Parallelizing the computation of Green functions for computational electromagnetism problems

Carlos Pérez-Alcaraz, Domingo Giménez, Tomás Ramírez
Departamento de Informática y Sistemas, University of Murcia, Spain

Alejandro Álvarez-Melcón, Fernando D. Quesada
Departamento de Tecnología de la Información y las Comunicaciones, Polytechnic University of Cartagena, Spain

PDSEC Workshop, Shanghai, May 25, 2012

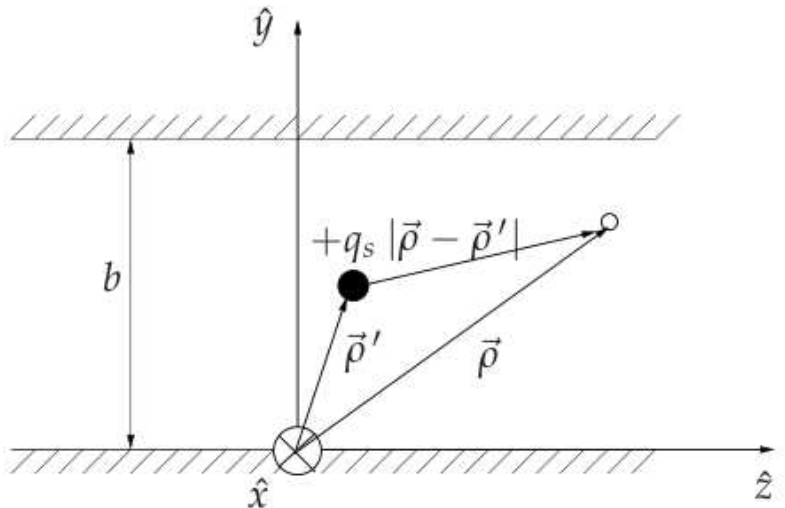
Outline

Green functions

- Used to solve non homogeneous differential equations with boundary conditions.
- Applied to waveguides, which are used in the design and analysis of integrated circuits MMIC (Monolithic Microwave Integrated Circuits).
- They can be expressed in the form of infinite series, in the spatial or spectral domain.
- It is necessary to calculate hundreds or thousands of Green functions.

Application to waveguides

- There is a parallel plate guide along the $\hat{z}$ axis.
- Inside this guide is a set of source and observer points which move in axes $\hat{y}$ and $\hat{z}$.
- The Green function associated to each pair of points is calculated.
- The two series in the Ewald method are computed.
- The number of terms can be fixed for all the pairs or be dynamically calculated as a function of the distance between the two points.



One-dimensional problem

{For each source point}
for $i = 1$ to m **do**
  {For each observer point}
  for $j = 1$ to n **do**
    {For the number of modes (terms)}
    {Calculation of summation in the spectral domain}
    for $k = 1$ to $nmod$ **do**
      trigonometric operations
    end for
    {Summation of the trigonometric functions}
    for $k = 1$ to $nmod$ **do**
      trigonometric operations
    end for
    Apply the method of acceleration of Kummer
  end for
end for

$$\text{Cost } O\left(m \cdot n \cdot nmod\right)$$

Two-dimensional problem

Initialization: obtain and sort modes
for $i = 1$ to m **do**
  for $j = 1$ to n **do**
    {Spectral part}
    for $k = 1$ to $nmod$ **do**
      $GF[i,j] += spectral(k)$
    end for
    {Spatial part}
    {For images in axes x and y}
    for $r = -mimag$ to $mimag$ **do**
      for $s = -imag$ to $nimag$ **do**
        $GF[i,j] += spatial(r,s)$
      end for
    end for
  end for
end for

Cost $O\left(m \cdot n \cdot mimag \cdot nimag\right)$

Systems

Low computational cost.
So, multicore+GPU versions are developed.

Experiments in the systems:

- `Luna`: 4 cores + NVIDIA GeForce 9800 GT, 112 cores
- `geatpc2`: 8 cores + NVidia Quadro FX 4600, 96 cores

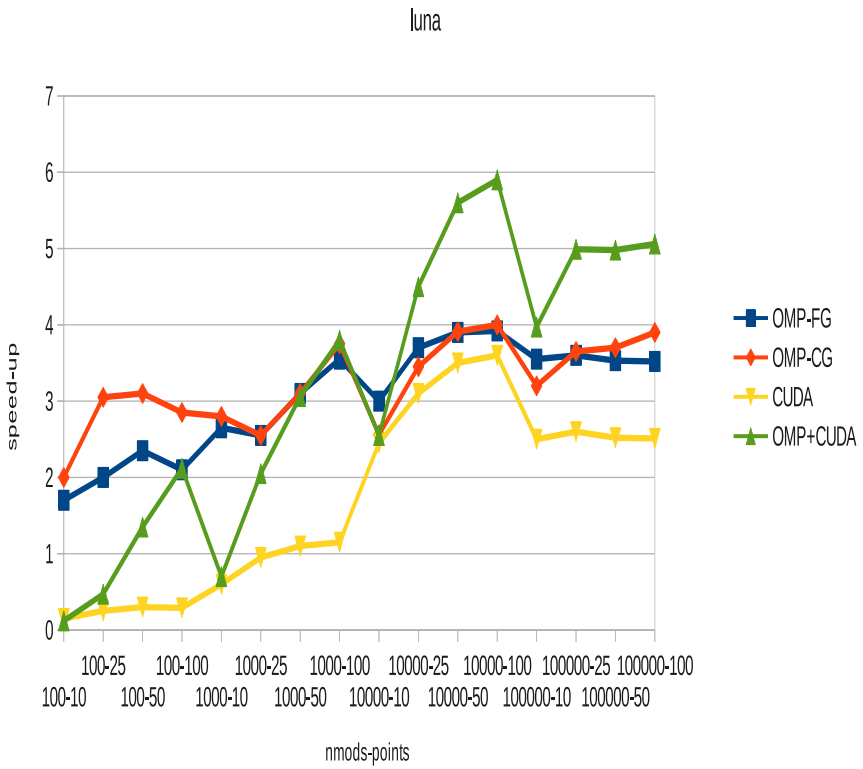
Computation of the spectral domain in CUDA

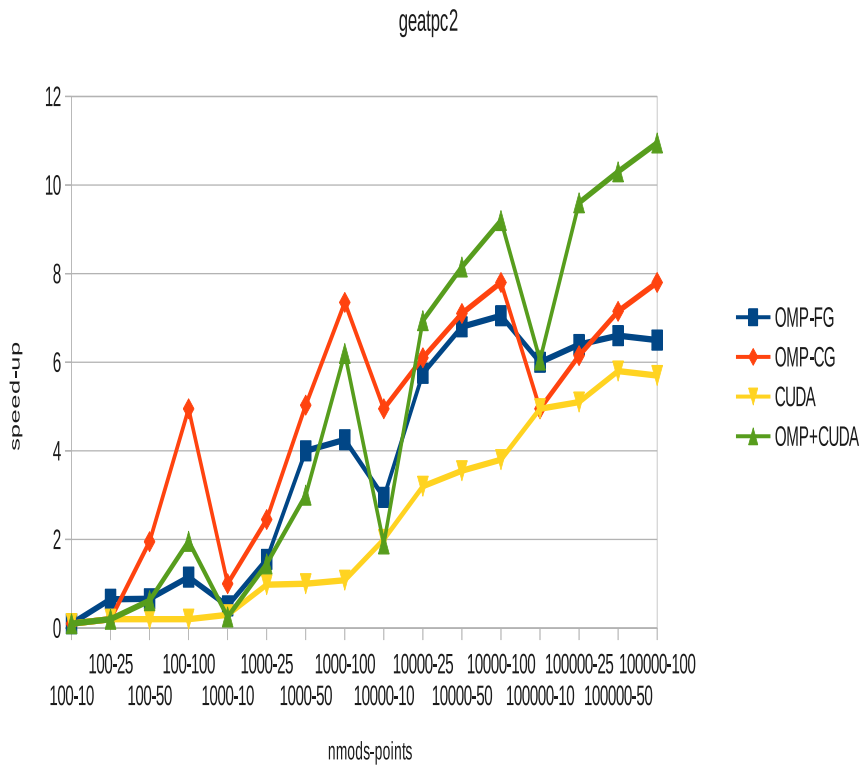
__global__ void spectgf_cuda_kernel (...):
$tn = THREADS\_PER\_BLOCK * blockId.y + threadId.y$
if $tn < nmod$ **then**
{Compute spectral domain Green functions}
$gf[tn, 0]$ = Direct function
$gf[tn, 1]$ = Derivative respect to z-axis
$gf[tn, 2]$ = Derivative respect to y-axis
end if
end_function
{The function calls the kernel}
void spectgf_cuda(...):
dim3 grid(1, $\lceil nmod / THREADS\_PER\_BLOCK \rceil$)
dim3 block(1, $THREADS\_PER\_BLOCK$)
spectgf_cuda_kernel<<<grid, block>>>(...)
end_function

One-dimensional implementations

- **1D-OMP-FG**: A fine grain version with OpenMP. The two innermost loops are parellelized.
- **1D-OMP-CG**: Coarse grain parallelism with OpenMP, parallelizing the work in the outermost loop.
- **1D-CUDA**: The computation of each Green function (fine grain parallelism) is performed by the GPU.
- **1D-OMP+CUDA**: Hybrid implementation. In a shared-memory program (with OpenMP) the number of threads generated is one more than the number of cores. One of the threads is in charge of calling the CUDA kernel. The other threads follow the coarse grain shared-memory scheme.

Speed-up





- For large problem sizes the speed-ups of the OpenMP implementations are satisfactory.
- The CUDA version has very low speed-up, with a high initialization cost.
- The best results are obtained by combining OpenMP and CUDA, with speed-ups higher than the number of cores.

Systems

Higher computational cost.
OpenMP, MPI and CUDA versions.

New systems considered:

- `Ben`: cc-NUMA with 128 cores
- `Arabí`: 102 nodes, each 8 cores
- `Hipatia`: 14 nodes, each 8 cores + 2 nodes, each 16 cores.
  2 nodes of 8 cores used

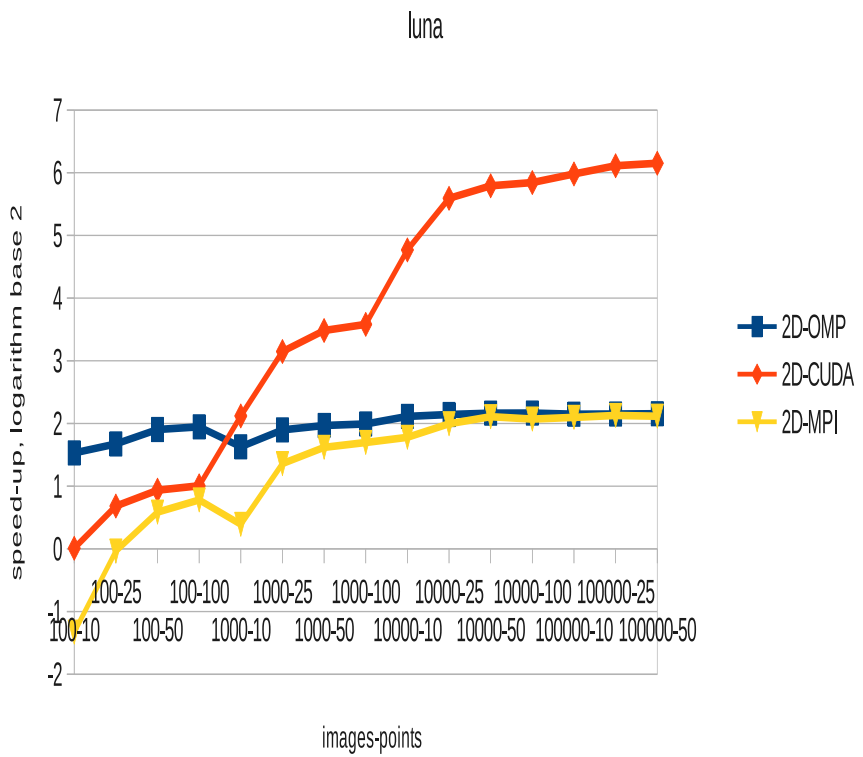
CUDA kernel for the computation of the spatial part

