How Pencil could go GPU — a roadmap?

M. Rheinhardt

May 12, 2015

Pencil to GPU: Specification sheet

Necessary requirements for transition

Pencil to GPU: Specification sheet

Necessary requirements for transition

(i) preserve full (or almost full) functionality

Pencil to GPU: Specification sheet

Necessary requirements for transition

(i) preserve full (or almost full) functionality

(ii) preserve open source property,
avoid dependence on specific vendor or architecture

Pencil to GPU: Specification sheet

Necessary requirements for transition

(i) preserve full (or almost full) functionality

(ii) preserve open source property,
avoid dependence on specific vendor or architecture

(iii) avoid extensive manual recoding

Pencil to GPU: Specification sheet

Necessary requirements for transition

(i) preserve full (or almost full) functionality

(ii) preserve open source property,
avoid dependence on specific vendor or architecture

(iii) avoid extensive manual recoding

(iv) achieve significant speedup (say > 10)

Pencil to GPU: Specification sheet

Necessary requirements for transition

(i) preserve full (or almost full) functionality

(ii) preserve open source property,
avoid dependence on specific vendor or architecture

(iii) avoid extensive manual recoding

(iv) achieve significant speedup (say > 10)

(v) preserve coherence and extensibility

Pencil to GPU: Specification sheet

Necessary requirements for transition

(i) preserve full (or almost full) functionality

(ii) preserve open source property,
avoid dependence on specific vendor or architecture

(iii) avoid extensive manual recoding

(iv) achieve significant speedup (say > 10)

(v) preserve coherence and extensibility

Disregarded for today

- particles
- CPU parallelization

Pencil to GPU: Specification sheet

from (ii)

- CUDA disfavored:
  tied to NVIDIA hardware,
  CUDA-Fortran not free
- OpenCL: a standard for programming of heterogeneous systems (GPU, multi-core CPU)

  Pro:
  - $\exists$ implementations for several platforms (NVIDIA, AMD, Intel)
  - $\exists$ FortranCL
    – a free wrapper library, easy to extend

  Con: performance perhaps worse vs. CUDA, implementation–dependent

Pencil to GPU: Specification sheet

from (ii)

- CUDA disfavored:
  tied to NVIDIA hardware,
  CUDA-Fortran not free
- OpenCL: a standard for programming of heterogeneous systems (GPU, multi-core CPU)

  Pro:
  - $\exists$ implementations for several platforms (NVIDIA, AMD, Intel)
  - $\exists$ FortranCL
    – a free wrapper library, easy to extend

  Con: performance perhaps worse vs. CUDA, implementation–dependent

from(iii):

- manual rewriting of only a small part of code acceptable
- even FortranCL requires kernels *to be written in C*
- $\Longrightarrow$ bulk of code transformation to be done *by a program*

Pencil to GPU: Specification sheet

from (iv)

- bulk of numerical work to be brought to GPU
  $\Longrightarrow \pm$ everything inside time-loop
- pursue optimum use of GPU memory
- exploit CPU-GPU concurrency

Pencil to GPU: Specification sheet

from (iv)

- bulk of numerical work to be brought to GPU
  $\Longrightarrow \pm$ everything inside time-loop
- pursue optimum use of GPU memory
- exploit CPU-GPU concurrency

from (i) and (v):

