PHSP'14

A parallel functional language for high performance finite difference stencil codes

Gerhard Zumbusch
Institut für Angewandte Mathematik

Friedrich-Schiller-Universität Jena

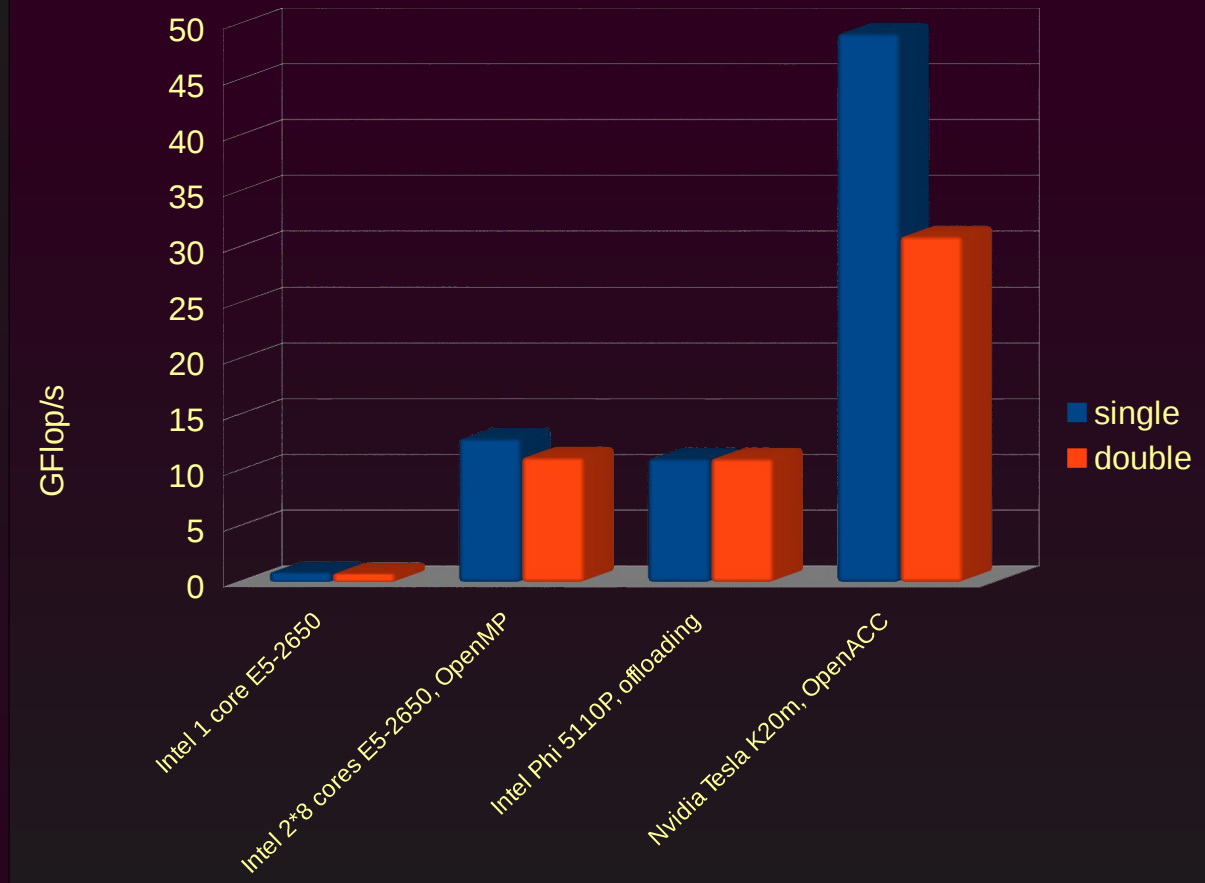
since 1558

Results: standard/naive/textbook loop

1D scalar,
constant coefficient
heat equation,
periodic boundary conditions,
explicit finite differences

$$u_{k,i} = r \cdot (u_{k-1,i-1} + u_{k-1,i+1}) + (1 - 2r)u_{k-1,i}$$

```
time loop k {
  space loop i {
    difference stencil
  }
}
```



instruction count

$$u_{k,i} = r \cdot (u_{k-1,i-1} + u_{k-1,i+1}) + (1 - 2r)u_{k-1,i}$$

```
vaddps  (%rdx),%ymm1,%ymm1
vmulps  (%rsi),%ymm0,%ymm0
vmulps  0x2e00(%rip),%ymm1,%ymm1
vaddps  %ymm1,%ymm0,%ymm0
```

} 1 cycle

} 1 cycle

1 add + 1 mult per cycle

128 bit SSE vector = 4 float or 2 double

128 bit ARM Neon vector = 4 float

Intel >= Sandy Bridge, AMD >= Bulldozer:

256 bit AVX vector = 8 float or 4 double

```
vaddps %ymm3, %ymm4, %ymm3
vmulps %ymm6, %ymm3, %ymm3
vfmadd132ps %ymm5, %ymm3, %ymm0
```

} 1 FMA

} 1 FMA

} 1 FMA

fused-multiply-add (FMA):