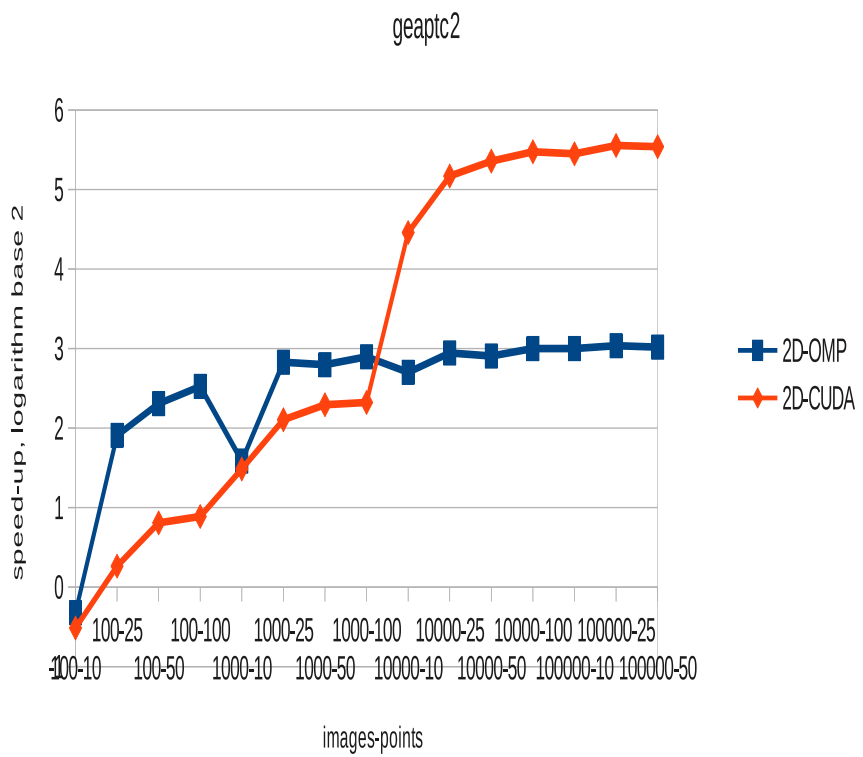
__global__ void spectgf_cuda_kernel (...):
mima = blockIdx.x {*mima* in [0,2*mimag*+1]}
nima = threadIdx.x {*nima* in [0,2*nimag*+1]}
mima -= gridDim.x / 2 {*mima* in [-*mimag*,*mimag*]}
nima -= blockDim.x / 2 {*nima* in [-*nimag*,*nimag*]}
...
Calculate spatial_gf(*mimag*, *nimag*)

Two-dimensional implementations

- **2D-OMP**: Parallelizing the first loop of the spatial part. The access to some variables to store partial results is done by reduction.
- **2D-CUDA**: Each thread is in charge of the computation of one image. An auxiliary matrix is used to store the partial sum obtained by each thread, and the values in the matrix are added sequentially.
- **2D-MPI**: The spatial part is parallelized, and the spectral part is done sequentially, like with OpenMP, and the final sum is obtained with `MPI_Reduce`.