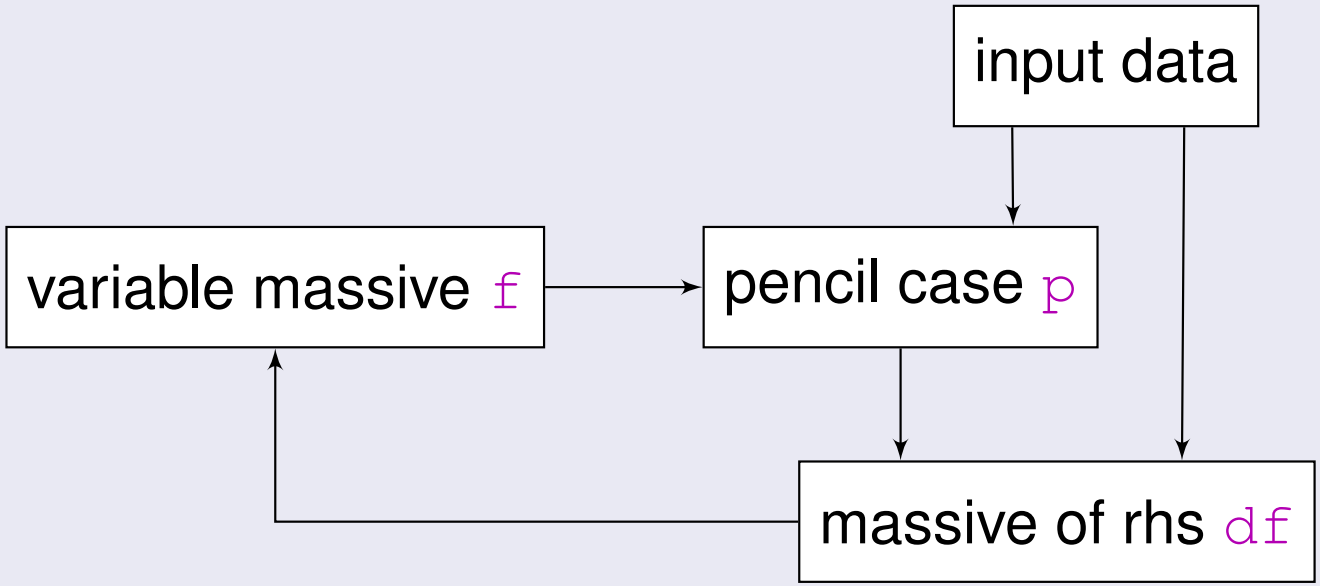
- no branching!
  $\Longrightarrow$ a switchable GPU module,
  $+$ few `if (lgpu)` clauses
- maintain code structure, in particular
  - pencil concept
  - switchable modules
  - flexibility of spatial and temporal discretizations

Pencil to GPU: A look at the code

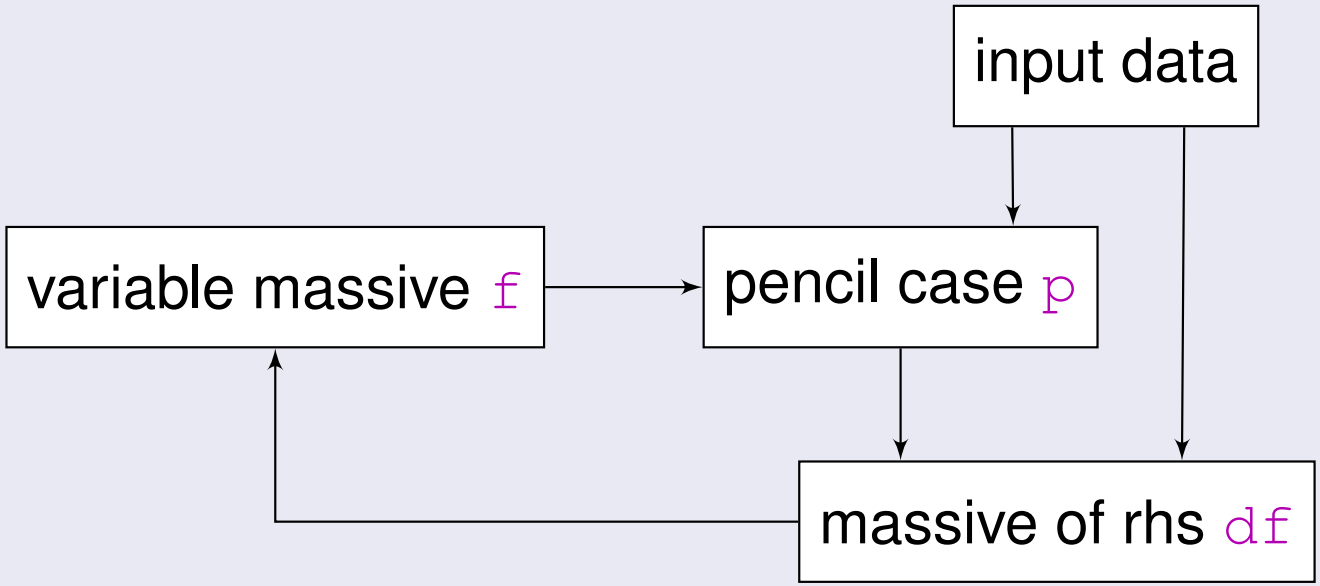
Structure of time loop body (simplified)

```
while not <time limit>
  df = 0
  do i = 1,n_substeps
    apply boundary conditions on system variables in array f
    do n = 1,nzgrid
      do m = 1,nygrid
        calculate all pencils in structure p ("pencil case")
        use p for cumulatively constructing rhs of pde system
        in array df
      enddo
    enddo
    estimate optimum time step dt
    f = f+beta(i)*dt*df
  enddo
endwhile
```

