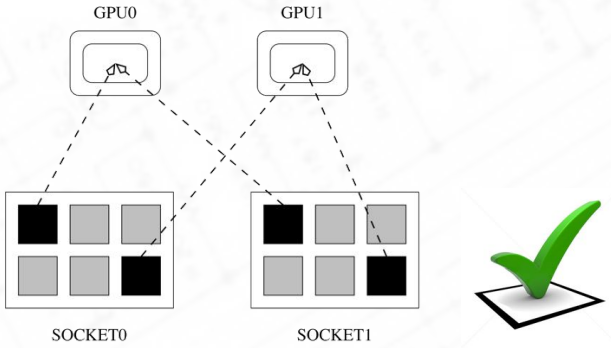
```
/* Create a communicator consisting of the ranks running on this node. */
MPI_Comm_group(MPI_COMM_WORLD, &lWorldGroup);
MPI_Group_incl(lWorldGroup, lNumRanksThisNode, lRanksThisNode, &lThisNodeGroup);

MPI_Comm_create(MPI_COMM_WORLD, lThisNodeGroup, &lThisNodeComm);

MPI_Comm_rank(lThisNodeComm, &lRankThisNode);
MPI_Comm_size(lThisNodeComm, &lSizeThisNode);

/* Attach all MPI processes on this node to the available
 * GPUs in round-robin fashion
 */
cudaGetDeviceCount(&lNumDevicesThisNode);

/* Multiple MPI can share a GPU...  */
for (i = 0; i < ngpus_per_process; i++) {
     qe_gpu_bonded[i] = lRankThisNode % lNumDevicesThisNode;
}
```

GPU memory management – Code Snippet (C)



```
/* Loop over GPU per process (serial >= 1, parallel always 1) */
for (i = 0; i < ngpus_per_process; i++) {

     /* Set active GPU device */
     if ( cudaSetDevice(qe_gpu_bonded[i]) != cudaSuccess) MPI_Abort(...);

     ierr = cudaMalloc ( (void**) &(dev_scratch_QE[i]), 0);
     if ( ierr != cudaSuccess) MPI_Abort(...);

     /* Query the real free memory, taking into account the "stack" */
     cuMemGetInfo(&free, &total);

     cuda_memory_allocated[i] = (size_t)
                         (free * __SCALING_MEM_FACTOR__ / procs_per_gpu);

     ierr = cudaMalloc ( (void**) &(dev_scratch_QE[i]), (size_t)
                                        cuda_memory_allocated[i] );
     if ( ierr != cudaSuccess) MPI_Abort(...);
}
```

MPI/Process-GPU scenarios

	Single GPU per MPI/Process	Multiple GPU per MPI/Process
SERIAL		
PARALLEL		

ADDUSDENS – Code Snippet (FORTRAN)

```
DO ih = 1, nh (nt)
     DO jh = ih, nh (nt)
          CALL qvan2 (ngm, ih, jh, nt, qmod, qgm, ylmk0)
          ijh = ijh + 1
          DO na = 1, nat
               IF ( ityp(na) .eq. nt) THEN
                    DO is = 1, nspin_mag
                         DO ig = 1, ngm
                              skk = eigts1 (mill (1,ig), na) * &
                                    eigts2 (mill (2,ig), na) * &
                                    eigts3 (mill (3,ig), na)
                              aux(ig,is) = aux(ig,is) + qgm(ig)*skk*becsum(ijh,na,is)
                         ENDDO
                    ENDDO
               ENDIF
          ENDDO
     ENDDO
ENDDO
```

CUDA ADDUSDENS – Code Snippet (C)

```
for( ih = 0, iih = 1; ih < nh[nt - 1]; ih++, iih++) {
  for( jh = ih, jjh = iih; jh < nh[nt - 1]; jh++, jjh++, ijh++ ) {

     qvan2_(ptr_ngm, &iih, &jjh, ptr_nt, qmod, qgm, ylmk0);
     /* Protective guard */
     cudaDeviceSynchronize();

     qecudaSafeCall(
       cudaMemcpy( qgm_D, qgm,  sizeof( double ) * ngm * 2, cudaMemcpyHostToDevice ));

     kernel_compute_aux<<<grid2, threads2>>>( eigts1_D, eigts2_D, eigts3_D, ig1_D,
          ig2_D, ig3_D, nr1, nr2, nr3, qgm_D, becsum_D, aux_D, na, nspin_mag, ngm,
          first_becsum, ijh, nat, ityp_D, nt );
     qecudaGetLastError("kernel launch failure");
  }
}

qecudaSafeCall( cudaMemcpy( aux, aux_D, sizeof( double ) * ( ngm * nspin_mag * 2 ),
cudaMemcpyDeviceToHost ) );
```

QVAN2 – Code Snippet (FORTRAN)

```
DO lm = 1, lpx (ivl, jvl)
     lp = lpl (ivl, jvl, lm)
     IF ( lp < 1 .or. lp > 49 ) CALL errore (... )

     ! find angular momentum l corresponding to combined index lp
     IF (lp == 1) THEN
          ...
     ELSEIF ( lp <= 4) THEN
          ...
     ELSEIF ( lp <= 9 ) THEN
          ...
     ELSEIF ( lp <= 16 ) THEN
          ...
     ELSEIF ( lp <= 25 ) THEN
          ...
     ELSE
          ...
     ENDIF
...
```

IS IT WORTH TO PORT TO CUDA?