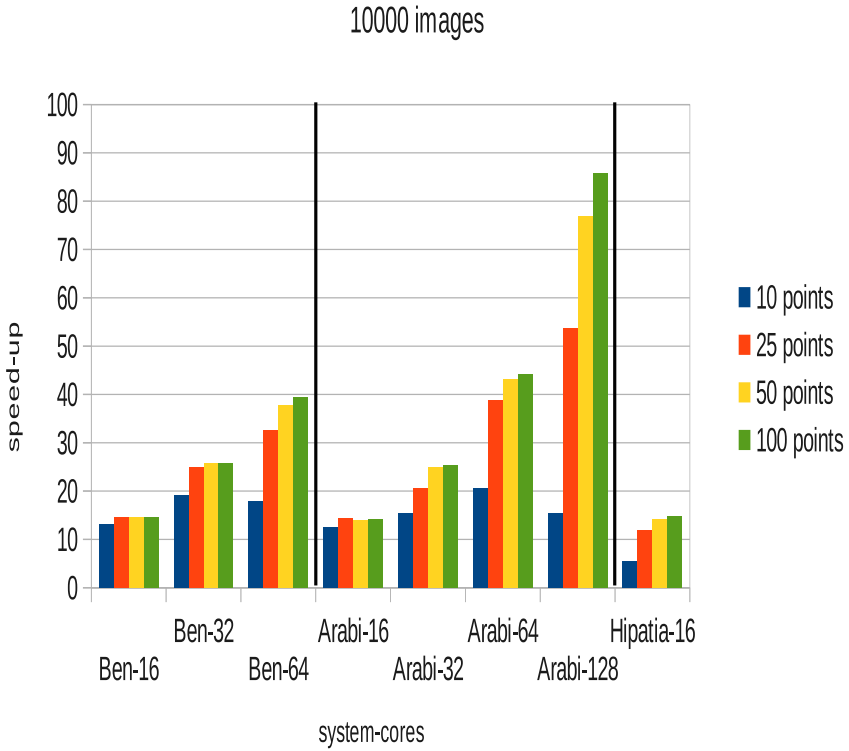
Speed-up

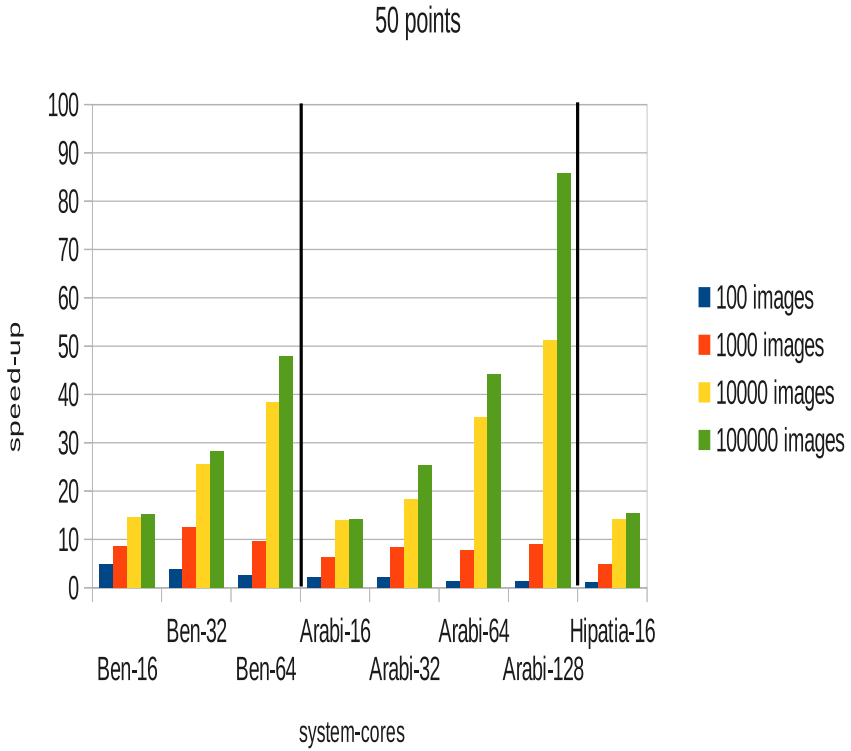




- Large speed-up of the CUDA algorithm for large problems.
- The points at which the CUDA algorithm is preferable change for different computational systems.

Scalability





- For very large problems it may be preferable to use a large system (shared-memory or distributed-memory), but this cames at a high cost.
- For small problems the best results are obtained with a number of threads or processes lower than the number of cores in the system, and the optimum number of processes changes for different systems.

Prob. size	dimension One	Two	Three
small	sequential	OpenMP	OpenMP+CUDA?
medium	OpenMP	OpenMP+CUDA	MPI?
large	OpenMP+CUDA	MPI	MPI+OpenMP+CUDA?

It is necessary to select

- the preferred algorithm
- the optimum number of cores

depending on the system and the problem size (the number of modes, points and images).

Modelling CPU+GPU computation ?

- Design: extend the ideas of modelling in multicore:

$$\frac{t_s}{c + s_{g/c}g} + t_{sc}c + t_{sk}g$$

  - t_s sequential time
  - c, g number of cores and of GPUs
  - $s_{g/c}$ speed-up of one GPU with respect to one core for the problem in question
  - t_{sc}, t_{sk} cost of generation of one core and one kernel

- Installation: use some installation methodology to estimate the values of the parameters in a particular system.
- Execution: for a particular input (problem size) and in a particular system (the computational system+the implemented algorithms) select the algorithm and the part of the computational system to use in the solution of the problem.

Modelling CPU+GPU computation ?

- Design: extend the ideas of modelling in multicore:

$$\frac{t_s}{c + s_{g/c}g} + t_{sc}c + t_{sk}g$$

  - t_s sequential time
  - c, g number of cores and of GPUs
  - $s_{g/c}$ speed-up of one GPU with respect to one core for the problem in question
  - t_{sc}, t_{sk} cost of generation of one core and one kernel

- Installation: use some installation methodology to estimate the values of the parameters in a particular system.
- Execution: for a particular input (problem size) and in a particular system (the computational system+the implemented algorithms) select the algorithm and the part of the computational system to use in the solution of the problem.

Modelling CPU+GPU computation ?

- Design: extend the ideas of modelling in multicore:

$$\frac{t_s}{c + s_{g/c}g} + t_{sc}c + t_{sk}g$$

  - t_s sequential time
  - c, g number of cores and of GPUs
  - $s_{g/c}$ speed-up of one GPU with respect to one core for the problem in question
  - t_{sc}, t_{sk} cost of generation of one core and one kernel

- Installation: use some installation methodology to estimate the values of the parameters in a particular system.
- Execution: for a particular input (problem size) and in a particular system (the computational system+the implemented algorithms) select the algorithm and the part of the computational system to use in the solution of the problem.