Flow of data


input data
variable massive f
pencil case p
massive of rhs df

Flow of data



Sizes

- input data: $<$ 1000 scalars,
  $<$ 100 1D arrays, size `nxgrid`, `nygrid` or `nzgrid`
- variable massive `f`: `nxgrid` $\times$ `nygrid` $\times$ `nzgrid`
  $\times$ number of variables
- pencil case `p`: `nxgrid` $\times$ number of pencils
- massive of rhs `df`: `nxgrid` $\times$ `nygrid` $\times$ `nzgrid`
  $\times$ number of advanced variables

Typical code

pencil calculation (`calc_pencils_*` subroutines)

```
if (lpencil(i_uu)) p%uu = f(l1:l2,m,n,iux:iuz)
if (lpencil(i_u2)) call dot2_mn(p%uu,p%u2)
if (lpencil(i_uij5)) call gij(f,iuu,p%uij5,5)
if (lpencil(i_sij)) call traceless_strain
      (p%uij,p%divu,p%sij,p%uu,lshear_rateofstrain)
```

Typical code

pencil calculation (`calc_pencils_*` subroutines)

```
if (lpencil(i_uu)) p%uu = f(l1:l2,m,n,iux:iuz)
if (lpencil(i_u2)) call dot2_mn(p%uu,p%u2)
if (lpencil(i_uij5)) call gij(f,iuu,p%uij5,5)
if (lpencil(i_sij)) call traceless_strain
      (p%uij,p%divu,p%sij,p%uu,lshear_rateofstrain)
```

rhs calculation (`d*_dtt` subroutines)

```
if (ladvection_velocity) df(l1:l2,m,n,iux:iuz) =
      df(l1:l2,m,n,iux:iuz) - p%ugu
if (ekman_friction /= 0) df(l1:l2,m,n,iux:iuz) =
      df(l1:l2,m,n,iux:iuz) - ekman_friction*p%uu
```

Typical code

pencil calculation (`calc_pencils_*` subroutines)

```
if (lpencil(i_uu)) p%uu = f(l1:l2,m,n,iux:iuz)
if (lpencil(i_u2)) call dot2_mn(p%uu,p%u2)
if (lpencil(i_uij5)) call gij(f,iuu,p%uij5,5)
if (lpencil(i_sij)) call traceless_strain
      (p%uij,p%divu,p%sij,p%uu,lshear_rateofstrain)
```

rhs calculation (`d*_dtt` subroutines)

```
if (ladvection_velocity) df(l1:l2,m,n,iux:iuz) =
      df(l1:l2,m,n,iux:iuz) - p%ugu
if (ekman_friction /= 0) df(l1:l2,m,n,iux:iuz) =
      df(l1:l2,m,n,iux:iuz) - ekman_friction*p%uu
```

- either copy from `f` or differential operation (on `p` or `f`)
  or linear combination of pencils;
  only a few (mostly scalar) input data used

PC transformation: Naïve approach

PC transformation: Naïve approach

- algorithm has *strict SIMD property* w.r.t. x, y, z
- $\Longrightarrow$ simplest: parallelize on GPU w.r.t. x (pencil direction)
- $\Longrightarrow$ rewrite all differential operations,
  including underlying difference formulae, as kernels;
  replace each operation on `p` and `df` by kernel call

PC transformation: Naïve approach

- algorithm has *strict SIMD property* w.r.t. x, y, z
- $\Longrightarrow$ simplest: parallelize on GPU w.r.t. x (pencil direction)
- $\Longrightarrow$ rewrite all differential operations,
  including underlying difference formulae, as kernels;
  replace each operation on `p` and `df` by kernel call

Pro:

- no need to copy input data to GPU,
  appear as actual parameters in kernel calls
- most code changes local, only modules `Deriv` and `Sub`
  need to be dubbed in (OpenCL-) C