CUDA ADDUSDENS – Code Snippet (C)

```
qecudaSafeCall( cudaMemcpy( ig1_D, ig1,  sizeof( int ) * ngm,
                                          cudaMemcpyHostToDevice ) );
...
qecudaSafeCall( cudaMemcpy( eigts1_D, eigts1,
         sizeof( double ) * (((nr1*2 + 1)*nat)*2), cudaMemcpyHostToDevice ) );
...
qecudaSafeCall( cudaMemcpy( aux_D, aux,
         sizeof( double ) * (ngm*nspin_mag*2), cudaMemcpyHostToDevice ) );
...

dim3 threads2(1, __CUDA_TxB_ADDUSDENS_COMPUTE_AUX__);
dim3 grid2( nspin_mag / 1 ? nspin_mag / 1 : 1,
(ngm+__CUDA_TxB_ADDUSDENS_COMPUTE_AUX__-1)/__CUDA_TxB_ADDUSDENS_COMPUTE_AUX__?
(ngm+__CUDA_TxB_ADDUSDENS_COMPUTE_AUX__-1)/__CUDA_TxB_ADDUSDENS_COMPUTE_AUX__:1);

qecudaSafeCall( cudaFuncSetCacheConfig(kernel_compute_aux,
                                       cudaFuncCachePreferShared) );
```

CUDA ADDUSDENS, Considerations

- CUDA kernel is mainly compute-bounded

```
ptxas info    : Compiling entry function
'_Z18kernel_compute_auxPKdS0_S0_PKiS2_S2_iiiS0_S0_PdiiiiiiS2_i' for 'sm_20'
ptxas info    : Function properties for _Z18kernel_compute_auxPKdS0_S0_PKiS2_S2_iiiS0_S0_PdiiiiiiS2_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 31 registers, 4096+0 bytes smem, 156 bytes cmem[0]
```

  - 1024 threads per block → OCCUPANCY = 0.66/0.67
  - Memory throughput is really low! (6.73/144 GBs)
  - It uses shared memory (SHARED/L1 as 48/16 or 16/48?)
- QVAN2 (OpenMP) and KERNEL_COMPUTE_AUX (CUDA) can overlap
- Best performance measured: 20x (Realistic? Yes*)

Q: Can be further optimized ??

ADDUSDENS – Code Snippet (FORTRAN)

```
DO ih = 1, nh (nt)
     DO jh = ih, nh (nt)
          CALL qvan2 (ngm, ih, jh, nt, qmod, qgm, ylmk0)
          ijh = ijh + 1
          DO na = 1, nat
               IF ( ityp(na) .eq. nt) THEN
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(is, skk, ig)
                    DO ig = 1, ngm
                         skk = eigts1 (mill (1,ig), na) * &
                               eigts2 (mill (2,ig), na) * &
                               eigts3 (mill (3,ig), na)
                         DO is = 1, nspin_mag
                              aux(ig,is) = aux(ig,is) + qgm(ig)*skk*becsum(ijh,na,is)
                         ENDDO
                    ENDDO
!$OMP END PARALLEL DO
               ENDIF
          ENDDO
...
```

"new" CUDA ADDUSDENS – Code Snippet (C)

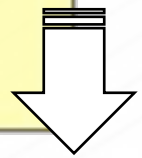
```
cudaStream_t stream[QE_NUM_SUPPORTED_STREAMS];
cudaStreamCreate(&stream[0]);

for( ih = 0, iih = 1; ih < nh[nt - 1]; ih++, iih++) {
  for( jh = ih, jjh = iih; jh < nh[nt - 1]; jh++, jjh++, ijh++ ) {

     qvan2_(ptr_ngm, &iih, &jjh, ptr_nt, qmod, qgm, ylmk0);
     cudaStreamSynchronize(stream[0]);

     cudaMemcpyAsync (qgm_D, qgm,
          sizeof( double ) * ngm * 2, cudaMemcpyHostToDevice, stream[0]);

     kernel_compute_aux<<<grid2, threads2, 0, stream[0]>>>( eigts1_D, eigts2_D,
          eigts3_D, ig1_D, ig2_D, ig3_D, nr1, nr2, nr3, qgm_D, becsum_D, aux_D, na,
          nspin_mag, ngm, first_becsum, ijh, nat, ityp_D, nt );
     // cudaStreamQuery()
  }
}
```

ADDUSDENS – Code Snippet (FORTRAN)

```
DO ih = 1, nh (nt)
     DO jh = ih, nh (nt)
          CALL qvan2 (ngm, ih, jh, nt, qmod, qgm, ylmk0)
          ijh = ijh + 1
          DO na = 1, nat
               IF ( ityp(na) .eq. nt) THEN
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(is, skk, ig)
                    DO ig = 1, ngm
                         skk = eigts1 (mill (1,ig), na) * &
                               eigts2 (mill (2,ig), na) * &
                               eigts3 (mill (3,ig), na)
                         DO is = 1, nspin_mag
                              aux(ig,is) = aux(ig,is) + qgm(ig)*skk*becsum(ijh,na,is)
                         ENDDO
                    ENDDO
!$OMP END PARALLEL DO
               ENDIF
          ENDDO
...
```

