Workshop on Aspects of Scientific Computing

Finite Difference Stencils in HPC Applications

Gerhard Zumbusch
Institut für Angewandte Mathematik

Friedrich-Schiller-Universität Jena

since 1558

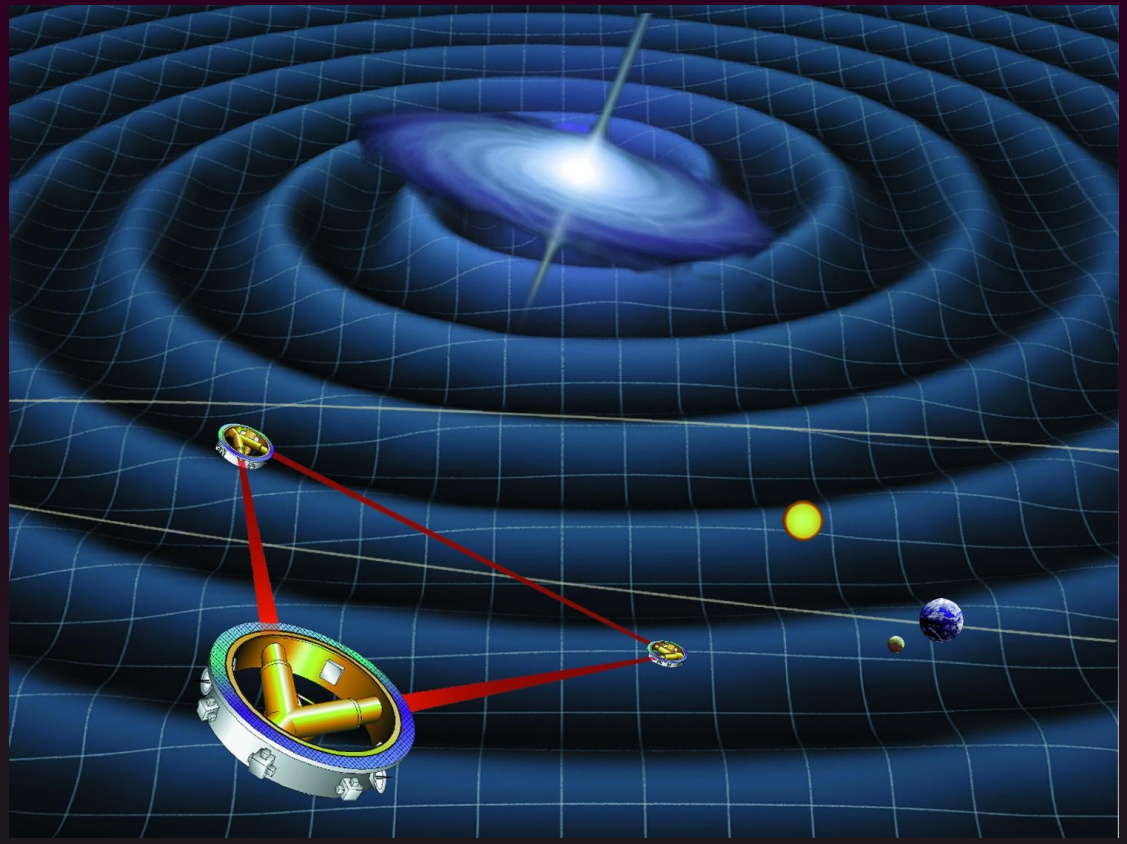
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray



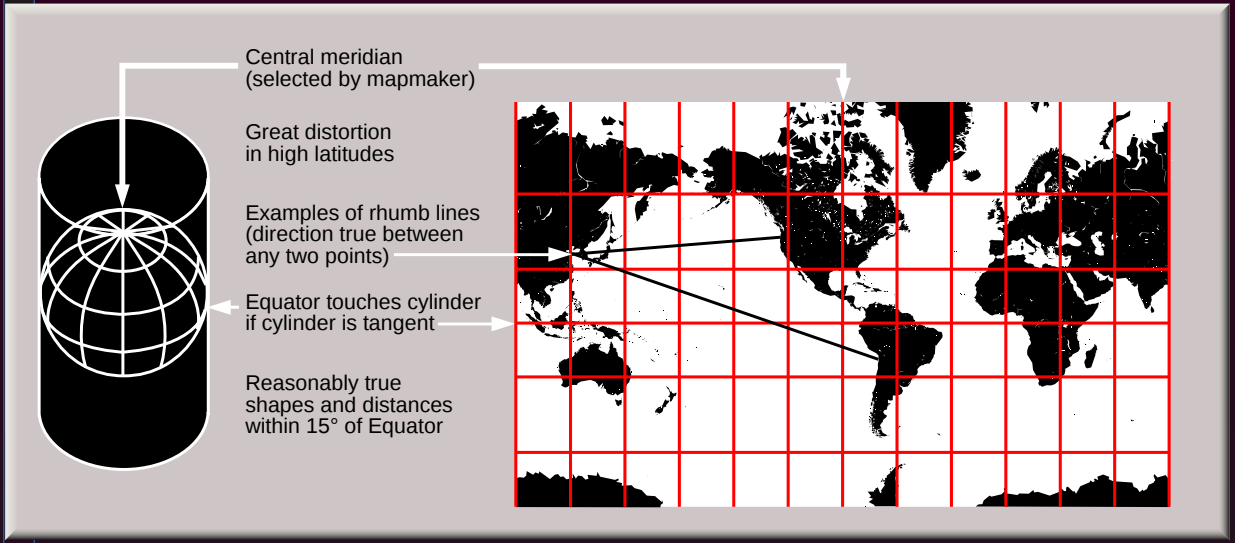
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray



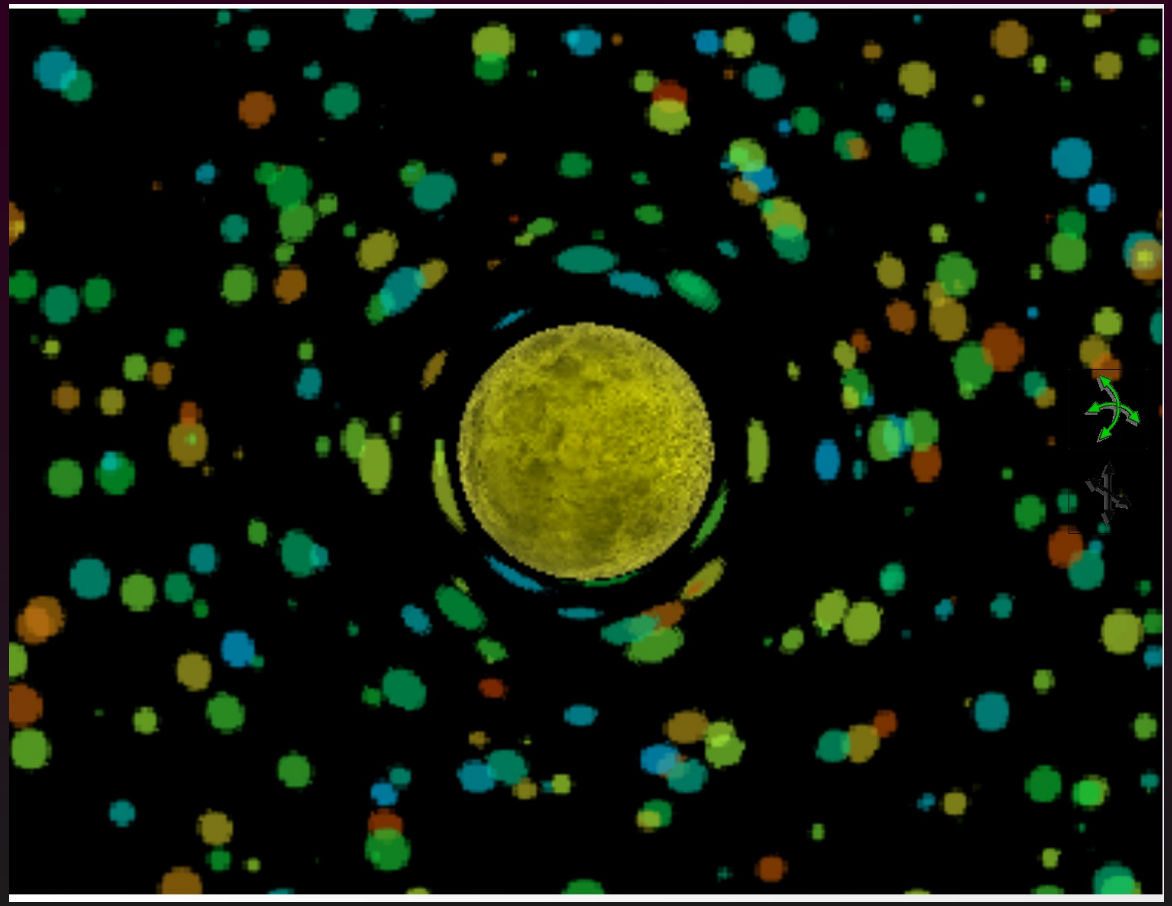
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray



GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

Position and velocity (x, v) in 4-space with respect to an intrinsic time

$$\begin{aligned} x^i(\tau)' &= v^i(\tau) \\ v^i(\tau)' &= \sum_{jk} \Gamma^i_{jk}(x(\tau)) v^j(t) v^k(\tau) \end{aligned}$$

with Christoffel symbol
$\Gamma^i_{jk} = \frac{1}{2} \sum_l g^{il} (d_j g_{kl} + d_k g_{jl} - d_l g_{jk})$,
g^{ij} inverse metric to g_{ij} and the light ray condition

$$\sum_{jk} g_{jk}(x(\tau)) v^j(\tau) v^k(\tau) = 0$$

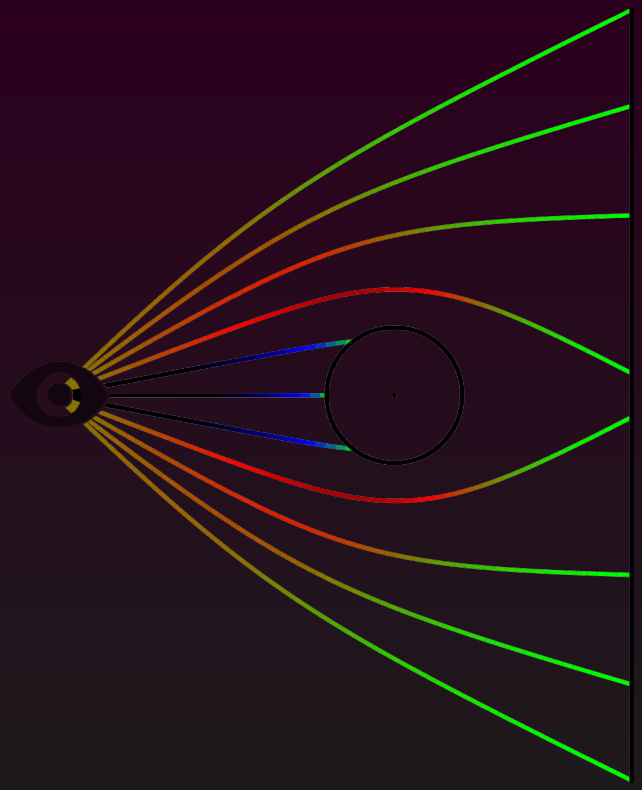
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray



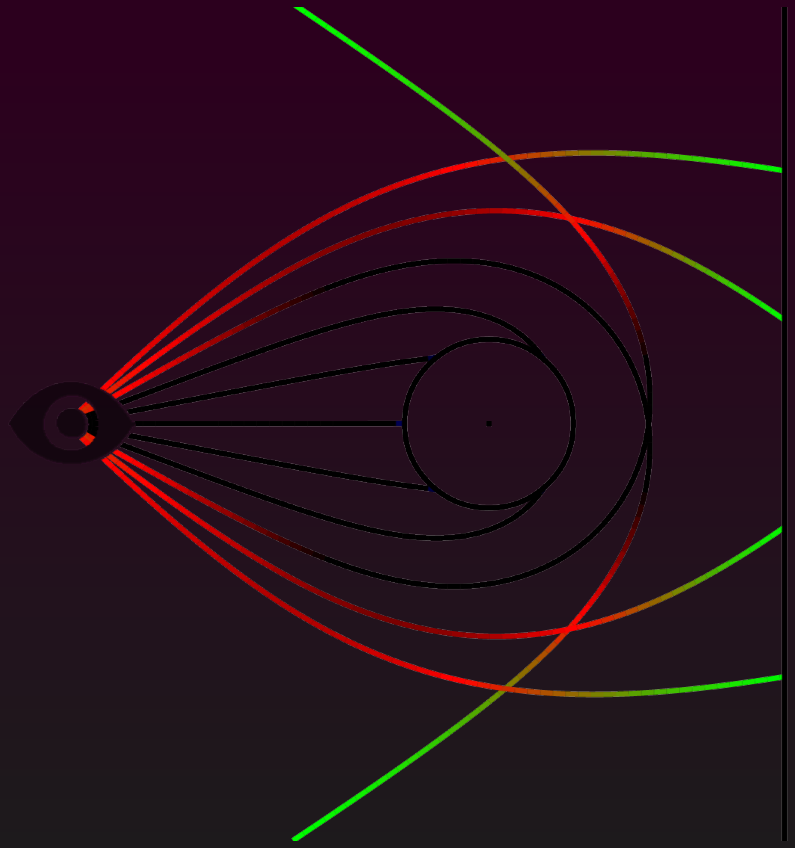
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray



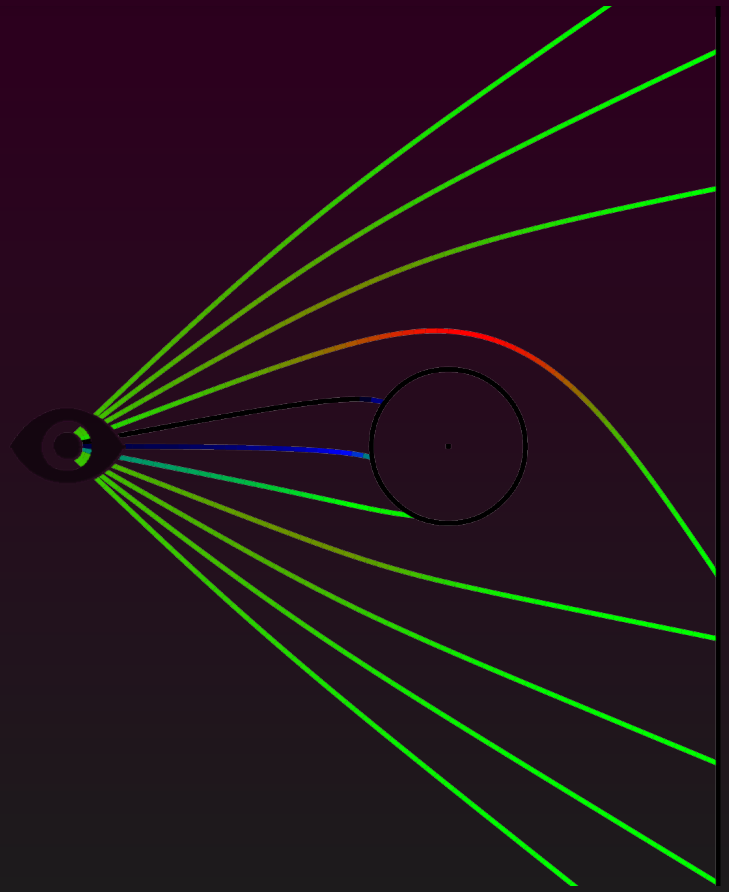
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray



GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

Euler scheme, piece-wise linear solution, straight rays:

$$
\begin{aligned}
x(\tau_{k+1}) &= x(\tau_k) + \Delta t v(\tau_k) \\
v(\tau_{k+1}) &= v(\tau_k) + \Delta t \sum_{jk} \Gamma^i_{jk}(x(\tau_k)) \\
&\quad \cdot v^j(\tau_k) v^k(\tau_k)
\end{aligned}
$$

GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

Avoid computation or storage of derivative terms $d_j g_{kl}$, use numeric differentiation:

$$d_j g_{kl} \approx \frac{1}{h} \left(g_{kl}(x + h e_j) - g_{kl}(x) \right)$$

GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

Avoid inverse g^{ij}, solve linear equation system with $g_{ij} \in R^{4\times 4\ \mathrm{sym}}$ instead: Symmetric (Cholesky) factorization $L \cdot D \cdot L^t$

$$L = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix}$$

GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

```
for (int j=0; j<4; j++) {
  float s = a[j][j];
  for (int k=0; k<j; k++) {
    float t = a[j][k];
    s -= a[k][k] * (t * t);
  }
  a[j][j] = s;
  for (int i=j+1; i<4; i++) {
    float s = a[i][j];
    for (int k=0; k<j; k++) {
      s -= a[k][k] * a[i][k] * a[j][k];
    }
    a[i][j] = s / a[j][j];
  }
}
```

GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

```
a21 = (a21) / a11;
a31 = (a31) / a11;
a41 = (a41) / a11;
a22 = a22 - a11 * (a21 * a21);
a32 = (a32 - a11 * (a31 * a21)) / a22;
a42 = (a42 - a11 * (a41 * a21)) / a22;
a33 = a33 - a11 * (a31 * a31) - a22 * (a32 * a32);
a43 = (a43 - a11 * (a41 * a31) - a22 * (a42 * a32)) / a33;
a44 = a44 - a11 * (a41 * a41) - a22 * (a42 * a42) - a33 * (a43 * a43);
float b1 = x[0];
float b2 = x[1];
float b3 = x[2];
float b4 = x[3];
b2 = b2 - b1 * a21;
b3 = b3 - b1 * a31 - b2 * a32;
b4 = b4 - b1 * a41 - b2 * a42 - b3 * a43;
b1 = b1 / a11;
b2 = b2 / a22;
b3 = b3 / a33;
b4 = b4 / a44;
b3 = b3 - b4 * a43;
b2 = b2 - b3 * a32 - b4 * a42;
b1 = b1 - b2 * a21 - b3 * a31 - b4 * a41;
x[0] = b1;
x[1] = b2;
x[2] = b3;
x[3] = b4;
```

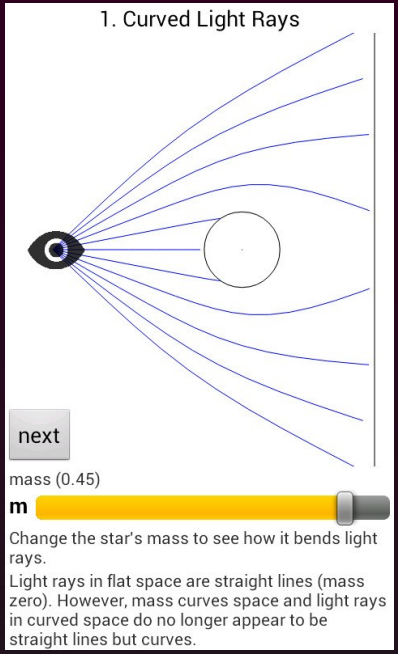
GR raytracer

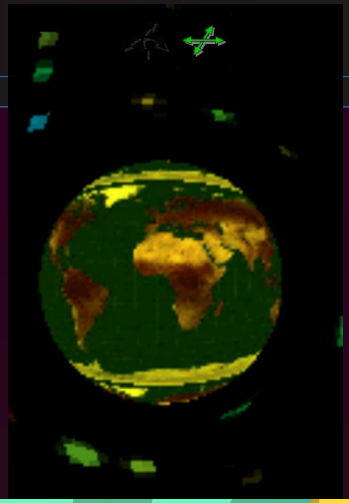
compute one geodesic per pixel
numeric representation of a metric