PC transformation: Naïve approach

- algorithm has *strict SIMD property* w.r.t. x, y, z
- $\Longrightarrow$ simplest: parallelize on GPU w.r.t. x (pencil direction)
- $\Longrightarrow$ rewrite all differential operations,
  including underlying difference formulae, as kernels;
  replace each operation on `p` and `df` by kernel call

Pro:

- no need to copy input data to GPU,
  appear as actual parameters in kernel calls
- most code changes local, only modules `Deriv` and `Sub`
  need to be dubbed in (OpenCL-) C

Con:

- number of threads $=$ `nxgrid`, but should be $\lesssim$ 10,000
- use of shared memory for `p`, `f`, `df` ruled out
  as scope and lifetime limited to those of kernel
- $\Longrightarrow$ optimum speedup not achievable

PC transformation: Doing better

- simplest: parallelize w.r.t. x and y with number of threads
  $=$ `nxgrid*(nygrid/ystep)`
  $\Longrightarrow$ `p` inflates by `nygrid/ystep`

  `ystep`: from trade-off between number of threads
  and available shared memory
- $\Longrightarrow$ all code within `mn` loop to be converted in *one kernel*
  $\Longrightarrow$ much more/more complex code to be touched
- `p` and `df` should (can?) sit completely in shared memory
  `f` perhaps not, as access is not frequent
  (rare for calculation of `p` and `df`
  $+$ once in each substep for integration)
  minimize by use of pencils
- needed input data to be transferred
  into GPU's constant (global) memory

PC transformation: Subtasks

- develop optimal recipe for shared memory use depending on device limitations and problem size
  (↗ “rolling cache”, experience from other GPU codes)
- code modules `Deriv` and `Sub` (conservative!) in OpenCL-C; make use of specific vector data types; promote use of registers ($\Longrightarrow$ avoid local arrays)
- comb manually through all `calc_pencils_*` and `d*_dtt` subroutines and their callees for
  - purification from any `nm` constant operations
  - minimization of auxiliary arrays (favor in-place work)
  - removal of non-standard behavior (early `f` modification)
  - insertion of helper directives for code converter
- code a module `GPU` for initializing GPU use (FortranCL)
- create a *code converter* for `calc_pencils_*` and `d*_dtt` subroutines and their callees

PC transformation: The GPU module

Tasks

- determines GPU type and memory limitations
- determines optimum scheme of GPU memory use
- initializes GPU memory with input data (grid, pencil mask etc.) in constant memory,
- performs data exchange between CPU and GPU (global memory) for `f` array
- loads GPU program (precompiled OpenCL-C code)

PC transformation: The code converter

Tasks

- generates C constant definitions for all compile-time constants, e.g. `i<variable name>`, `i_<pencil name>`, `l<module name>` (from `cparam.f90`, `cparam.local`, `cparam.inc`)
- generates C constant definitions for accessing pencil case `p` as an array (from `cparam_pencils.inc`)
- parses for
  - output (usually informative) $\Longrightarrow$ remove
  - error handling $\Longrightarrow$ modify into "return with error code"
  - used input data: generate constant definitions, e.g. `#define l_<name>` for any logicals `l<name>` `#define i[xyz]_<name>` for any vector of dimension `n[xyz]grid` etc.
    $\Longrightarrow$ generate array references

PC transformation: The code converter

More tasks

- collects all `calc_pencils_*` and `d*_dtt` calls into one kernel function (OpenCL C)
- transforms all callees into device functions
- generates code for declaration of global-scope data on GPU (avoids excessive parameter lists) and transfer from CPU to GPU (Fortran)
- replace global reduction operations by dedicated OpenCL library calls

PC transformation: The code converter

More tasks

- collects all `calc_pencils_*` and `d*_dtt` calls into one kernel function (OpenCL C)
- transforms all callees into device functions
- generates code for declaration of global-scope data on GPU (avoids excessive parameter lists) and transfer from CPU to GPU (Fortran)
- replace global reduction operations by dedicated OpenCL library calls

How to be implemented?

- preprocessor — Fortran to C converter — postprocessor
- *processor: Perl script?
- Fortran to C converter: open source?, yacc, lex ? (no experience)

PC transformation: keep CPU busy

Concurrent with GPU calculations

- CPU–GPU data transfer
- application of boundary conditions
- I/O and related calculations
- calculation of diagnostics, in particular averages

via *non-blocking OpenCL command queue*

Proposals most welcome !!!