CUDA NEWD – Code Snippet (C)

```
for( ih = 0, iih = 1; ih < nh[nt - 1]; ih++, iih++ ) {
  for( jh = ih, jjh = iih; jh < nh[nt - 1]; jh++, jjh++ ) {

     qvan2_(ptr_ngm, &iih, &jjh, ptr_nt, qmod, qgm, ylmk0);

     qecudaSafeCall( cudaMemcpy( qgm_D, qgm,
          sizeof( double ) * ngm * 2, cudaMemcpyHostToDevice ) );

     for( na = 0;  na < nat; na++ ){
          if( ityp[na] == nt ) {

               blocksPerGrid = ( (nspin_mag * ngm * 2 ) + 256 - 1) / 256;
               kernel_compute_qgm_na<<<blocksPerGrid, threadsPerBlock>>>(
                         eigts1_D, eigts2_D, eigts3_D, ig1_D, ig2_D, ig3_D,
                         qgm_D, nr1, nr2, nr3, na, ngm, qgm_na_D );
               qecudaGetLastError("kernel launch failure");

               ...
```



CUDA NEWD – Code Snippet (C)

```
                cublasSetPointerMode(vlocHandles[0] , CUBLAS_POINTER_MODE_DEVICE);
                ...

                for( is = 0; is < nspin_mag; is++ ){

#if defined(__CUDA_3)
                        dtmp[is] = cublasDdot( ngm * 2,
                                        (double *) aux_D + (is * ngm * 2), 1,
                                        (double *) qgm_na_D, 1);
#else
                        cublasDdot(vlocHandles[0] , ngm * 2,
                                        (double *) aux_D + (is * ngm * 2), 1,
                                        (double *) qgm_na_D, 1, &dtmp_D[is] );
#endif

                }

                ...
```

Data on the HOST

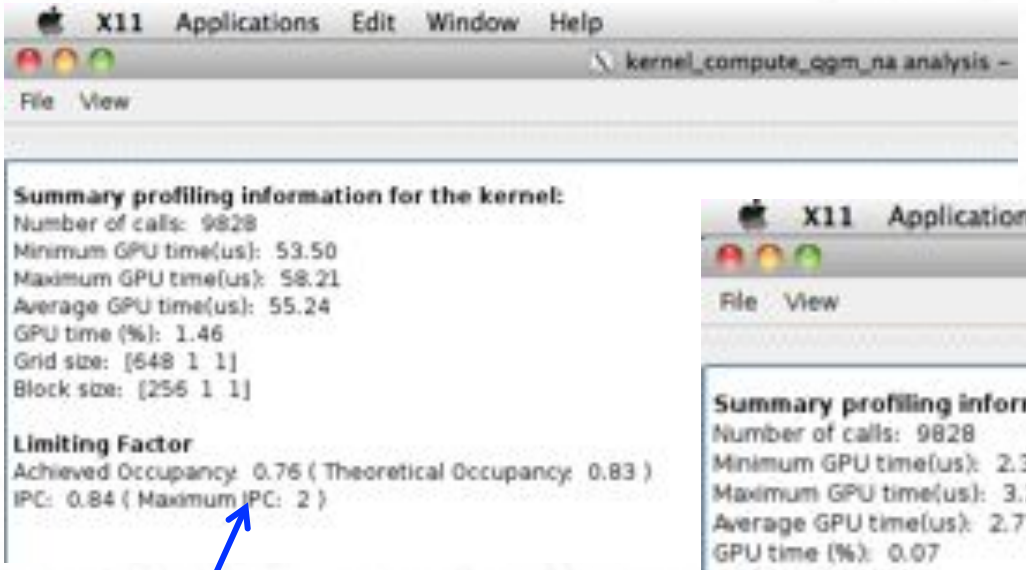
Data on the HOST or on the DEVICE?