Nvidia Fermi GPU, warp = 32 float, 1FMA/cycle

Nvidia Kepler GPU, warp = 32 float, 6FMA/cycle

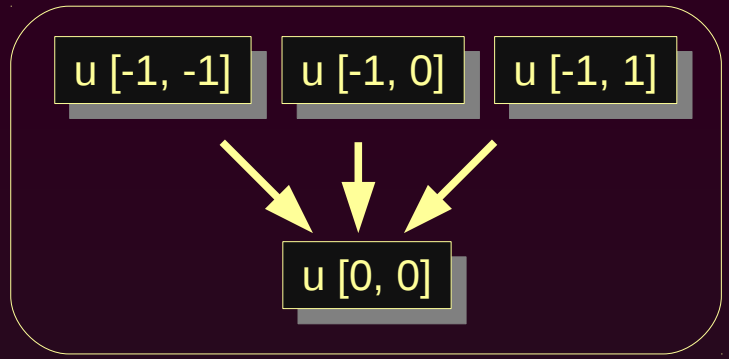
AMD >=Bulldozer, AVX, 1FMA/cycle

Intel Haswell, AVX, 2FMA/cycle

Intel Phi, 16 float or 8 double, 1FMA/cycle

scalar loop

```
// finite difference stencil
float calc (float u[]) {
  return a0 * u[1] + a1 * ( u[0] + u[2] );
}
// spatial loop
for (int x=0; x<stepx; x++) {
  u2[x] = calc (u[x-1..x+1]);
}
u = u2;
```



3 load

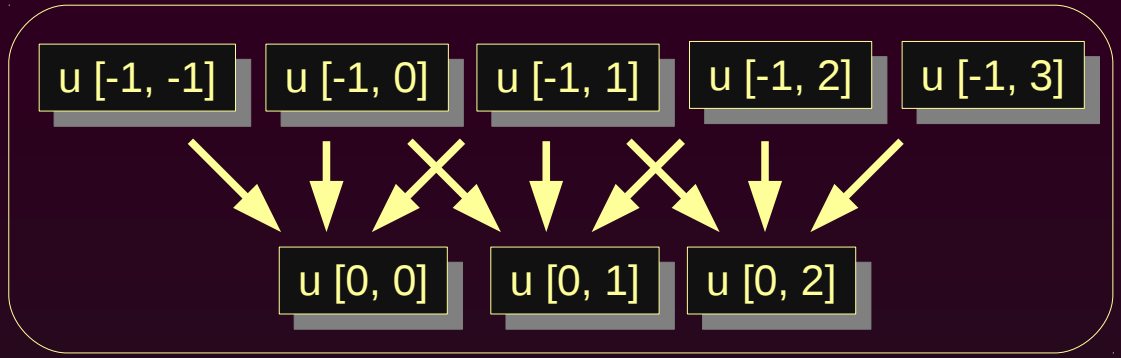
2 add, 2 mult

1 store

→ 1 flop/mem op

sliding window

```
r0 = u[0]; r1 = u[1];       // load memory
for (int x=0; x<stepx*3; x=x+3) {
  r2 = u[x+2];              // load memory
  u[x ] = calc (r0, r1, r2);   // store memory
  r0 = u[x+3];              // load memory
  u[x+1] = calc (r1, r2, r0);    // store memory
  r1 = u[x+4];              // load memory
  u[x+2] = calc (r2, r0, r1);    // store memory
}
```