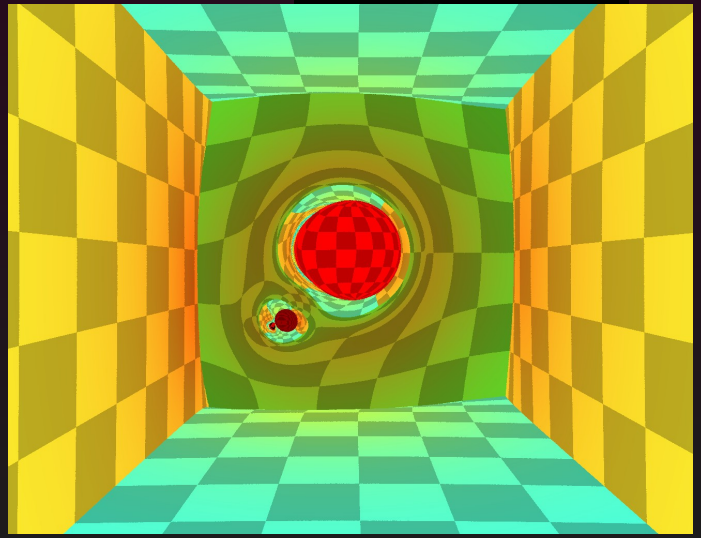
solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray







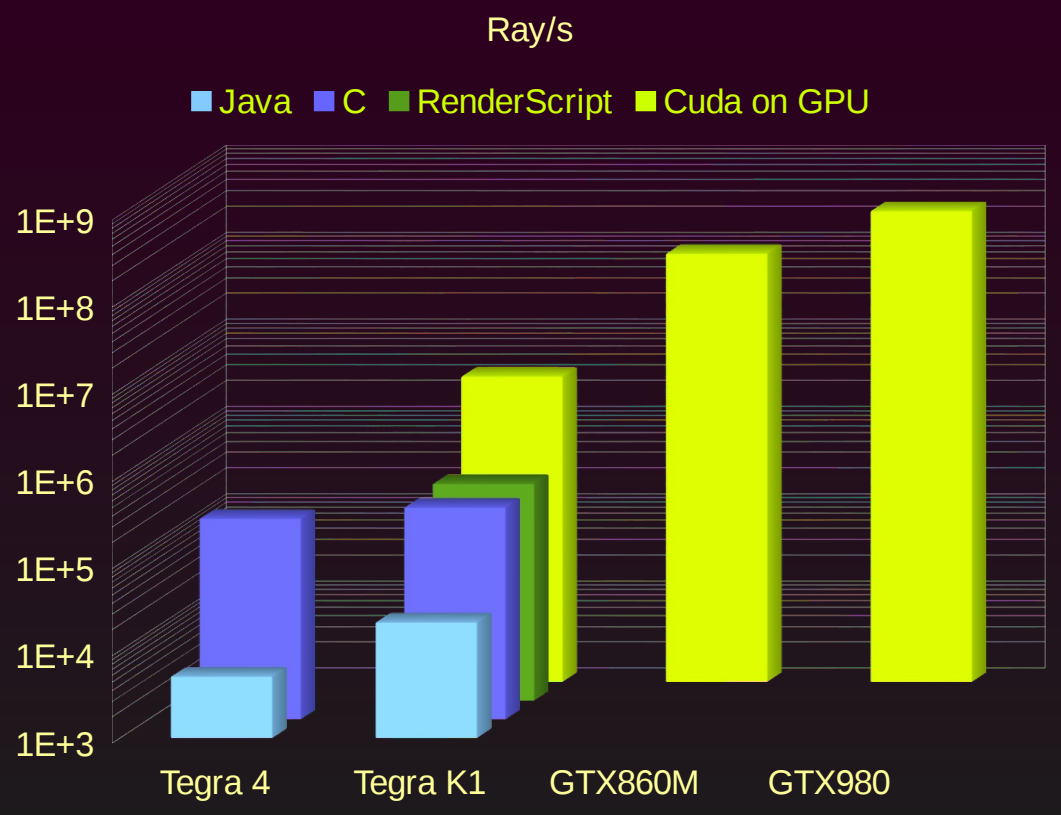
GR raytracer

compute one geodesic per pixel
numeric representation of a metric

solve geodesic equation numerically
- low precision ODE solver
- numerical derivatives
- inverse metric by LL^t factorization
- with loop unrolling
- embarrassingly parallel

CPU:
Android
Google Play “GRRay”

GPU: Nvidia Cuda+OpenGL
Linux/Microsoft Windows
github.com/zumbusch/gr-ray

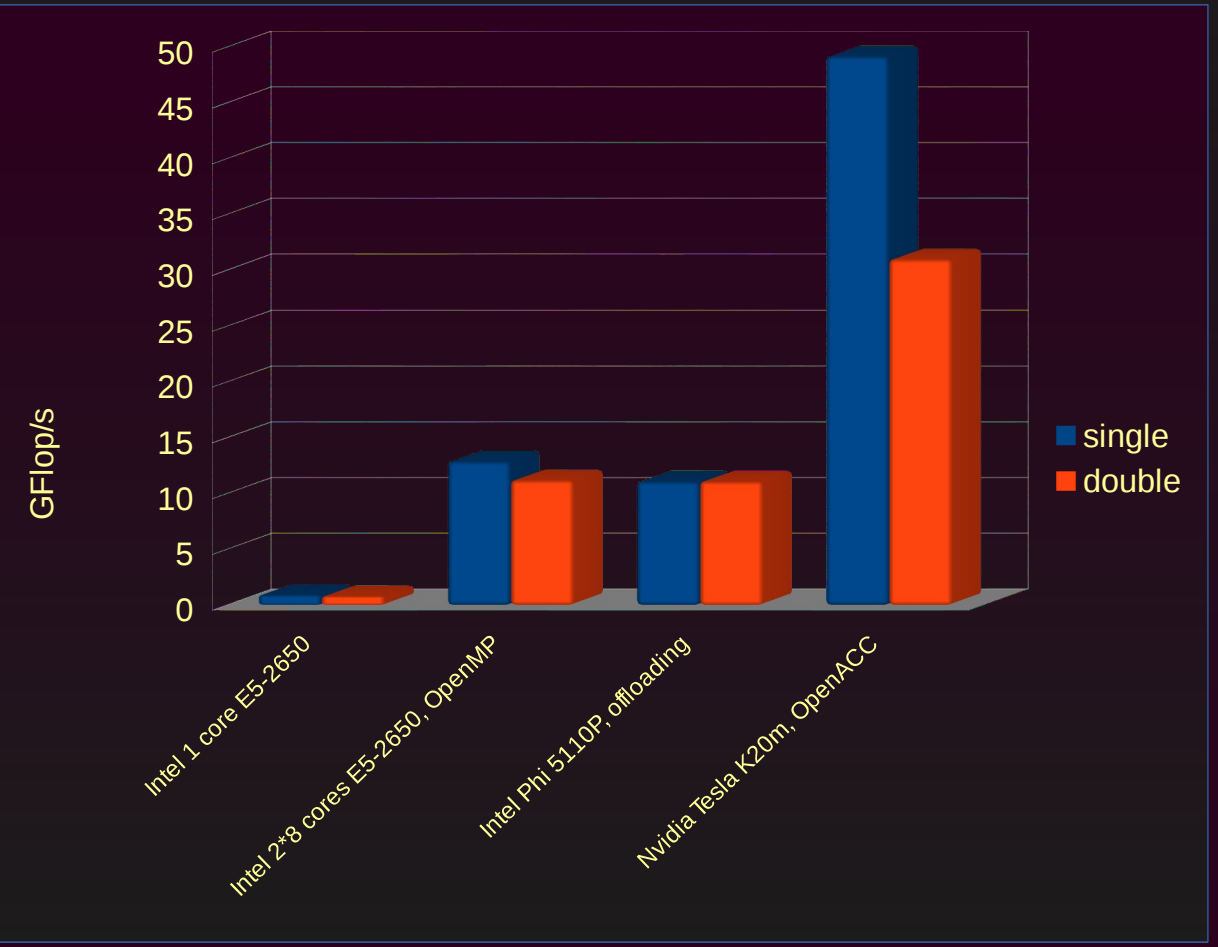


Results: standard/naive/textbook loop

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

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

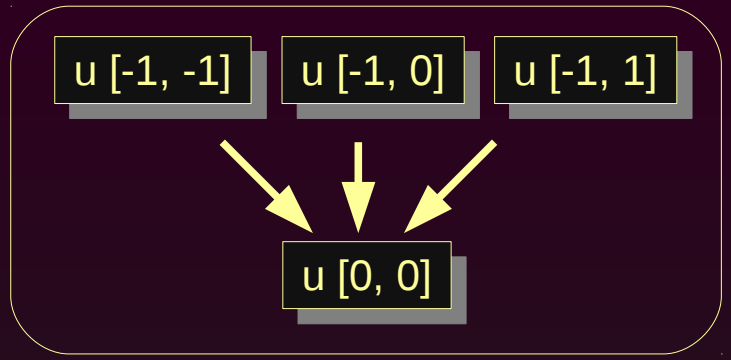
AMD >=Bulldozer, AVX, 1FMA/cycle

Intel Haswell, AVX, 2FMA/cycle

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

scalar loop, memory bandwith limited

```
// 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

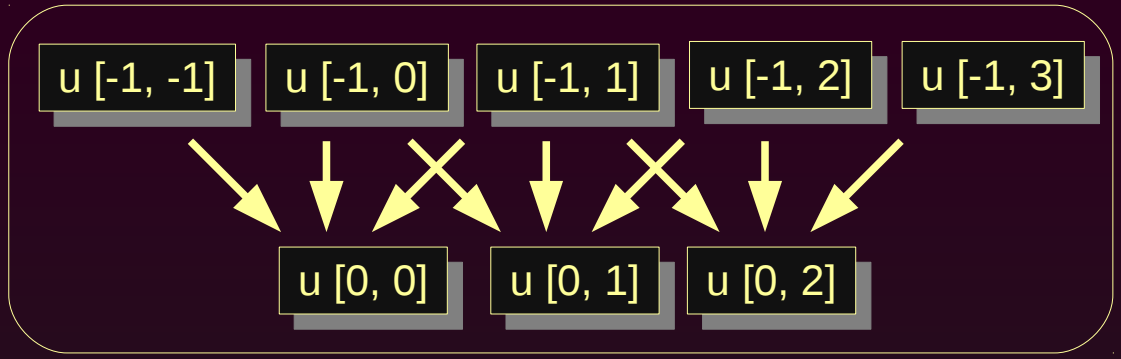
2 add, 2 mult

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
  }
```



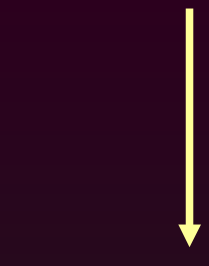
N+2 load

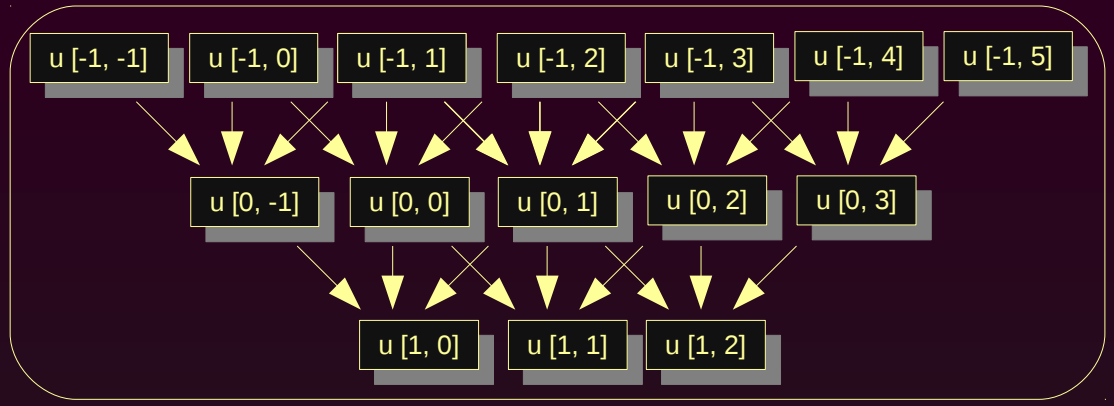
2N add, 2N mult

N store

→ 2 flop/mem op

Space-time, register/cache limited





N+2T load

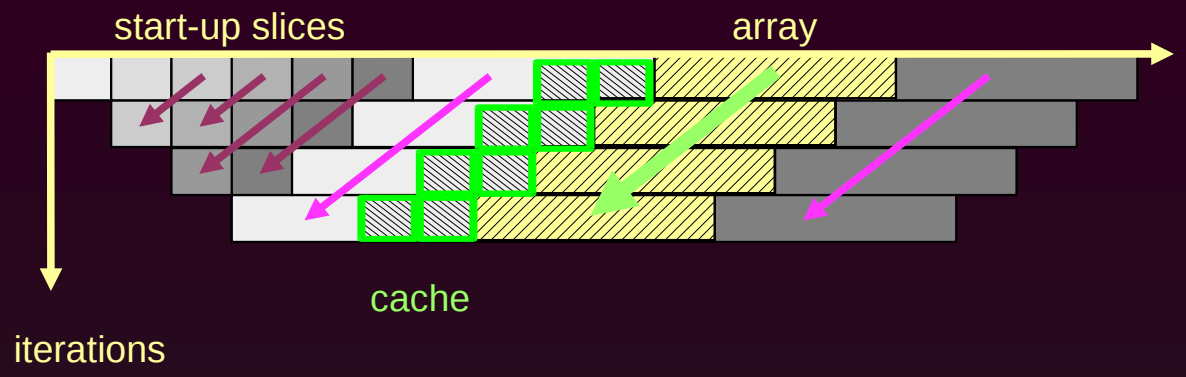
$2NT+2+T^2$ add, $2NT+2+T^2$ mult

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];
}
```



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

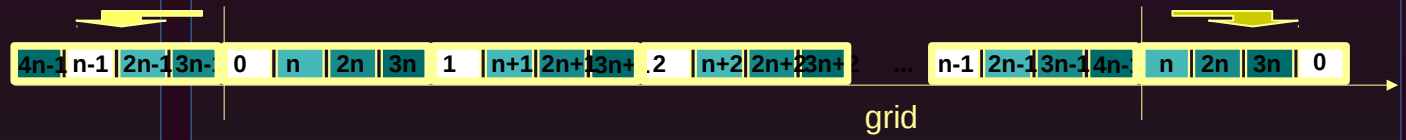
vectorization strategies

```
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]) );
```

```
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;
}
```

unaligned memory access

interleaved memory layout



Compiler Target: Vector Instruction Sets

current CPU/ GPU architectures are parallel vector processors

- multi-core CPU
- SMP CPUs

CPU vector units: SSE, AVX, AltiVec, Neon, …

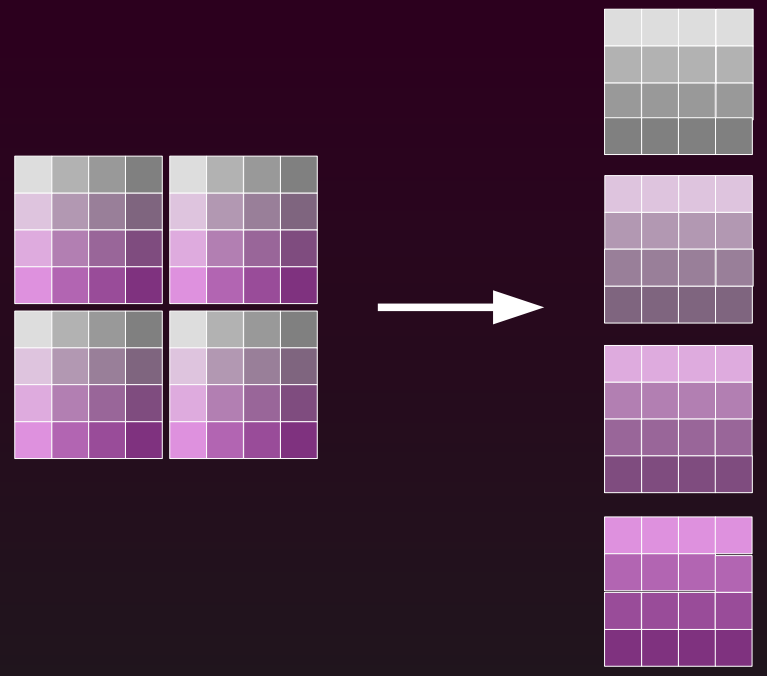
Nvidia Cuda:
- thread block % 32: vectors of length 32
- thread block / 32: multi-threading on one multi-processor
- grid of blocks: multi-threading on all multi-processors of a GPU

OpenCL:
- work group of local work items: vectors + multi-threading on one processor
- work groups on compute units: multi-threading on processors

scalar	scalar arithmetic, 1 float or 1 double	
sse	x86 SSE1: 128 bit vector, 4 floats	-msse
	SSE2: 4 floats or 2 doubles	-msse2
	SSE4.1: more advanced ops	-msse4
altivec	PowerPC AltiVec: 128 bit vector, 4 floats or 2 doubles	-maltivec
spu	Cell BE SPU: 128 bit vector, 4 floats (or 2 doubles)	spu-g++
neon	ARM Neon: 128 bit vector, 4 floats	-mfpu=neon
avx	x86 AVX: 256 bit vector, 8 floats or 4 doubles	-mavx
phi	new Larabee instruction set: 512 bit vector	
	Intel Phi: 16 floats or 8 doubles	-mmic
	AVX-512	
cuda	Nvidia GPU: synchronous warp, 32 floats/ doubles	nvcc
	compute >=3.0: warp shuffle	nvcc -arch=sm_30
opencl	OpenCL: vector, on GPUs 32 or 64 floats/ doubles	-lOpenCL

Optimization: Data Interleaving for Vectors & parallel Processors

```
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;
}
```



grid
4 sub-domains

memory
vectors of 4

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

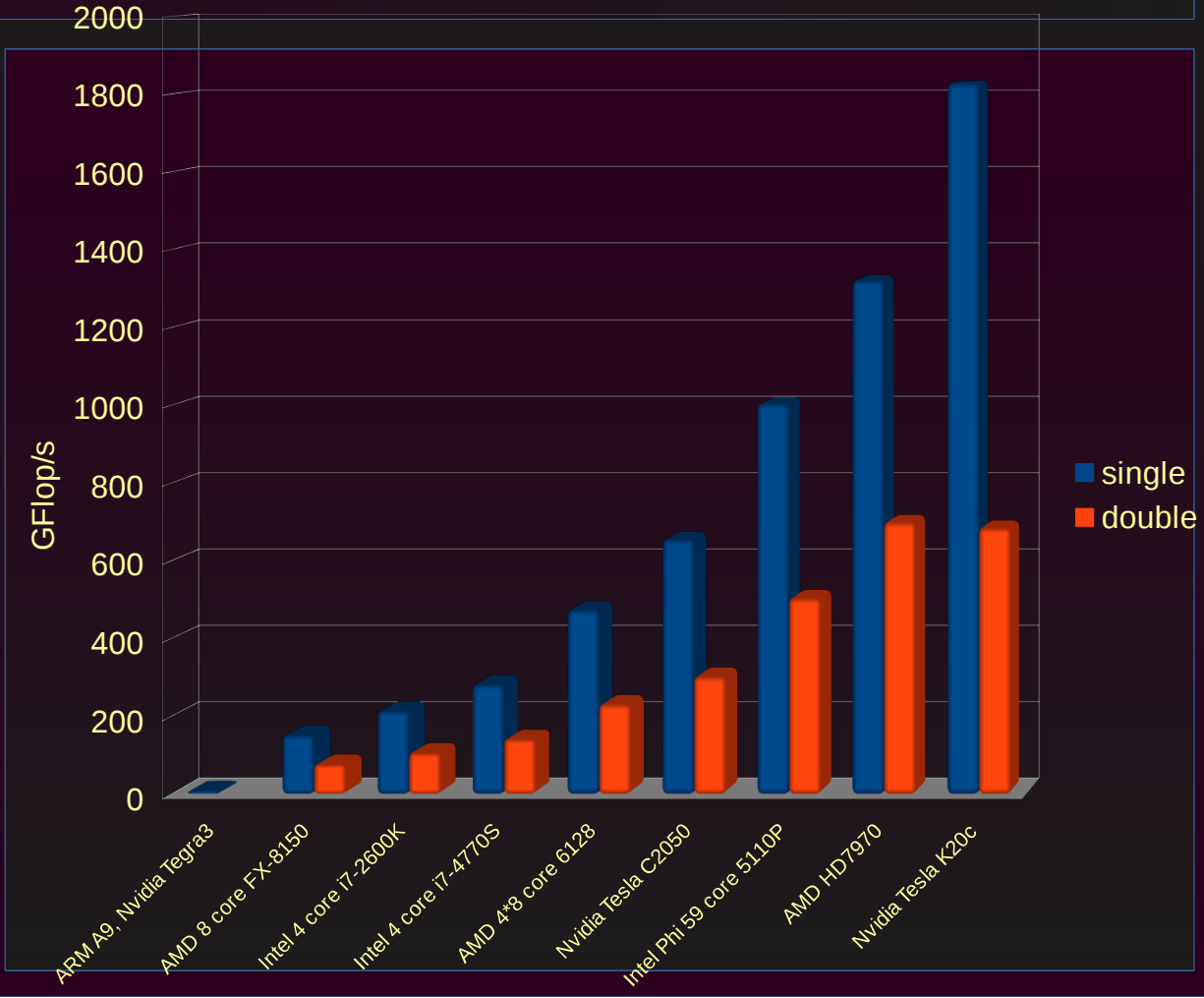
flop/s float	flop/s double	Processor
2.80368e+11	1.40064e+11	4 core Intel Haswell
2.17196e+11	1.08554e+11	4 core Intel Sandy Bridge
1.5238e+11	7.51819e+10	4 module Bulldozer
2.39837e+10	1.18196e+10	2 core Intel Netburst
5.48074e+09		4 core ARM v7
1.43877e+12	5.26778e+11	Nvidia Kepler, K20
7.23759e+11	1.03476e+11	Nvidia Fermi, GTX590
1.01081e+12	5.00217e+11	Intel MIC
9.84205e+10		Intel HD4000

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

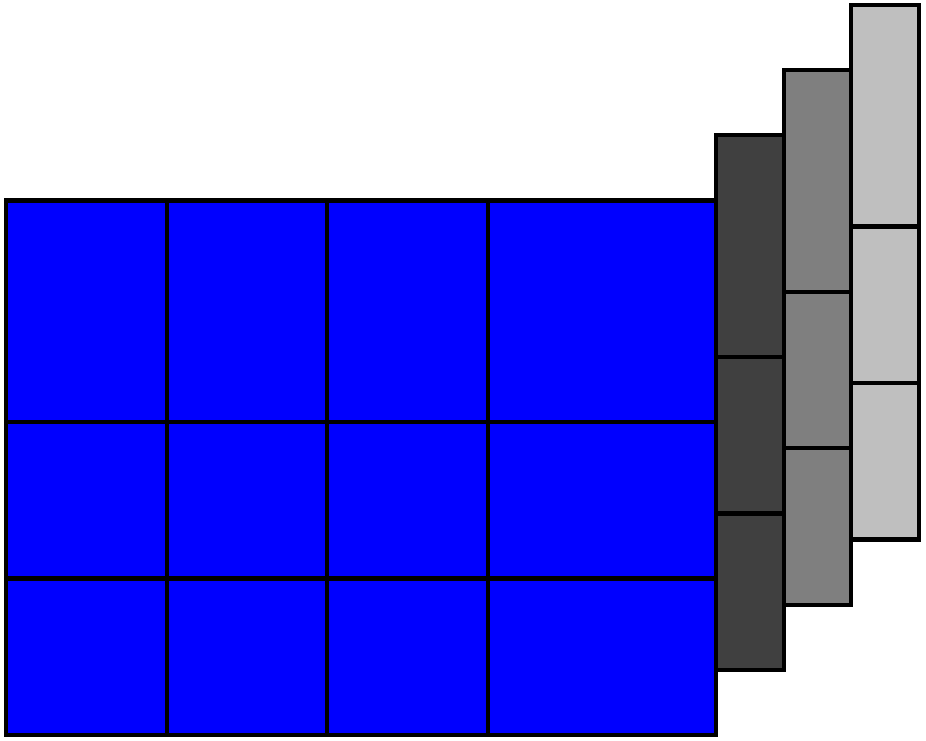


2D

re-order and aggregate operations
re-order memory layout
transform loops to allow compiler optimize the rest

for parallelism, vectorization, memory cache

CPU and GPU benchmark codes
PThreads, Cuda, OpenCL
github.com/zumbusch/bench_fd1



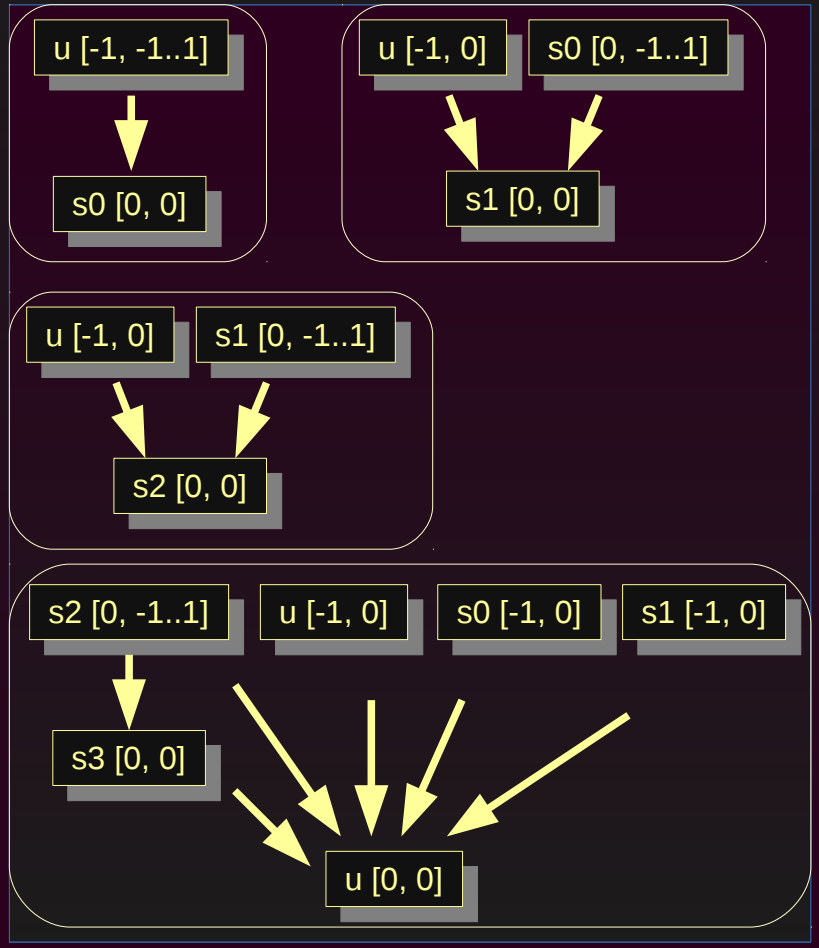
max performance in G Flop/s

processor		1D single	1D double	2D single	2D double	3D single	3D double
Nvidia Kepler	GK110, Tesla K20c	1829	685	355	179	211	123
Nvidia Kepler	GK104, GTX 680	1638	98.3	309	75.0	219	60.4
Nvidia Fermi	GF100, Tesla C2050	655	302	241	116	196	98.6
Nvidia Fermi	1*GF110, GTX 590	794	114	304	78.5	236	69.4
AMD Southern Islands	GCN, HD 7970	1322	699	421	473	211	137
AMD Northern Islands	VLIW4, HD 6990	560	403	96.4	79.1	62.7	52.5
Intel Sandy Bridge	one core i7-2600	59.6	29.8	40.8	20.4	21.2	10.4
AMD Bulldozer	one module FX-8150	44.2	22.1	22.7	11.3	13.0	7.2

time slicing

sliding window

4 stage Runge-Kutta scheme

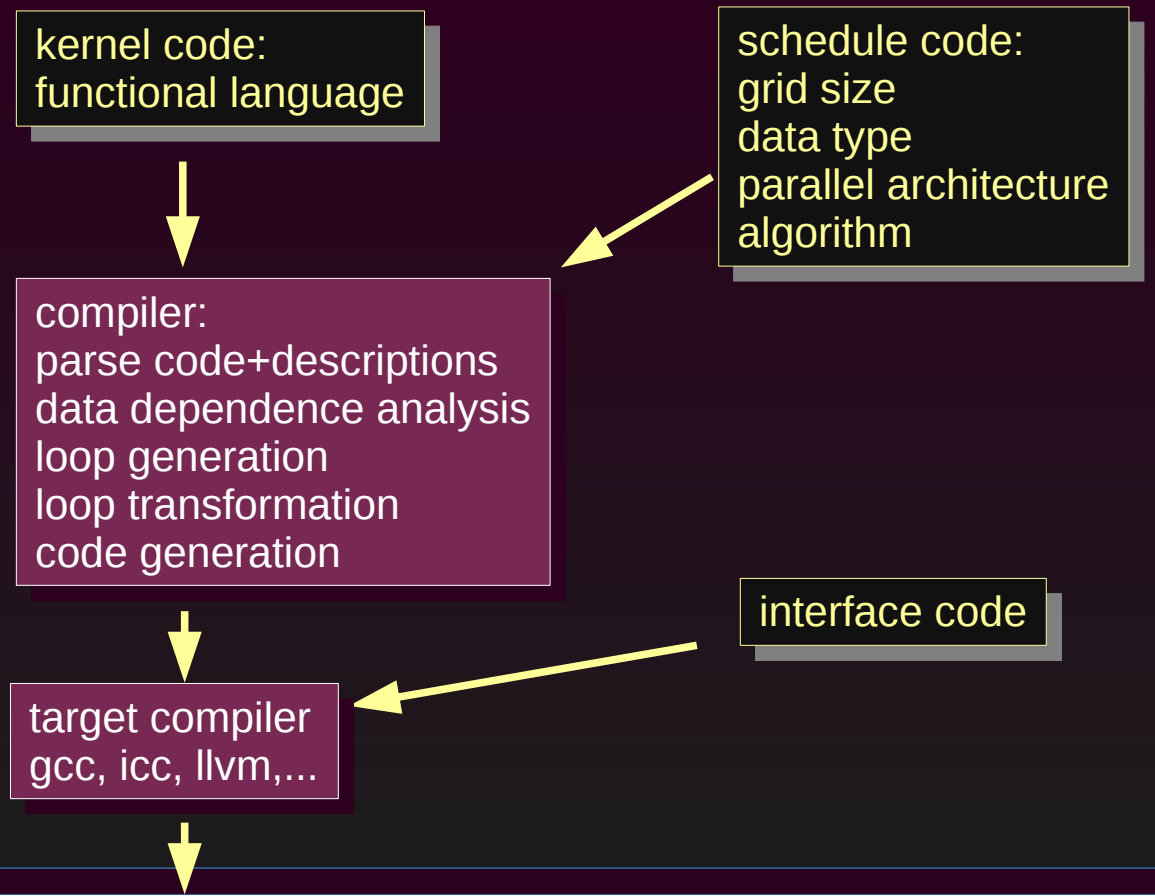



```
RK4 [t, i] {
    r = .25;
    s0 [t, i] = if (i <= 0 || i >= 1023, // stage 0
        0.0,
        (r / 2) * (u [-1, -1] + u [-1, 1]) + (1 - r) * u [-1, 0]
    );    // t old u values
    s1 [t, i] = if (i <= 0 || i >= 1023, // stage 1
        0.0,
        u [-1, 0] + (r / 2) * (s0 [0, -1] + s0 [0,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

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;
// 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;
```