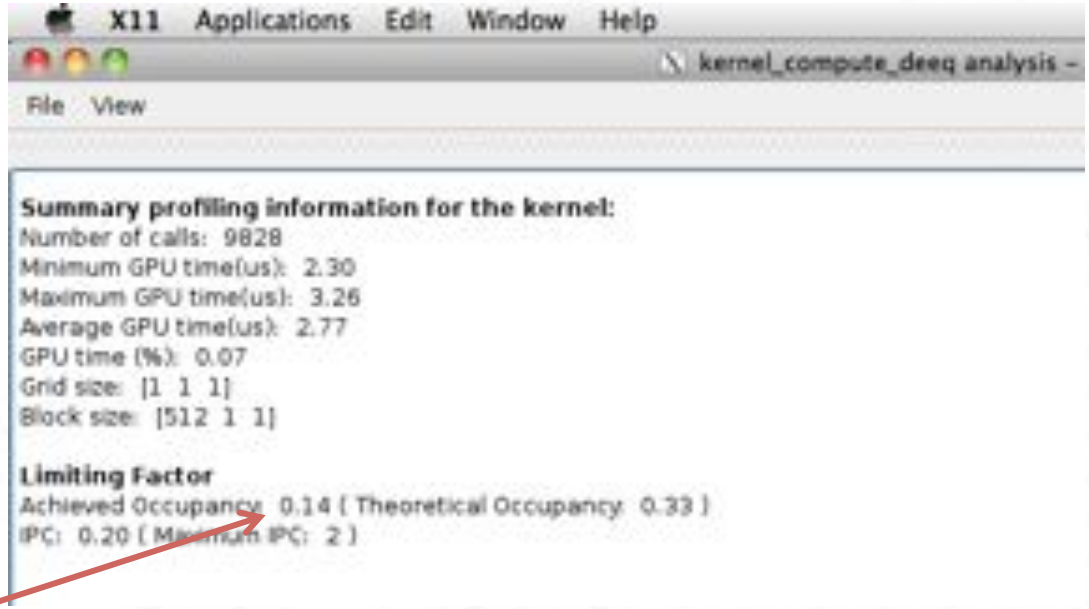
CUDA NEWD – Code Snippet (C)

```
                ...

#if defined __CUDA_3
                qecudaSafeCall( cudaMemcpy( dtmp_D, dtmp,
                          sizeof( double ) * nspin_mag, cudaMemcpyHostToDevice ) );
#endif

                blocksPerGrid = ( (nspin_mag) + 256 - 1) / 256;
                kernel_compute_deeq<<<blocksPerGrid, threadsPerBlock>>>( qgm_D, deeq_D,
                          aux_D, na, nspin_mag, ngm, nat, flag, ih, jh, nhm, omega,
                          fact, qgm_na_D, dtmp_D );
                qecudaGetLastError("kernel launch failure");
          }
     }
  }
}
```

Poor parallelism (nspin_mag), CUDA kernel dropped!

CUDA NEWD – Profiling Occupancy

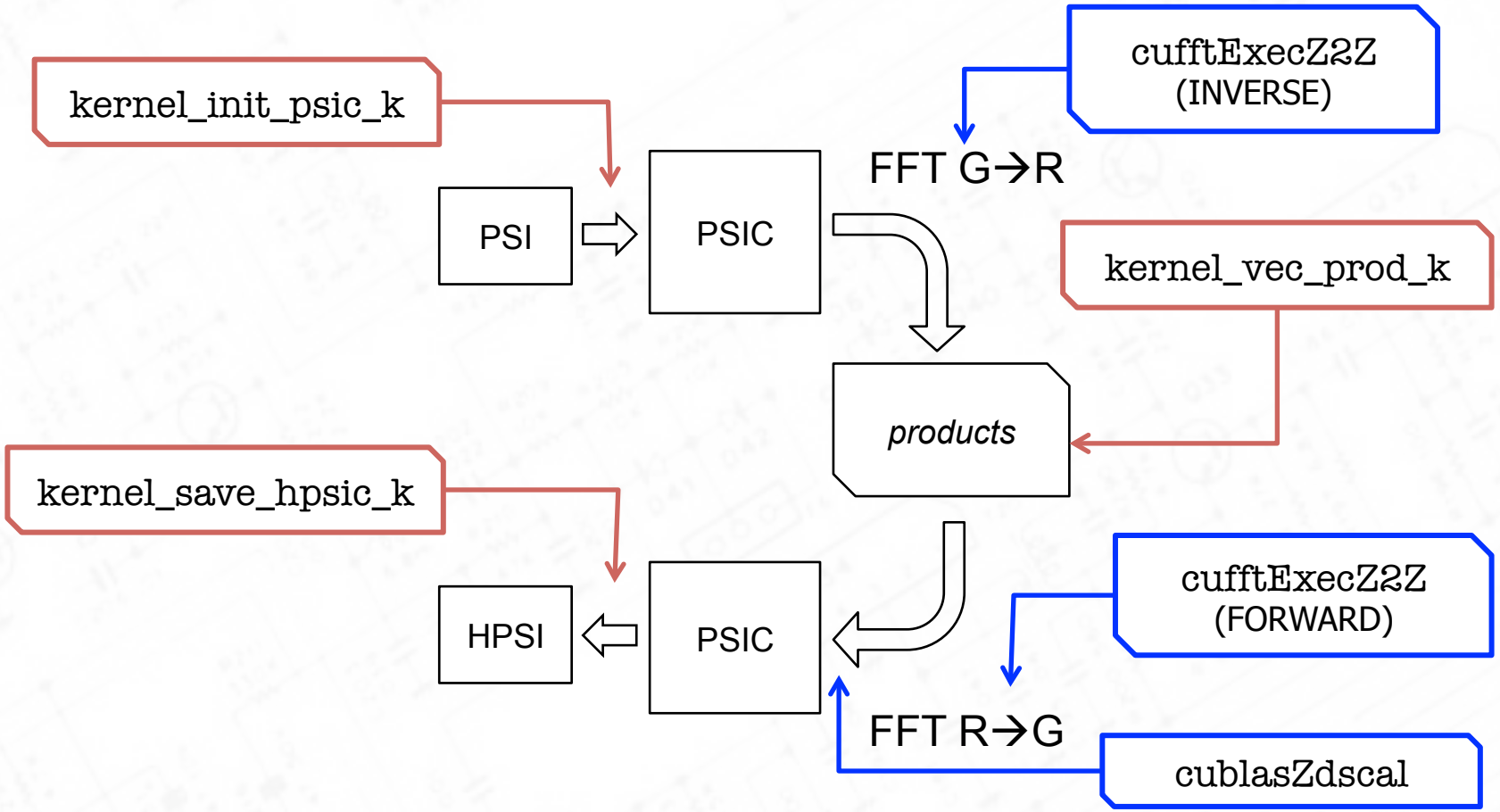




GOOD!