N+2 load

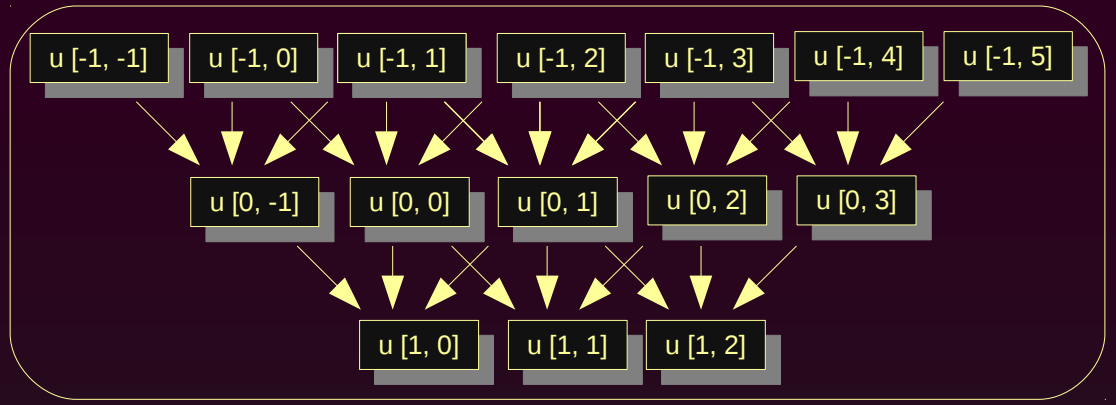
2N add, 2N mult

N store

→ 2 flop/mem op

space-time

T iterations



N+2T load

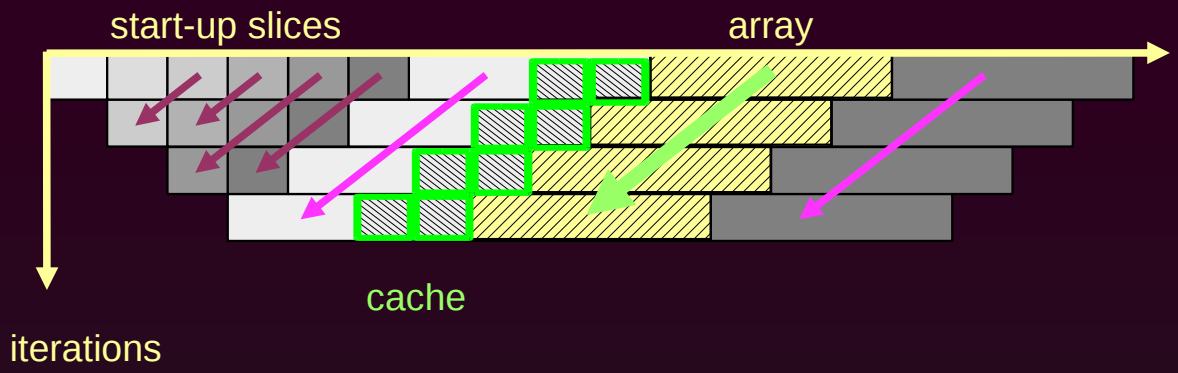
2NT+2+T² add, 2NT+2+T² mult

N store

→ 2T flop/mem op

space-time-slicing (2)

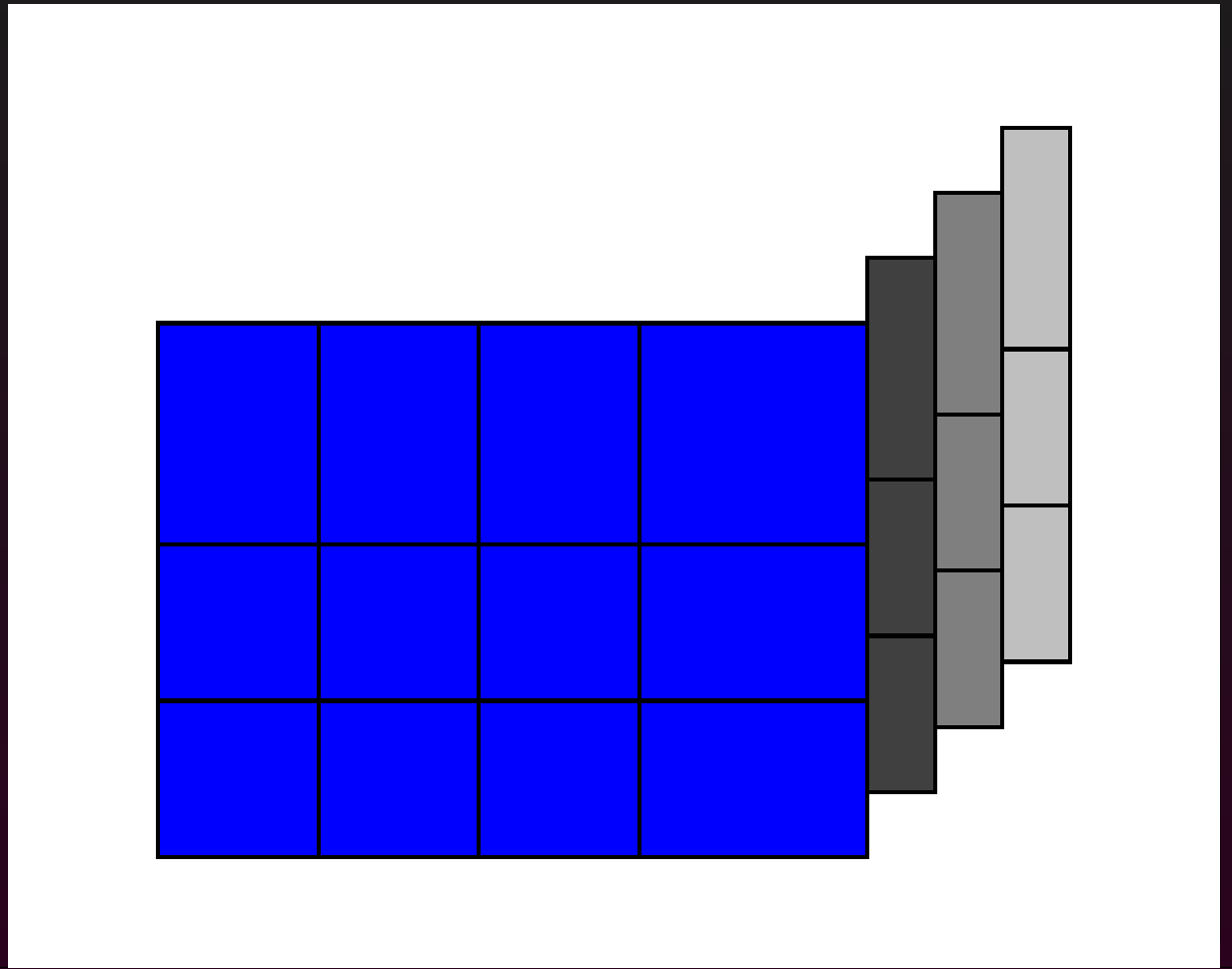
```
// p stencil width
// s slice width
for (i=(stepi-1)*s; i>=0; i=i-s) {
  // load memory
  cache[0..s-1] = u[i..i+s-1];
  for (k=0; k<stepk*p; k=k+p) {
    // load cache
    cache[s..s+p-1] = u[i+k+s..i+k+s+p-1];
    cache[0..s-1] = stencil (cache[0..s+p-1]);
    // store cache
    u[i+k+p..i+k+2*p-1] = cache[0..p-1];
  }
  // store memory
  u[i+(stepk+1)*p..i+stepk*p+s-1] = cache[p..s-1];
}
```



2NT/s+T² cache load
2NT/s cache store
N+2T global load
N global store
NT+T² stencil ops

space-time slicing 2D

animation

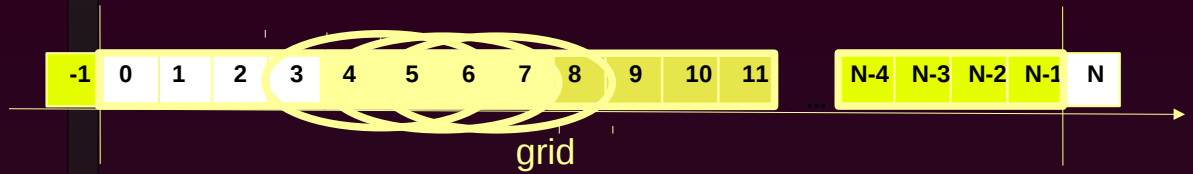


vectorization strategies

```
for (int i=0; i<n; i+=l)

 u2[i] = stencil_vec (
          load_unaligned (u[i-1]),
          load (u[i]),
          load_unaligned (u[i+1]) );
```

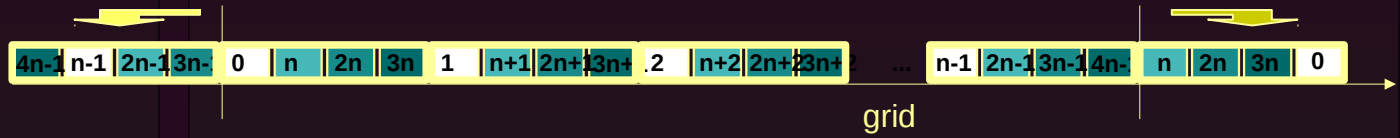
unaligned memory access




```
vec u0 = rotate_right (u[n-1]);
vec u1 = u[0];
u[n] = rotate_left (u1);

for (int i=0; i<n; i++) { // unroll
  vec u2 = u[i+1];
  v[i] = stencil_vec (u0, u1, u2);
  u0 = u1; u1 = u2;
}
```

interleaved memory layout

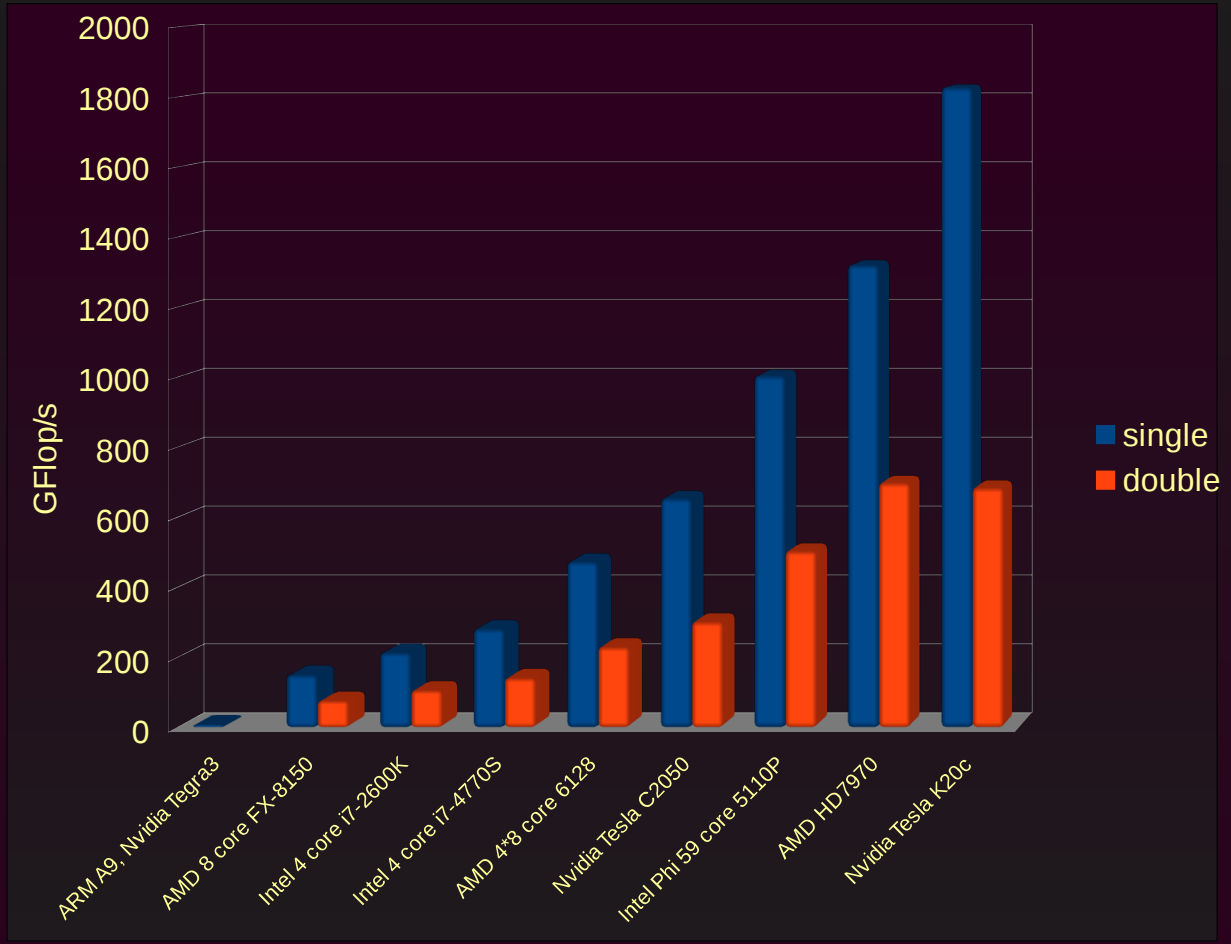


Results: (hand) optimized Space-Time Slicing (+auto tuning)

1D scalar,
constant coefficient
heat equation,
periodic boundary conditions,
explicit finite differences

$$u_{k,i} = r \cdot (u_{k-1,i-1} + u_{k-1,i+1}) + (1 - 2r)u_{k-1,i}$$

- space-time loop
- vectorization
- OpenMP parallel
- loop tiling
- optimized parameters



Language: stencil example (> code generation)

differential equation:
heat transfer

$$\frac{\partial}{\partial t}u = \frac{\partial^2}{\partial x^2}u$$

explicit finite difference stencil

$$u_{k,i} = r \cdot (u_{k-1,i-1} + u_{k-1,i+1}) + (1 - 2r)u_{k-1,i}$$

C style standard loops

```
r = .25;
for (k=0; k<20; k++) {
  u [k, 0] = u [k, 1023] = 0;
  for (i=1; i<1023; i++)
    u [k, i] = r * (u [k-1, i-1] + u [k-1, i+1])
             + (1-2*r) * u [k-1, i];
}
```

```
heat [k, i] {  //define a kernel, kernel name, iteration variables
               // k time dimension
               // i one space dimension
    r = .25;   // CFL constant
    u [k, i] = // array u. scalar field in space/time
      if (k <0, // if
        1, // then initial data
        if (i <= 0 || i >= 1023, // else if left/ right boundary
            0.0, // then zero boundary values
              // else stencil operator:
            r * (u [-1, -1] + u [-1, 1]) + (1-2*r) * u [-1, 0]
             //          left,       right,          center
             //      old time step
        )
    );
}
```

Language: stencil Example (2)

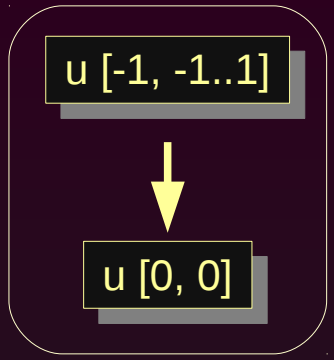
C style syntax
functional language
- single static assignement
- assignement order not specified
- ~ shader + data dependency

instruction order not specified
results can be computed
- once
- in parallel
- cached, pre-fetched
- or re-computed

storage scheme not specified
values can be stored in
- global, shared memory
- local memory, cache
- register

```
heat [k, i] {  //define a kernel, kernel name, iteration variables
               // k time dimension
               // i one space dimension
    r = .25;   // CFL constant
    u [k, i] = // array u. scalar field in space/time
      if (k <0, // if
        1, // then initial data
        if (i <= 0 || i >= 1023, // else if left/ right boundary
            0.0, // then zero boundary values
              // else stencil operator:
            r * (u [-1, -1] + u [-1, 1]) + (1-2*r) * u [-1, 0]
             //          left,       right,          center
             //      old time step
        )
    );
}
```

Language: stencil Example (3)

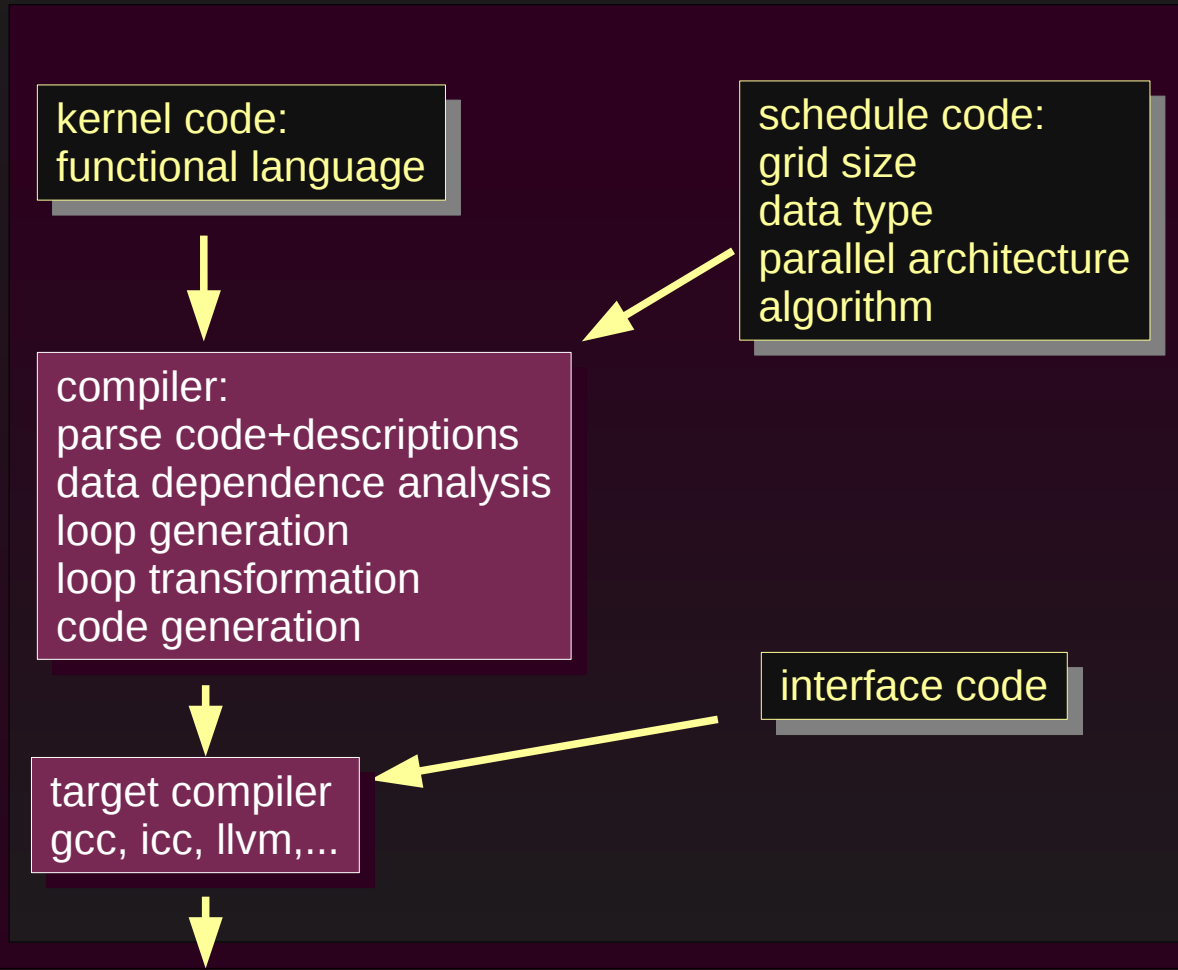



```
heat [k, i] {  //define a kernel, kernel name, iteration variables
               // k time dimension
               // i one space dimension
    r = .25;   // CFL constant
    u [k, i] = // array u. scalar field in space/time
      if (k <0, // if
        1, // then initial data
        if (i <= 0 || i >= 1023, // else if left/ right boundary
            0.0, // then zero boundary values
              // else stencil operator:
            r * (u [-1, -1] + u [-1, 1]) + (1-2*r) * u [-1, 0]
             //          left,       right,          center
             //      old time step
        )
    );
}
```

Compiler

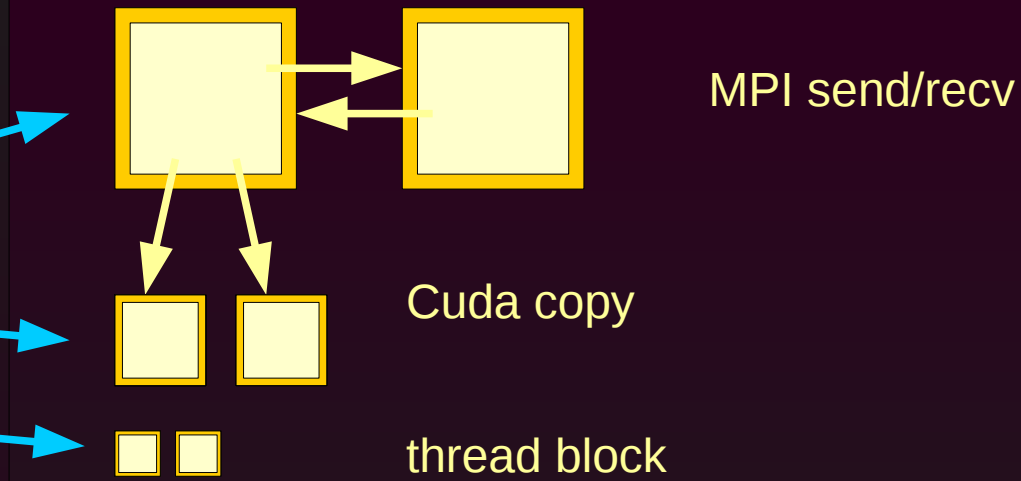
code generation:

separate the specification of
- kernel code
- data distribution
- data storage
- parallelization strategy
- vectorization strategy
- loop and instruction order
- code optimization
- grid size
- data type (float/ double)



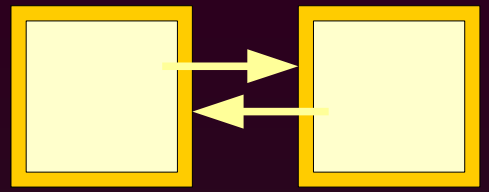
schedule code: examples

```
// start and number of time steps
time 0, 128;
// grid size and data type
domain 0, 65536, 0, 65536 float;
// 2*2 MPI processes
split no 2, 2 mpi block;
// 16 GPU multiprocessors
split no 2, 8 gpu cuda;
// GPU threads / vectors
split size 32, 8 gputhread;
```



schedule code: examples

```
// start and number of time steps
time 0, 128;
// grid size and data type
domain 0, 65536, 0, 65536 float;
// 2*2 MPI processes
split no 2, 2 mpi block;
// multi-threading in 2nd dimension
split no 1, 8 thread openmp;
// space-time-skewing
timesplit 32;
// loop unroll
split size 32, 2 tile unroll;
// vectorization in 1st dimension
split size 8, 1 simdinter avx;
```

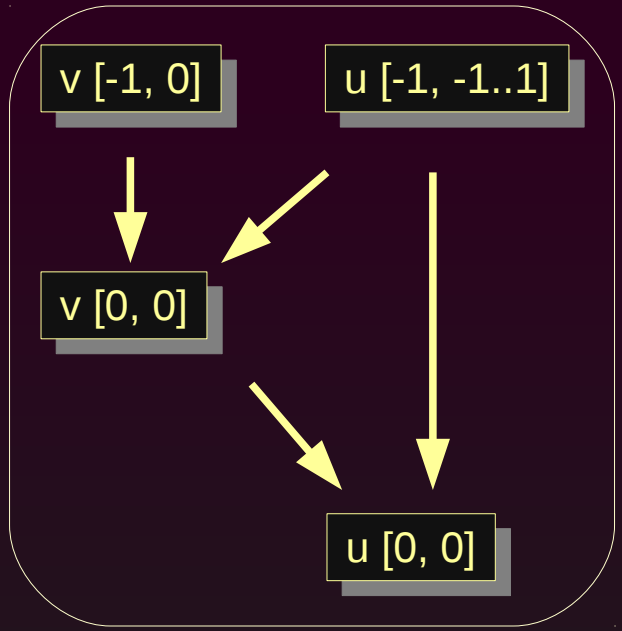


MPI send/recv

Multi threading

AVX vectors

Language: Systems of Equations



loop k

```
wave [t, i] {
    dx = 3.1415927 / (1024 - 1);
    dt = .5 * dx;
    r = dt / sqr (dx);
    v [t, i] =  // v field component in space/time
      if (t < 0 || i <= 0 || i >= 1023,
        0.0,
        dt * v [-1, 0] + r * (u [-1, -1] + u [-1, 1]) - 2 * r * u [-1, 0]
    );    // old u values
    u [t, i] =  // u field component in space/time
      if (t < 0 || i <= 0 || i >= 1023,
        sin (i * dx),
        u [-1, 0] + dt * v [0, 0]
    );    // current v, old u values
}
```

Language: Multi-Stages

$$\frac{\partial}{\partial t}u = \frac{\partial^2}{\partial x^2}u$$ heat transfer

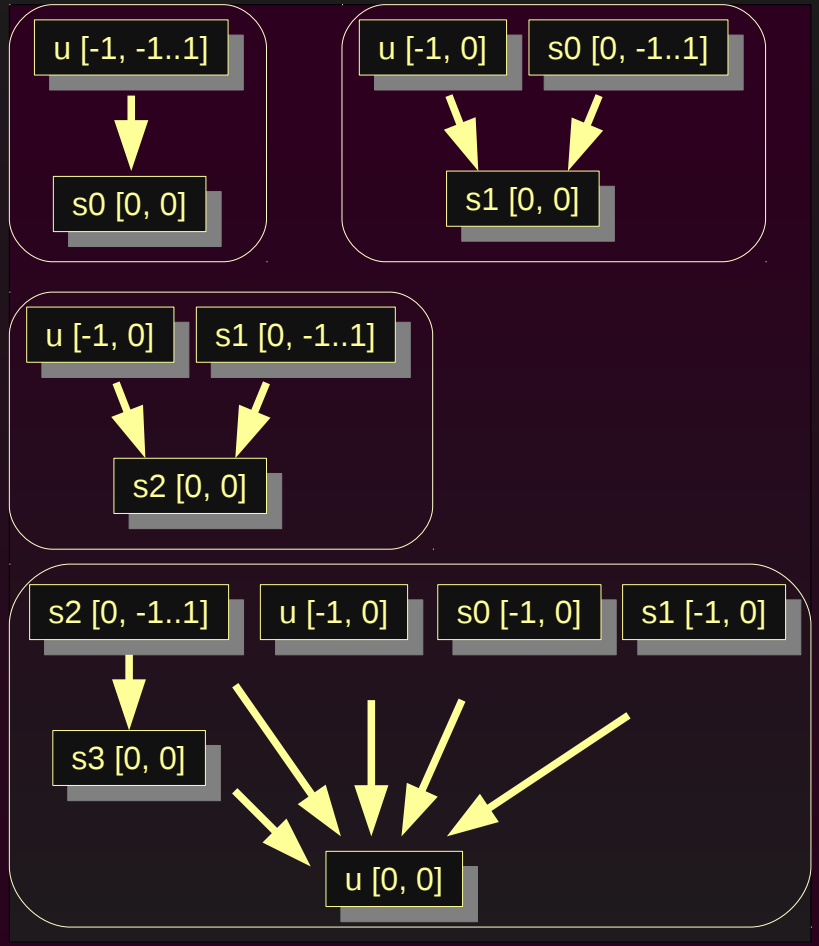
4-stage Runge-Kutta scheme

$$\begin{aligned}
k_0 &= \frac{\partial^2}{\partial x^2}u_{i-1} \\
k_1 &= \frac{\partial^2}{\partial x^2}\left(u_{i-1} + \frac{\Delta t}{2}k_0\right) \\
k_2 &= \frac{\partial^2}{\partial x^2}\left(u_{i-1} + \frac{\Delta t}{2}k_1\right) \\
k_3 &= \frac{\partial^2}{\partial x^2}\left(u_{i-1} + (\Delta t)k_2\right) \\
u_i &= u_{i-1} + \frac{\Delta t}{6}\left(k_0 + 2k_1 + 2k_2 + k_3\right)
\end{aligned}$$

$$\begin{aligned}
s0_i &= (1-r)u_{k-1,i} + \tfrac{r}{2}(u_{k-1,i-1} + u_{k-1,i+1}) \\
s1_i &= u_{k-1,i} - r \cdot s0_i + \tfrac{r}{2}(s0_{i-1} + s0_{i+1}) \\
s2_i &= u_{k-1,i} - 2r \cdot s1_i + r(s1_{i-1} + s1_{i+1}) \\
s3_i &= -r \cdot s2_i + \tfrac{r}{2}(s2_{i-1} + s2_{i+1}) \\
u_{k,i} &= \tfrac{1}{3}(s0_i + 2s1_i + s2_i + s3_i - u_{k-1,i})
\end{aligned}$$

```
RK4 [t, i] {
    r = .25;
    s0 [t, i] = if (i <= 0 || i >= 1023, // stage 0
        0.0,
        (r / 2) * (u [-1, -1] + u [-1, 1]) + (1 - r) * u [-1, 0]
    );    // t old u values
    s1 [t, i] = if (i <= 0 || i >= 1023, // stage 1
        0.0,
        u [-1, 0] + (r / 2) * (s0 [0, -1] + s0 [0, 1]) - r * s0 [0, 0]
    );    // t old u and new s0 values
    s2 [t, i] = if (i <= 0 || i >= 1023, // stage 2
        0.0,
        u [-1, 0] + r * (s1 [0, -1] + s1 [0, 1]) - (2 * r) * s1 [0, 0]
    );    // t old u and new s1 values
    s3 [t, i] = if (i <= 0 || i >= 1023, // stage 3
        0.0,
        (r / 2) * (s2 [0, -1] + s2 [0, 1]) + r * s2 [0, 0]
    );    // t old u and new s2 values
    u [t, i] = if (t < 0,                // new value
        1.0,
        (s0 [0, 0] + 2 * s1 [0, 0] + s2 [0, 0] + s3 [0, 0] - u [-1, 0]) / 3
    );    // t old u and new s0, s1, s2, s3 values
}
```

Language: Multi-Stages (2)




```
RK4 [t, i] {
    r = .25;
    s0 [t, i] = if (i <= 0 || i >= 1023, // stage 0
        0.0,
        (r / 2) * (u [-1, -1] + u [-1, 1]) + (1 - r) * u [-1, 0]
    );    // t old u values
    s1 [t, i] = if (i <= 0 || i >= 1023, // stage 1
        0.0,
        u [-1, 0] + (r / 2) * (s0 [0, -1] + s0 [0, 1]) - r * s0 [0, 0]
    );    // t old u and new s0 values
    s2 [t, i] = if (i <= 0 || i >= 1023, // stage 2
        0.0,
        u [-1, 0] + r * (s1 [0, -1] + s1 [0, 1]) - (2 * r) * s1 [0, 0]
    );    // t old u and new s1 values
    s3 [t, i] = if (i <= 0 || i >= 1023, // stage 3
        0.0,
        (r / 2) * (s2 [0, -1] + s2 [0, 1]) + r * s2 [0, 0]
    );    // t old u and new s2 values
    u [t, i] = if (t < 0,                // new value
        1.0,
        (s0 [0, 0] + 2 * s1 [0, 0] + s2 [0, 0] + s3 [0, 0] - u [-1, 0]) / 3
    );    // t old u and new s0, s1, s2, s3 values
}
```

A parallel functional language for high performance finite difference stencil codes

stencil kernel:
- functional description:
- differential equations
- image processing

schedule kernel:
- distributed memory
- accelerator
- SIMD vectors

data layout
- derived

generate (hoefully)
- high performace kernel codes
- find good storage and execution order in space-time

```
program     := [repeat kernel]
kernel      := [kernelid] [ [list itervar+] ] { [repeat statement] }
statement   := [id] [opt [ [list itervar+] ]] = [expr] ;
expr        := [number]
             | [expr] [arithmeticop] [expr]
             | - [expr]
             | ( [expr] )
             | [functionid] ( [list argexpr+] )
             | [id] [opt [ [list integer+] ]]
             | [bds0 | bds1] [opt [ [itervar] ]]
boolexpr    := [expr] [compareop] [expr]
             | [boolexpr] [logicop] [boolexpr]
             | ! [boolexpr]
             | ( [boolexpr] )
             | [functionid] ( [list argexpr+] )
argexpr     := [expr]
             | [boolexpr]
arithmeticop := * | / | + | -
logicop      := || | && | ^^
compareop    := == | != | < | > | <= | >=
```