BAD!

CUDA VLOC_PSI_K serial – Computation flow



CUDA VLOC_PSI_K serial – CUFFT invocation

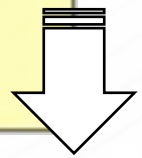
```
cufftHandle p_global;
qecheck_cufft_call( cufftPlan3d( &p_global, nr3s, nr2s, nr1s, CUFFT_Z2Z ) );
tscale = 1.0 / (double) ( size_psic );
if( cufftSetStream(p_global, vlocStreams[0]) != CUFFT_SUCCESS )
     exit( EXIT_FAILURE );

for( ibnd =  0; ibnd < m; ibnd = ibnd + 1 )
     ...
     qecheck_cufft_call( cufftExecZ2Z( p_global, psic_D, psic_D, CUFFT_INVERSE ) );
     ...
     qecheck_cufft_call( cufftExecZ2Z( p_global, psic_D, psic_D, CUFFT_FORWARD ) );
#if defined(__CUDA_3)
     cublasZdscal(size_psic, tscale, (cuDoubleComplex *) psic_D, 1);
#else
     cublasZdscal(vlocHandles[0], size_psic, &tscale, (cuDoubleComplex *)psic_D, 1);
#endif
     ...
}
qecheck_cufft_call( cufftDestroy(p_global) );
```

Dimension reversed!
$\{n_3, n_2, n_1\}$, not $\{n_1, n_2, n_3\}$

Data always on the device!
Result is an array, not a scalar

KERNEL_INIT_PSIC_K – Code Snippet (CUDA)

```
__global__ void kernel_init_psic_k( const int * nls, const int * igk,
                                    const double * psi, double *psic,
                                    const int N, const int ibnd )
{
     /* Compute access location base on kernel configuration lunch */
     int ix = blockIdx.x * blockDim.x * blockDim.y +
              threadIdx.y * blockDim.x + threadIdx.x;

     int psic_index_nls;
     int psi_index = ( ix + ( ibnd * N ) ) * 2;

     /* psi becomes psic. N is the length of igk */
     if ( ix < N ) {
          psic_index_nls = ( nls[ igk[ ix ] - 1 ] - 1 ) * 2;
          psic[ psic_index_nls ] = psi[ psi_index ];
          psic[ psic_index_nls + 1 ] = psi[ psi_index + 1 ];
     }
}
```



KERNEL_INIT_PSIC_K – Considerations

- CUDA kernel is mainly memory-bounded

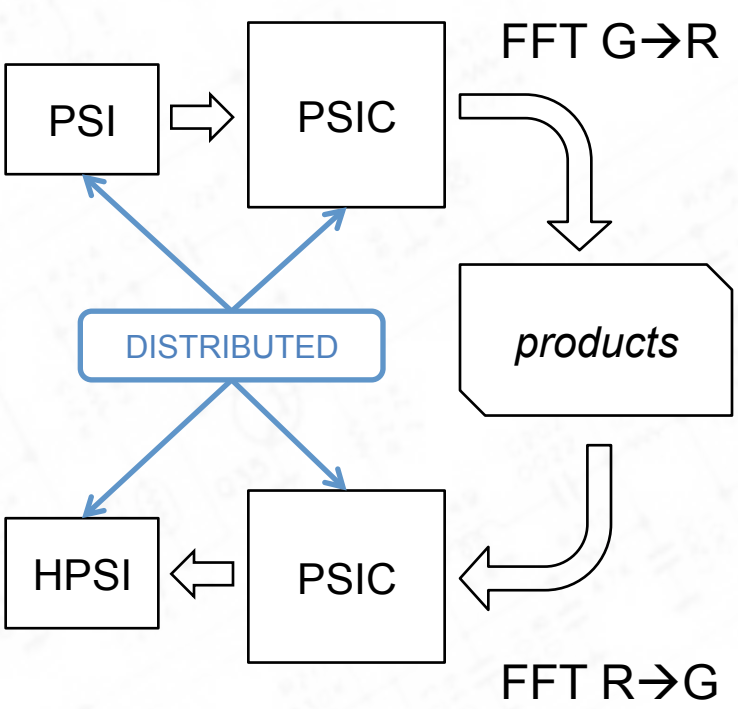
```
ptxas info    : Compiling entry function '_Z18kernel_init_psic_kPKiS0_PKdPdii' for 'sm_20'
ptxas info    : Function properties for _Z18kernel_init_psic_kPKiS0_PKdPdii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 10 registers, 72 bytes cmem[0]
```

  - Memory throughput is closer to the effective bandwidth (~90%)
  - Low number of registers → we can use low amount of threads (like 64 or 128) to schedule more blocks per SM → this is good if more processes use the same GPU
- Profiler shows that there are lots of non-coalescence accesses to global memory…

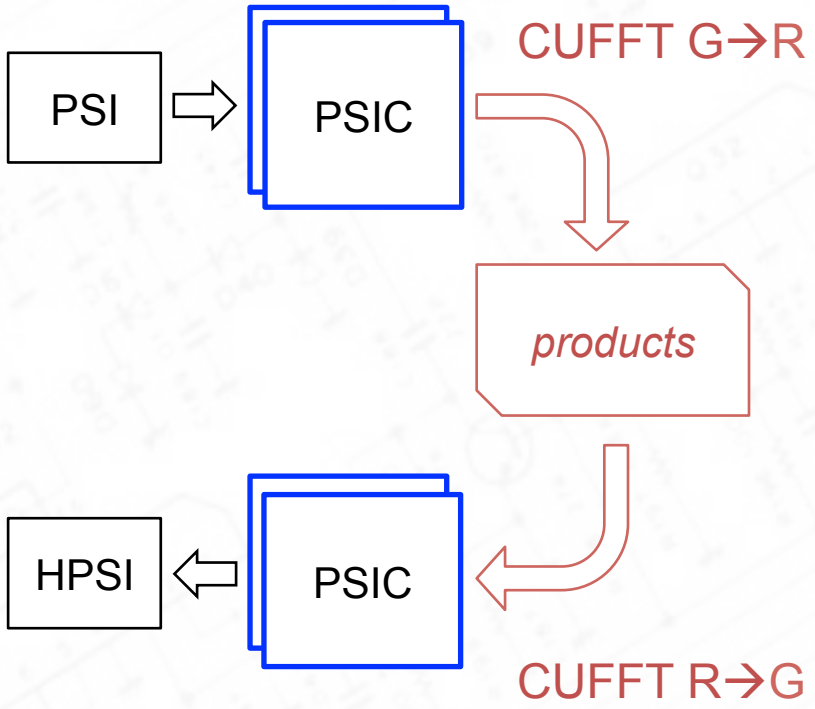
Q: Can be further optimized ??

A: Not really and not easily, we cannot change the data structure and break the code!

CUDA VLOC_PSI_K – parallel

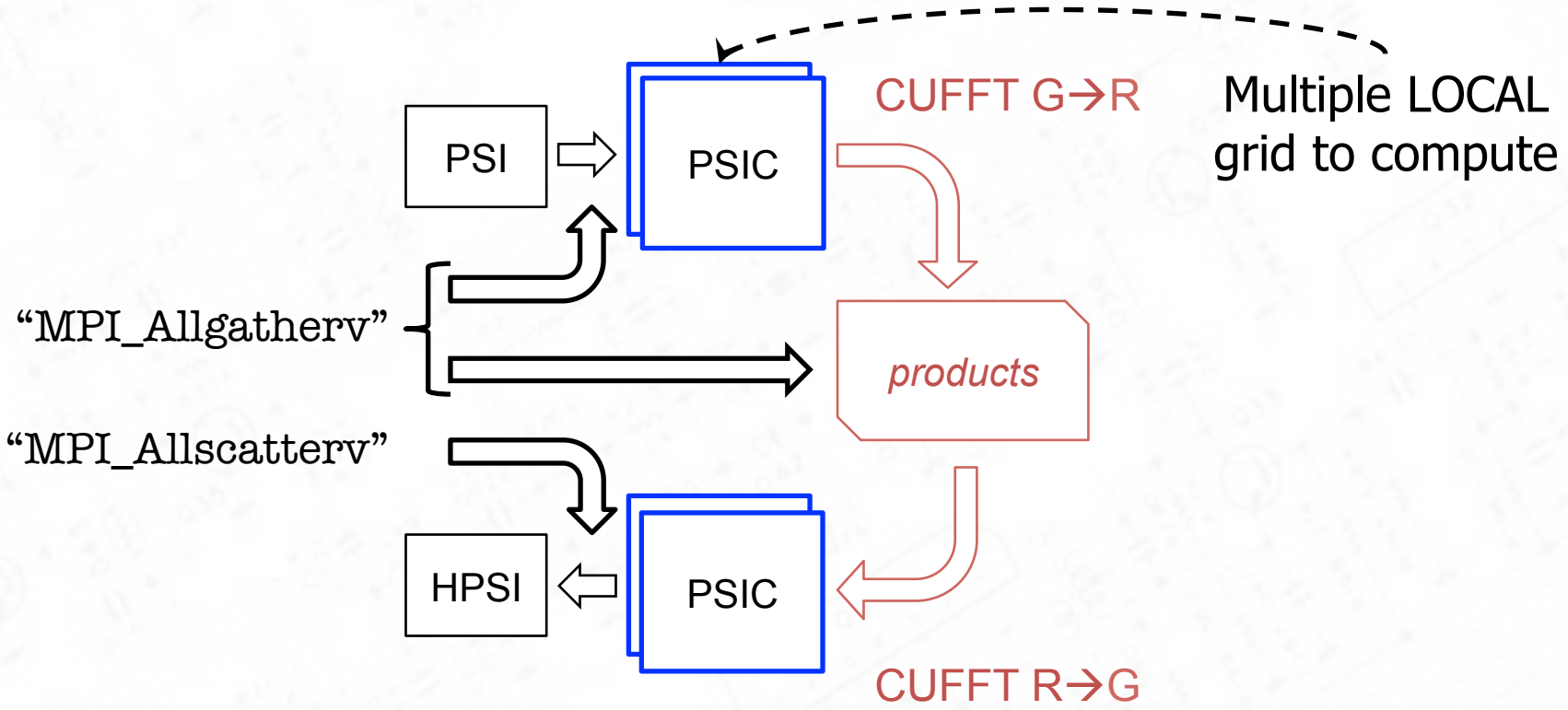


VLOC_PSI serial



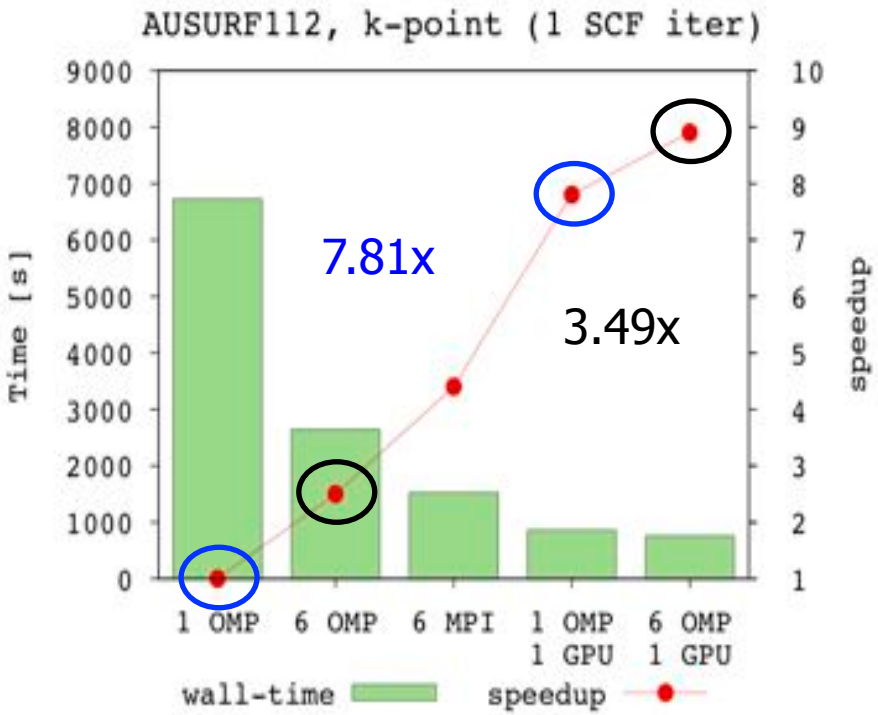
vloc_psi parallel

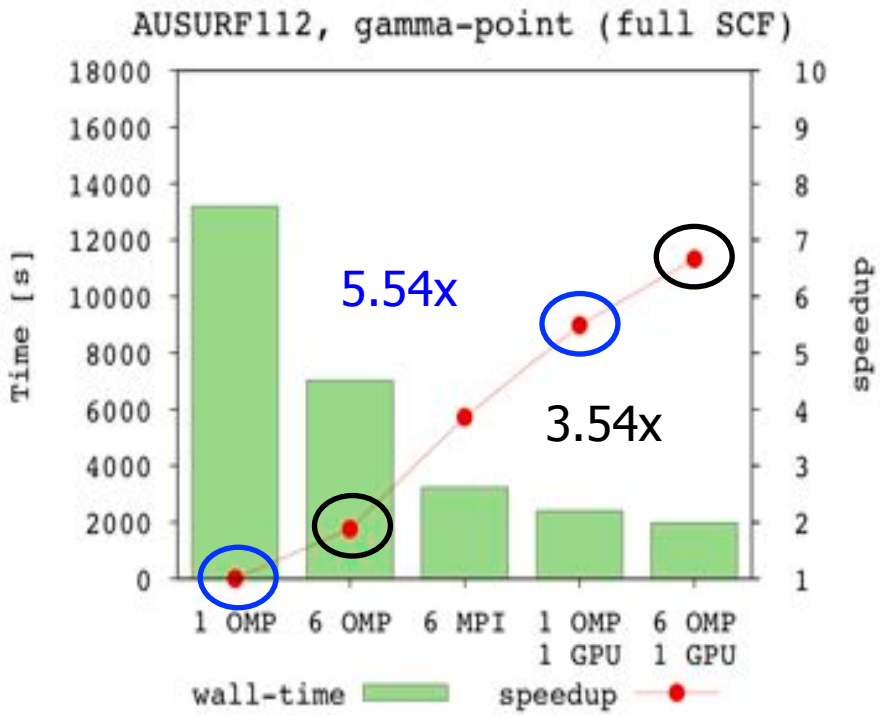
CUDA VLOC_PSI_K – parallel



Overlapping is possible!!

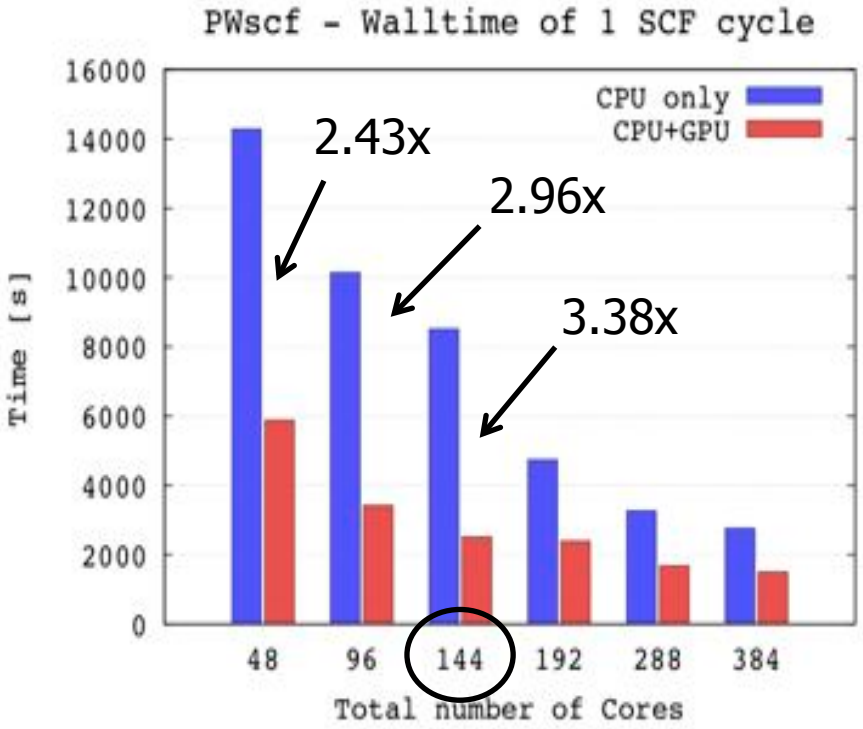
PWscf GPU, results & benchmarks
(serial – AUSURF112)



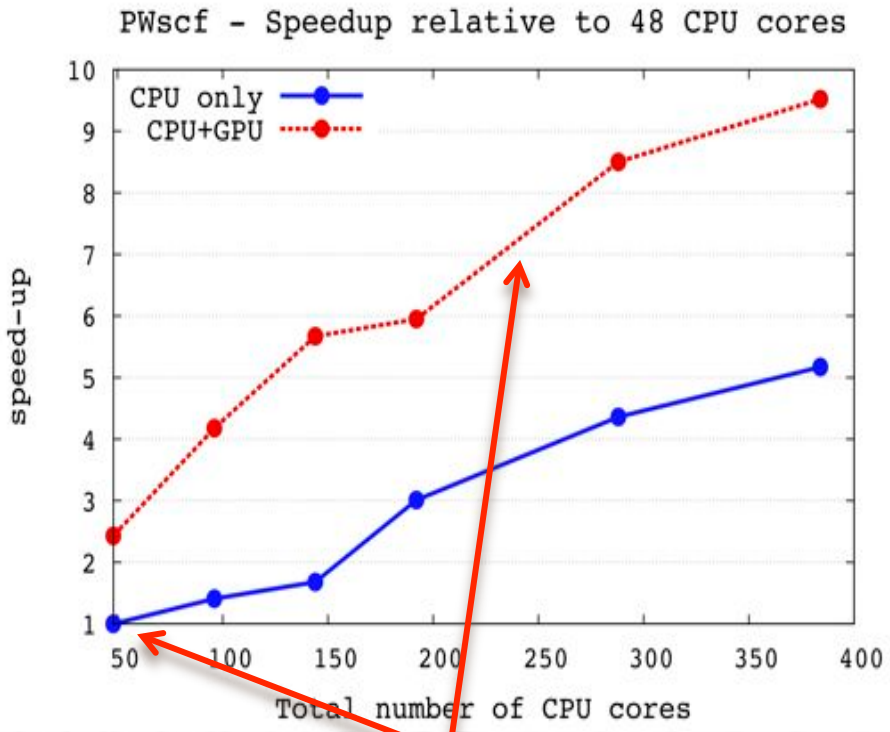


Please, be fair!

PWscf GPU, results & benchmarks
(parallel – MGST-hex)



72 MPI x 2 OpenMP + 24 GPUs
(GPU:MPI = 3:1)



Acceleration, not Scalability!

WHEN and WHY more MPI processes per GPU is good?

Usually CUDA optimizations are performed starting from a serial code

→ Visual Profiler (or directly on text file)

Introducing a parallelization means distribute data

→ compute-footprint of some CUDA kernels might decrease or transfer-time overcomes compute-time (even worst!)

GPU performs better its "duty" (accelerator) when there is enough computation to exploit all the parallelism of all SM

→ let's safely share it and its resources

Less speed-up of a single piece of computation but there is more interleaved work on the GPU coming from different processes

→ kernels are less efficient but more can run concurrently → PERFORMANCE!

Can I achieve a better acceleration?

ALWAYS! But remember that…

Amdahl law still exists!

Def: *In a massively parallel context, an upper limit for the scalability of parallel applications is determined by the fraction of the overall execution time spent in non-scalable operations.*

Thank you for your attention!

No CPUs or GPUs have been damaged during the preparation of this talk (-: