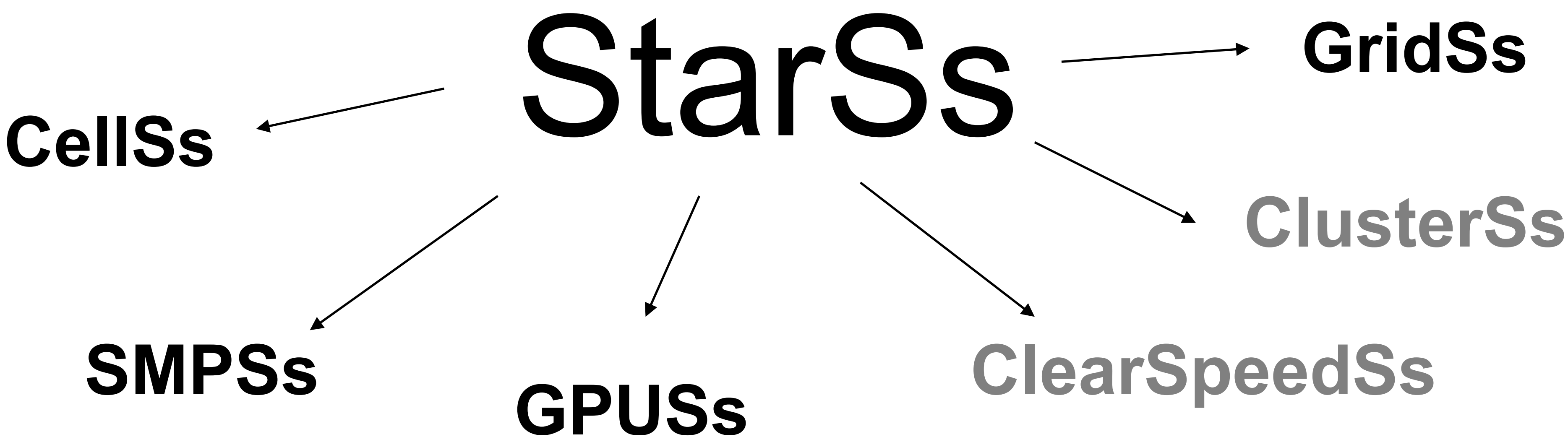
Barcelona
Supercomputing
Center
Centro Nacional de Superc

Exploiting Multicore Processors and GPUs with OpenMP and OpenCL

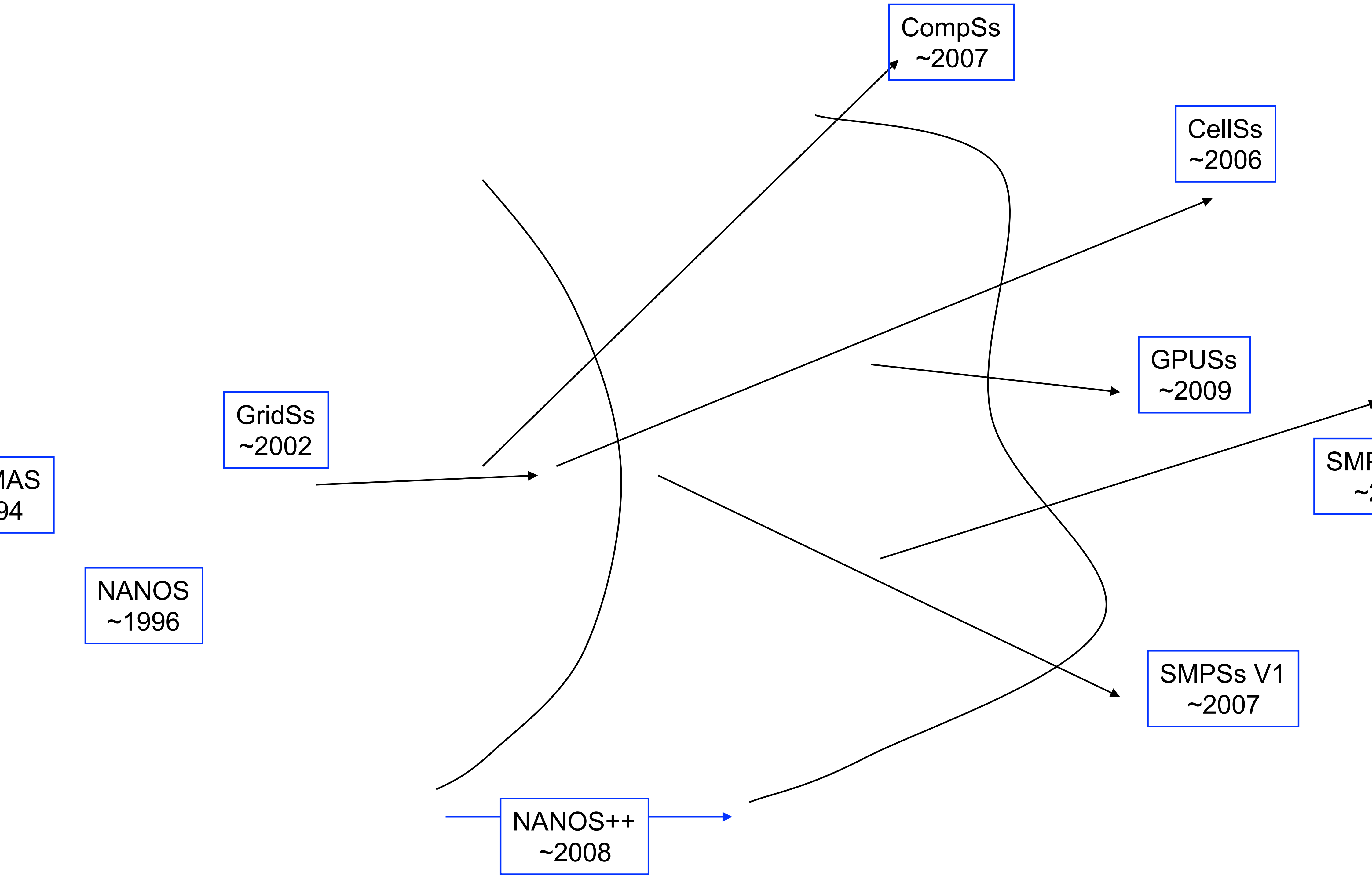
Rosa M. Badia
BSC

Clusters, Clouds, and Grids for Scientific Computing
CCGSC 2010

- StarSs, OpenMP and OpenMPT
- Programming models for accelerators
- Handling accelerators and heterogeneity
- Examples and results
- Conclusions



- StarSs
  - A “node” level programming model
  - C/Fortran + directives
  - Nicely integrates in hybrid MPI/StarSs
  - Natural support for heterogeneity
- Programmability
  - Incremental parallelization/restructure
  - Abstract/separate algorithmic issues from resources
  - Disciplined programming
- Portability
  - “Same” source code runs on “any” machine
    - Optimized task implementations will result in better performance.
  - “Single source” for maintained version of a application
- Performance
  - Asynchronous (data-flow) execution and loc awareness
  - Intelligent Runtime: specific for each type of target platform.
    - Automatically extracts and exploits parallelisr
    - Matches computations to resources


CompSs
~2007
CellSs
~2006
GPUSs
~2009
GridSs
~2002
MAS
94
NANOS
~1996
SMPSs V1
~2007
NANOS++
~2008

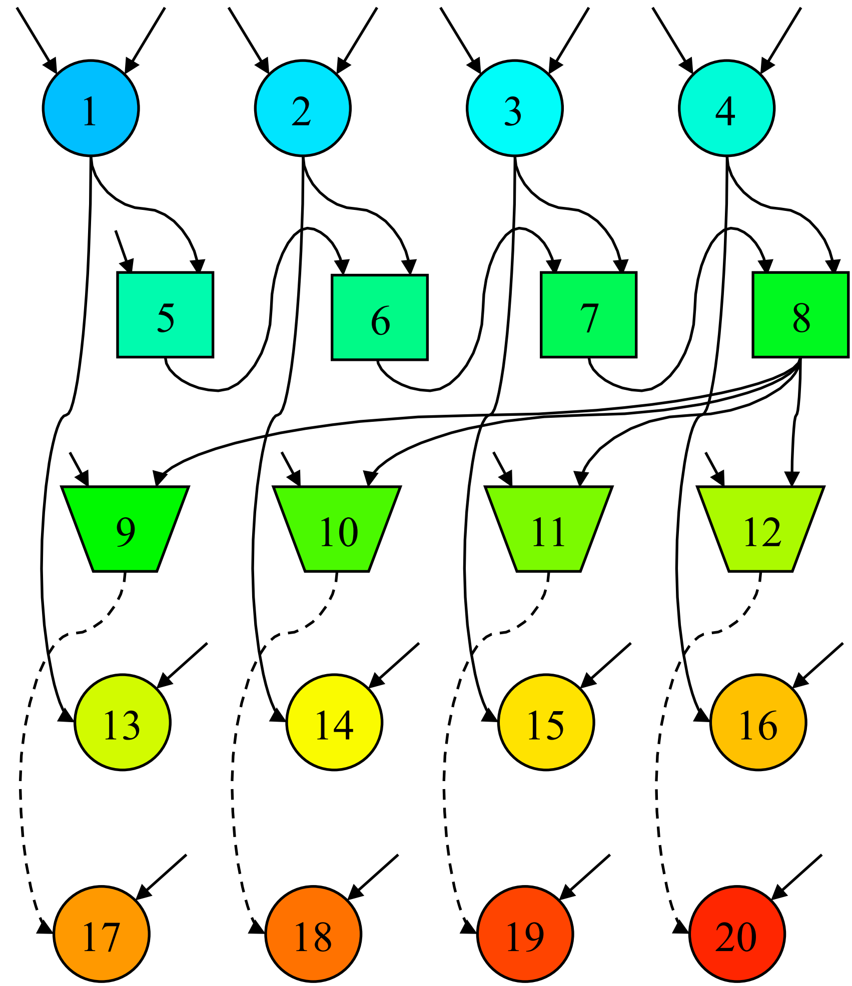
```
add3 (float A[BS], float B[BS],
      float C[BS]);

cale_add (float sum, float A[BS],
          float B[BS]);

ccum (float A[BS], float *sum);
```

```
=0; i<N; i+=BS)                    // C=A+B
d3 ( &A[i], &B[i], &C[i]);

=0; i<N; i+=BS)                   // sum(C[i])
um (&C[i], &sum);

=0; i<N; i+=BS)                   // B=sum*A
le_add (sum, &E[i], &B[i]);

=0; i<N; i+=BS)                   // A=C+D
d3 (&C[i], &D[i], &A[i]);

=0; i<N; i+=BS)                   // E=G+F
d3 (&G[i], &F[i], &E[i]);
```

```
a css task input(A, B) output(C)
add3 (float A[BS], float B[BS],
       float C[BS]);
a css task input(sum, A) inout(B)
cale_add (float sum, float A[BS],
           float B[BS]);
a css task input(A) inout(sum)
ccum (float A[BS], float *sum);
```

```
=0; i<N; i+=BS)                 // C=A+B
d3 ( &A[i], &B[i], &C[i]);

=0; i<N; i+=BS)                // sum(C[i])
um (&C[i], &sum);

=0; i<N; i+=BS)                // B=sum*A
le_add (sum, &E[i], &B[i]);

=0; i<N; i+=BS)                // A=C+D
d3 (&C[i], &D[i], &A[i]);

=0; i<N; i+=BS)                // E=G+F
d3 (&G[i], &F[i], &E[i]);
```

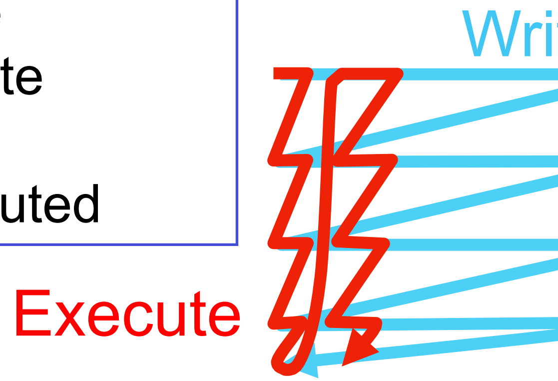
Compute dependences @ task instantiati

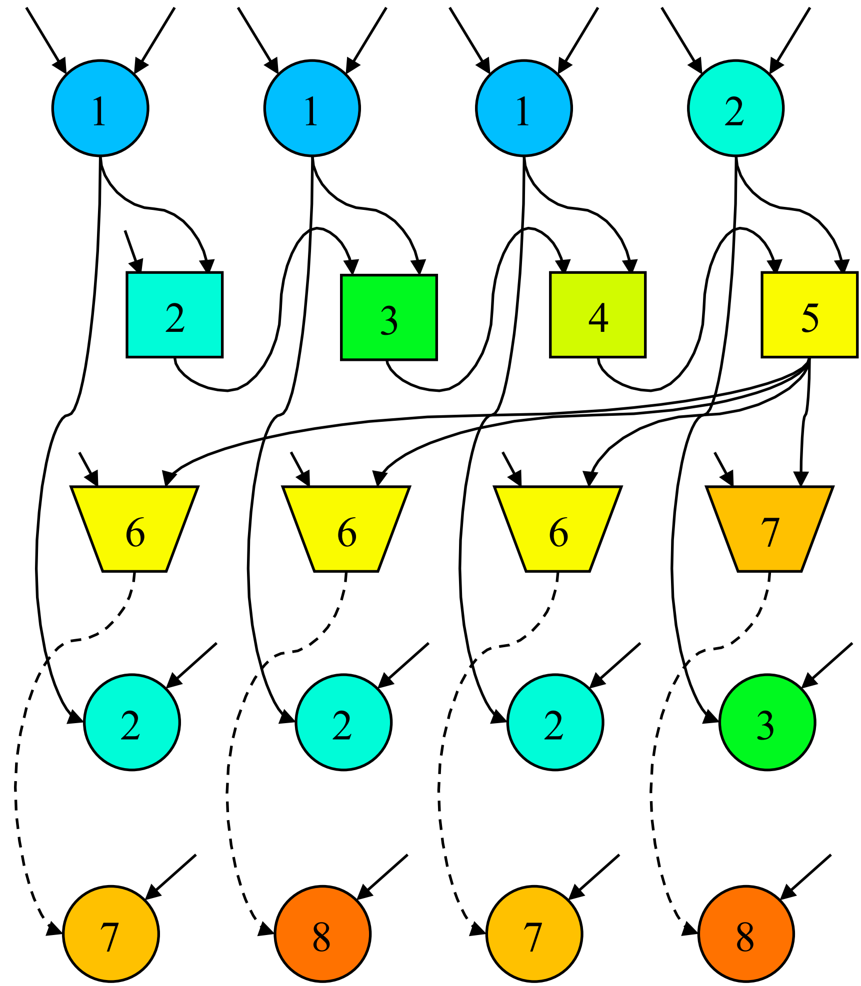


```
a css task input(A, B) output(C)
add3 (float A[BS], float B[BS],
        float C[BS]);
a css task input(sum, A) inout(B)
cale_add (float sum, float A[BS],
            float B[BS]);
a css task input(A) inout(sum)
ccum (float A[BS], float *sum);
```

```
=0; i<N; i+=BS)                     // C=A+B
d3 ( &A[i], &B[i], &C[i]);

=0; i<N; i+=BS)                    // sum(C[i])
um (&C[i], &sum);

=0; i<N; i+=BS)                    // B=sum*A
le_add (sum, &E[i], &B[i]);

=0; i<N; i+=BS)                    // A=C+D
d3 (&C[i], &D[i], &A[i]);

=0; i<N; i+=BS)                    // E=G+F
d3 (&G[i], &F[i], &E[i]);
```

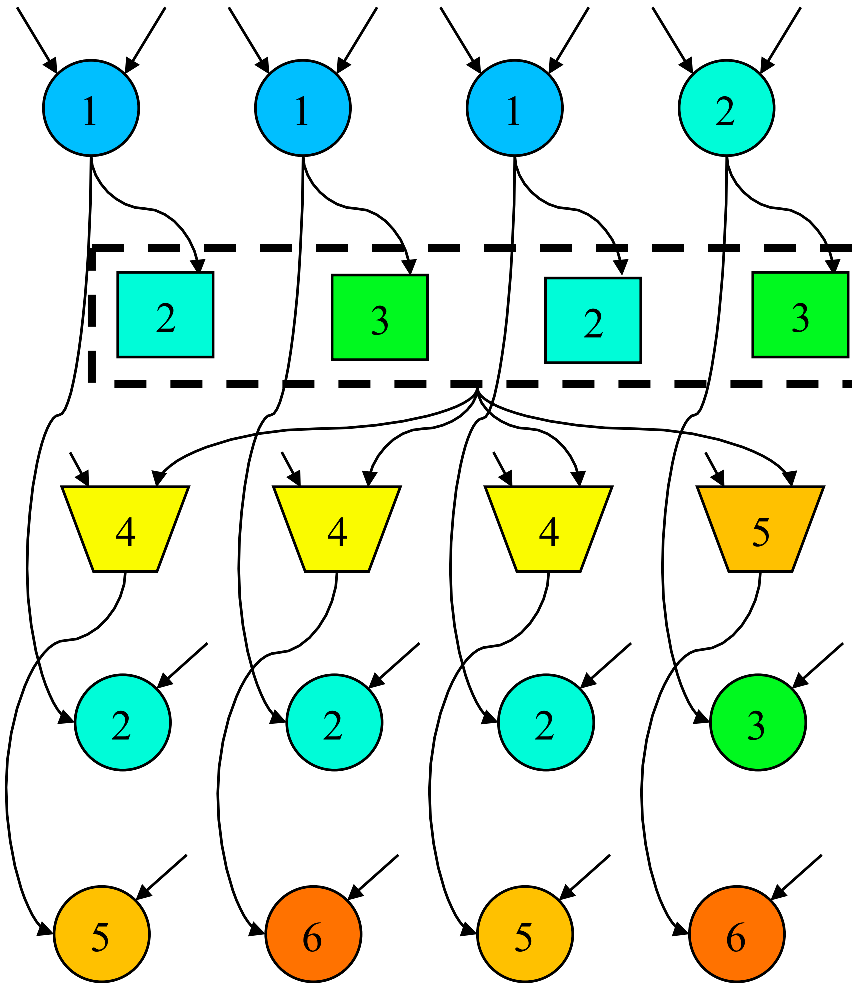
Decouple
how we write
from
how it is executed





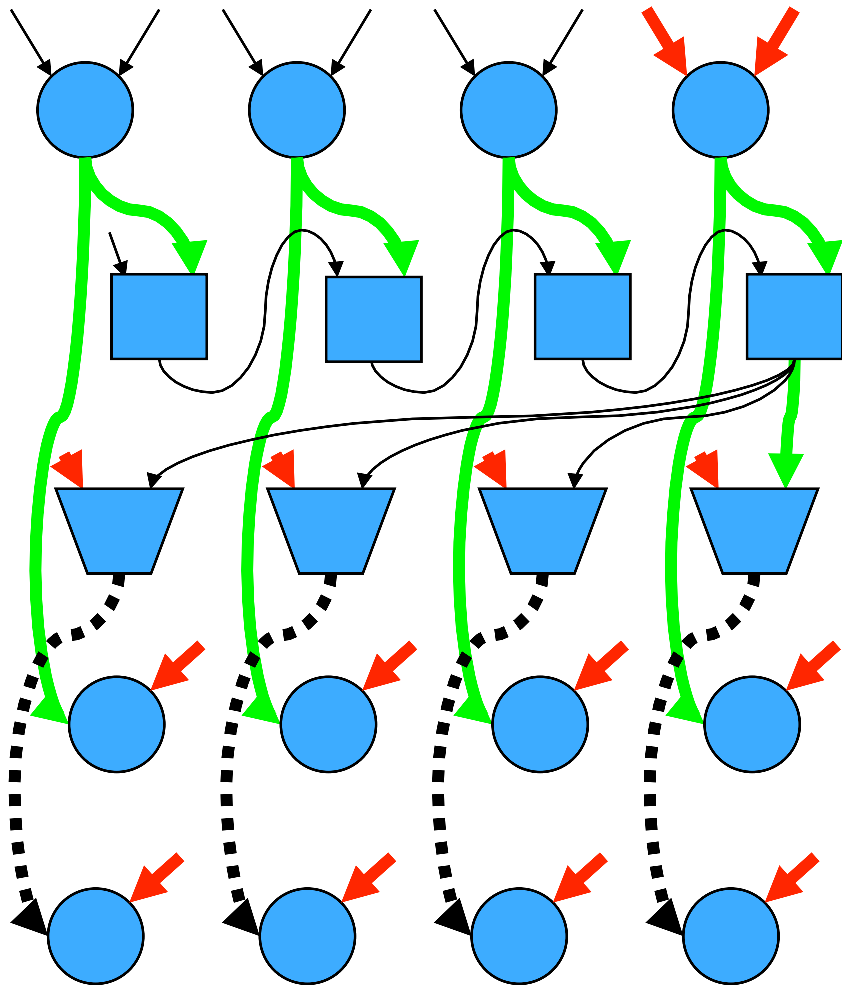
```
a css task input(A, B) output(C)
add3 (float A[BS], float B[BS],
        float C[BS]);
a css task input(sum, A) inout(B)
cale_add (float sum, float A[BS],
            float B[BS]);
a css task input(A) inout(sum) reduction(sum)
ccum (float A[BS], float *sum);
```

```
=0; i<N; i+=BS)                    // C=A+B
d3 ( &A[i], &B[i], &C[i]);

=0; i<N; i+=BS)                    // sum(C[i])
um (&C[i], &sum);

=0; i<N; i+=BS)                    // B=sum*A
le_add (sum, &E[i], &B[i]);

=0; i<N; i+=BS)                    // A=C+D
d3 (&C[i], &D[i], &A[i]);

=0; i<N; i+=BS)                    // E=G+F
d3 (&G[i], &F[i], &E[i]);
```



- **Flat global address space seen by programmer**
- Flexibility to dynamically traverse dataflow graph “optimizing”
  - Concurrency. Critical path
  - Memory access: data transfers performed by run time

- Opportunities for
  - Prefetch
  - Reuse
  - Eliminate antidependences (rename)
  - Replication management
    - Coherency/consistency handled by the runtime



```
pragma css task  [input ( parameters ) ] \
                 [output ( parameters ) ] \
                 [inout ( parameters )] \
                    [target device( [cell, smp, cuda] ) ] \
                    [implements ( task_name ) ] \
                    [reduction ( parameters ) ] \
                 [ highpriority ]

pragma css wait on ( data_address )
pragma css barrier
pragma css mutex lock ( variable )
pragma css mutex unlock( variable )
```

```
parameters: parameter [ , parameter ]*
parameter: variable_name  {[dimension]
```

MP

t parallelism.

oin

provide some more flexibility

ality information. Global Addressing

g

Sta

Implicit paralle

“atomic” ta

Explicit data access informa

Local addres

Single work gener

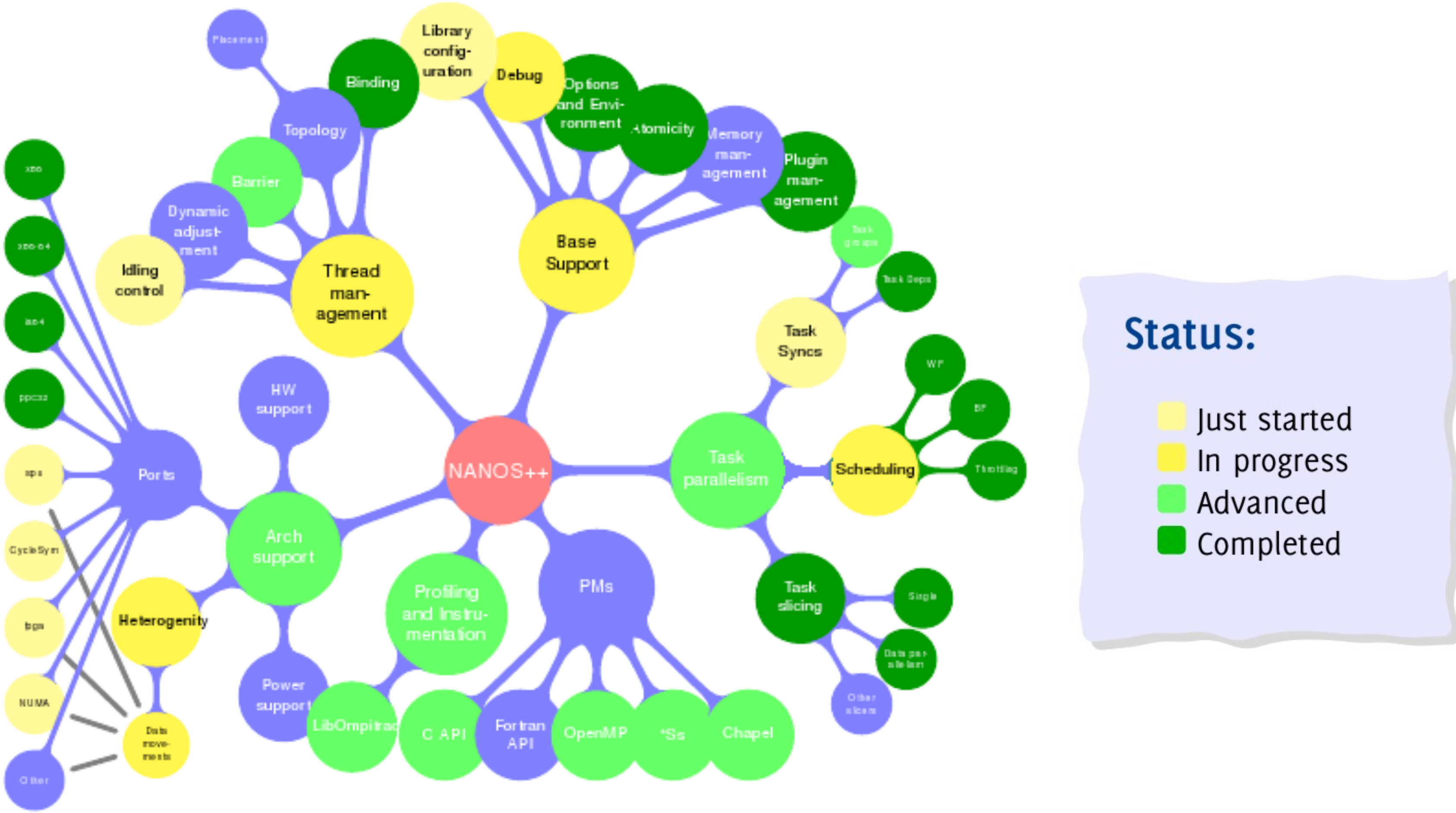
OpenMPT

, et all, “A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures” IWOMP 2009 & IJPP

et al, “Extending the OpenMP Tasking Model to Allow Dependent Tasks” IWOMP 2008, LNCS & IJPP

- Compiler (mercurium) and runtime (NANOS++)
- Support/integrate: OpenMP, StarSs, Chapel,…

Mercurium



cit kernel execution submission

plicit kernel synchronization

```
/*Enqueue a kernel run call */
    status = clEnqueueNDRangeKernel(
                commandQueue,
                kernel,
                2,
                NULL,
                globalThreads,
                localThreads,
                0,
                NULL,
                &events[0]);

    if(!sampleCommon->checkVal(
            status,
            CL_SUCCESS,
            "clEnqueueNDRangeKernel failed."))
        return SDK_FAILURE;


    /* wait for the kernel call to finish execution */
    status = clWaitForEvents(1, &events[0]);
    if(!sampleCommon->checkVal(
            status,
            CL_SUCCESS,
            "clWaitForEvents failed."))
        return SDK_FAILURE;
```

```
cute the kernel over the entire range of our 1d input data set
 using the maximum number of work group items for this device

obal = count;
r = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL,
;
 (err)

  printf("Error: Failed to execute kernel!\n");
  return EXIT_FAILURE;


Wait for the command commands to get serviced before reading back results
```

- Host program deleting:
  - Comments
  - Some return code checks
  - Checks of block sizes supported by device

```
eofday(&timev1, NULL);

ROR(clGetPlatformIDs(16, platforms, &num_platforms));
ROR(clGetPlatformInfo(platforms[0], CL_PLATFORM_PROFILE
e_buffer,&param_value_size_ret));
ROR(clGetDeviceIDs(platforms[0], device_type, 1, &devic
ROR(clGetDeviceInfo(device_list,CL_DEVICE_PREFERRED_VEC
dth,NULL));
ROR(clGetDeviceInfo(device_list,CL_DEVICE_MAX_WORK_ITEM
ROR(clGetDeviceInfo(device_list,CL_DEVICE_GLOBAL_MEM_SI
ROR(clGetDeviceInfo(device_list,CL_DEVICE_LOCAL_MEM_SIZ
ROR(clGetDeviceInfo(device_list,CL_DEVICE_MAX_MEM_ALLOC
ROR(clGetDeviceInfo(device_list, CL_DEVICE_EXTENSIONS,
ROR(clGetDeviceInfo(device_list,CL_DEVICE_MAX_COMPUTE_U

= (ntasks == -1) ? max_compute_units : ntasks;
ize  = (cl_ulong) (max_alloc_size / 8);
roups = array_size / (vector_width * local_work_group_si

= clCreateContext(0, (cl_uint) 1, &device_list, NULL, N
e = clCreateCommandQueue(context,device_list, CL_QUEUE_
array_size;
rray_size = (array_size < 16) ? 16 : array_size;
malign((void **) &rawbuf, 128, sizeof(cl_double) * 7 *
  = &rawbuf[0 * sizeof(cl_double) * malloc_array_size]
  = &rawbuf[1 * sizeof(cl_double) * malloc_array_size]
  = &rawbuf[2 * sizeof(cl_double) * malloc_array_size]
  = &rawbuf[3 * sizeof(cl_double) * malloc_array_size]
```

```
cpflag_fptr = (unsigned int *) cpflag; S0_fptr = (float *) S0; K_fptr = (float *)
r_fptr = (float *) r; sigma_fptr = (float *) sigma; T_fptr = (float *) T;
answer_fptr = (float *) answer;
int memsize = array_size * (double_flag ? sizeof(double) : sizeof(float));
memobjs[0] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, cpflag, &rc);
memobjs[1] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, S0, &rc);
memobjs[2] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, K, &rc);

memobjs[3] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, r, &rc);
memobjs[4] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, sigma, &rc);
memobjs[5] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, T, &rc);
memobjs[6] = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, memsize, answer, &rc);

bs_source = load_program (kernel_source_file, &rc);


program = clCreateProgramWithSource (context, 1, (const char**)(&bs_source), NULL

if (cod
-DRANGE_LO
if (cod

char nu

sprintf
strcat(
strcat(

sprintf
strcat(
strcat(

rc = cl

kernel

rc = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
rc = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
rc = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);
rc = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *) &memobjs[3]);
rc = clSetKernelArg(kernel, 4, sizeof(cl_mem), (void *) &memobjs[4]);
rc = clSetKernelArg(kernel, 5, sizeof(cl_mem), (void *) &memobjs[5]);
rc = clSetKernelArg(kernel, 6, sizeof(cl_mem), (void *) &memobjs[6]);


gettimeofday(&timev4, NULL);
```

```
int bsop_rangeLS(int double_flag, cl_ulong array_size, cl_ulong local_wor
{
   gettimeofday(&timev5, NULL);

   global_work_size[0] = (size_t) (n_workgroups*local_work_group_size);
   local_work_size[0] = (size_t) local_work_group_size;

   rc = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
              global_work_size, local_work_size, 0, NULL, &event[0]);

   if(rc != CL_SUCCESS) {
              fprintf(stderr, "Executing the kernel failed...rc=%d\n",rc);

   clWaitForEvents((cl_uint) 1, event);

   return 0;
}
```

```
bsop_ref  (unsigned int4 cpflag, float4
       float4 r, float4 sigma, float4 T
t4 d1, d2, Nd1, Nd2, expval, k1, n1, k2
   accum2, candidate_answer1, candidate
 flag1, flag2;

 log(S0/K) + (r + HALF * sigma*sigma)*T
= (sigma * sqrt(T));
al = exp(ZERO - r * T);
 d1 - sigma * sqrt(T);
1 = (d1 < ZERO);
2 = (d2 < ZERO);
 fabs(d1);
 fabs(d2);
 ONE / (ONE + NCDF * d1);
 ONE / (ONE + NCDF * d2);
m1 = A4 + A5 * k1;
m2 = A4 + A5 * k2;
m1 = k1 * accum1 + A3;
m2 = k2 * accum2 + A3;
m1 = k1 * accum1 + A2;
m2 = k2 * accum2 + A2;
m1 = k1 * accum1 + A1;
m2 = k2 * accum2 + A1;
m1 = k1 * accum1;
m2 = k2 * accum2;
 exp(ZERO - HALF * d1 * d1);
 exp(ZERO - HALF * d2 * d2);
= INV_ROOT2PI;
= INV_ROOT2PI;
idate_answer1 = ONE - n1 * accum1;
idate_answer2 = ONE - n2 * accum2;
= SELECT(candidate_answer1, (ONE - candida
= SELECT(candidate_answer2, (ONE - candida
 = S0 * Nd1 - K * expval * Nd2;
```

```
__kernel __attribute__((reqd_work_group_size(LWGSIZE, 1, 1))
void bsop_kernel (__global unsigned int4 *dm_cpflag,
                  __global float4 *dm_S0,
                  __global float4 *dm_K,
{
K
}
```

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void bsop_kernel (__global const FIXED *cpflag_dm,
                  __global const FLOAT *S0_dm,
                  __global const FLOAT *K_dm,
                  __global const FLOAT *r_dm,
                  __global const FLOAT *sigma_dm,
                  __global const FLOAT *T_dm,
                  __global       FLOAT *answer_dm,
                  int task_id,
                  __local        FIXED *lm_cpflag,
                  __local        FLOAT *lm_S0,
                  __local        FLOAT *lm_K,
                  __local        FLOAT *lm_r,
                  __local        FLOAT *lm_sigma,
                  __local        FLOAT *lm_T,
                  __local        FLOAT *answer,
                  int n, size_t stride) {
  int i, j;
  event_t event;
  stride >>= STRIDESHIFT;

  for (j = n*task_id; j < n*(task_id+1); j+=stride) {
    event = async_work_group_copy(lm_cpflag, (cpflag_dm +
 stride, (event_t) 0);
    event = async_work_group_copy(lm_S0,(S0_dm+j),stride,
         …
    wait_group_events(1, &event);
    for(i = 0; i < stride; i++)
      answer[i] = bsop_ref(lm_cpflag[i],lm_S0[i],lm_K[i],
r[i],lm_sigma[i], lm_T[i]);
```

```
op_test (int double_flag, cl_ulong array_size, cl_ulong local_work_group_size)

, bsize;

meofday(&timev5, NULL);

i=0; i<array_size; i+=local_work_group_size) {
ze= ((i+local_work_group_size)>array_size) ? array_size - i : local_work_group_size
p_ref_float ( bsize, &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &r_fptr[i],
              &sigma_fptr[i], &T_fptr[i],
              &answer_fptr[i]);

a css barrier
meofday(&timev6, NULL);
```

```
float bsop_reference_float (unsigned int cpflag, f
                            float K, float r, floa
                            float T) {
   float d1, d2, c, p, Nd1, Nd2, expval, answer;
   d1 = logf(S0/K) + (r + 0.5*sigma*sigma)*T;
   d1 /= (sigma * sqrt(T));
   expval = exp(-r * T);
   d2 = d1 - sigma * sqrt(T);
   Nd1 = Nf(d1); Nd2 = Nf(d2);
   c = S0 * Nd1 - K * expval * Nd2;
   p = K * expval * (1.0 - Nd2) - S0 * (1.0 - Nd1)
   answer = cpflag ? c : p;
   return answer;
}
```

```
s task input(size, cpflag_fptr[size], S0_fptr[size], K_fptr[size], r_fptr[size], \
             sigma_fptr[size], T_fptr[size]) \
       output (answer_fptr[size])
ref_float (cl_ulong size, unsigned int * cpflag_fptr, float * S0_fptr, float * K_fptr,
           float * r_fptr, float * sigma_fptr, float * T_fptr, float * answer_fptr)



 0; i < size; i++) {
_fptr[i] = bsop_reference_float (cpflag_fptr[i], S0_fptr[i],
```

```
t bsop_test (int double_flag, cl_ulong array_size, cl_ulong local_work_group_size)

signed long long esp;
t i;

ttimeofday(&timev5, NULL);

t ii;
 for (ii = 0; ii < array_size; ii+=local_work_group_size) {
ma omp task private (i)
     for (i=ii; (i<ii+local_work_group_size) && (i<array_size); i+=OCLN) {
        answer_fptr[i] = bsop_reference_float(cpflag_fptr[i], S0_fptr[i], K_fptr[i],
              r_fptr[i], sigma_fptr[i], T_fptr[i]);
     }
 }
ma omp taskwait

ttimeofday(&timev6, NULL);

turn 0;
```

```
float bsop_reference_float (unsigned int cpflag, flo
                            float K, float r, float
                            float T) {
   float d1, d2, c, p, Nd1, Nd2, expval, answer;
   d1 = logf(S0/K) + (r + 0.5*sigma*sigma)*T;
   d1 /= (sigma * sqrt(T));
   expval = exp(-r * T);
   d2 = d1 - sigma * sqrt(T);
   Nd1 = Nf(d1); Nd2 = Nf(d2);
   c = S0 * Nd1 - K * expval * Nd2;
   p = K * expval * (1.0 - Nd2) - S0 * (1.0 - Nd1);
   answer = cpflag ? c : p;
   return answer;
}
```

- Same main program, just another **implementation of the task**
- Using **OpenCL clean SIMD code** but not OpenCL kernel declarations
- No need for manual overlap between computation and transfers

```
float4 bsop_ref  (unsigned int4 cpflag, float4 S0, float4 K,
                 float4 r, float4 sigma, float4 T) {
   float4 d1, d2, Nd1, Nd2, expval, k1, n1, k2, n2, accum1,
          accum2, candidate_answer1, candidate_answer2, call,
   int4 flag1, flag2;

   d1 = log(S0/K) + (r + HALF * sigma*sigma)*T;
   d1 /= (sigma * sqrt(T));
   expval = exp(ZERO - r * T);
   d2 = d1 - sigma * sqrt(T);
   flag1 = (d1 < ZERO);
   flag2 = (d2 < ZERO);
   d1 = fabs(d1);
   d2 = fabs(d2);
   k1 = ONE / (ONE + NCDF * d1);
   k2 = ONE / (ONE + NCDF * d2);
   accum1 = A4 + A5 * k1;
   accum2 = A4 + A5 * k2;
```

```
gma css task input(size, cpflag_fptr[size], S0_fptr[size], K_fptr[size], r_fptr[size], \
                  sigma_fptr[size], T_fptr[size]) \
            output (answer_fptr[size])
osop_ref_float (cl_ulong size, unsigned int * cpflag_fptr, float * S0_fptr, float * K_fptr,
                float * r_fptr, float * sigma_fptr, float * T_fptr, float * answer_fptr)

pedef __attribute__((vector_size(16))) float float4;
pedef __attribute__((vector_size(16))) int   int4;
oat4 bsop_ref (int4, float4, float4, float4, float4, float4);

r (int i = 0; i < size; i+=4) {
*((float4 *) &answer_fptr[i]) = bsop_ref (*((int4 *) &cpflag_fptr[i]),  *((float4 *) &S0_fptr[i]),
                                         *((float4 *) &K_fptr[i]),      *((float4 *) &r_fptr[i]),
                                         *((float4 *) &sigma_fptr[i]), *((float4 *) &T_fptr[i]));
```

multiplication (host side)

```
atrixMulOnDevice(float* M, float* N, float* P, int Width)

 size = Width * Width * sizeof(float);
oat* Md, Nd, Pd;

 Allocate and Load M, N to device memory
daMalloc(&Md, size);
daMemcpy(Md, M, size, cudaMemcpyHostToDevice);

daMalloc(&Nd, size);
daMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

 Allocate P on the device
daMalloc(&Pd, size);


 Setup the execution configuration
m3 dimGrid(1, 1);
m3 dimBlock(Width, Width);


 Launch the device computation threads!
trixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
daMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

 Free device matrices
daFree(Md); cudaFree(Nd); cudaFree (Pd);
```

main program code

```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

// M * N on the device
    MatrixMulOnDevice(M, N, P);

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

Matrix multiplication (device side)

```
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matr
 P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < M.width; ++k)
    {
         float Melement = M.elements[ty * M.pitch + k];
         float Nelement = Nd.elements[k * N.pitch + tx];
         Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
```

program code

```
main( void ){
  ...
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        matmul_tile (A[i][k], B[k][j], C[i][j]);
  ...
}
```

Main program:

- No explicit data transfers or allocation
- No explicit execution configuration
- The same StarSs main program can be used

Task (devic

```
__global__ void matmul_cuda ( float * A, float * B, float * C, int wA, int
 wB ){
  int bx = blockIdx.x;   int by = blockIdx.y;
  int tx = threadIdx.x;  int ty = threadIdx.y;

  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd = aBegin + wA - 1;   int aStep  = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;       int bStep = BLOCK_SIZE * wB;
  float Csub = 0;

  for( int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep ){
    __shared__ float As[ BLOCK_SIZE ][ BLOCK_SIZE ];
    __shared__ float Bs[ BLOCK_SIZE ][ BLOCK_SIZE ];
    As[ ty ][ tx ] = A[ a+wA * ty + tx ];
    Bs[ ty ][ tx ] = B[ b+wB * ty + tx ];
    __syncthreads( );
    for( int k = 0;: k < BLOCK_SIZE; k++ )
      Csub += As[ ty ][ k ] * Bs[ k ][ tx ];
    __syncthreads( );
  }
}
```

(host side)

```
ma css task input(A[BS][BS], B[BS][BS]) inout( C[BS][BS] )
ma css target device (CUDA)
matmul_tile (float *A, float *B, float *C ){
matmul_cuda <<<dimGrid, dimBlock>>>(A, B, C, BS, BS);
```

main program code

```
void matmul ( int m, int l, int n, int mDIM , in
 int nDIM ,
float ** A, float ** B, float ** C)
{
for (i = 0;i < mDIM ; i++) {
   for (j = 0; j < nDIM ; j++) {
      for (k = 0; k < lDIM ; k++) {
         matmul_block (A[i* lDIM +k],B[k* nDIM +
                C[i* nDIM +j ]);
         }
      }
   }
# pragma omp taskwait
}
```

x multiplication: multiple kernel implementations

```
t int NB = 512;
agma omp target device (smp , cell ) copy_deps
agma omp task inout ([ NB*NB] C) input ([ NB*NB] A, [NB*NB] B)
 matmul_block ( float * A, float * B, float * C)

penCL kernel


agma omp target device ( cuda ) copy_deps implement (matmul_block)
 matmul_block_gpu ( float * A, float * B, float * C)

UDA kernel
```

Dependences: not all arguments in directionality claus

Heterogeneous devices

Differen
implementat

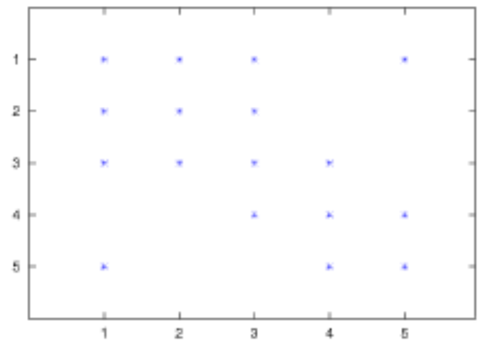
Separation dependences/trans

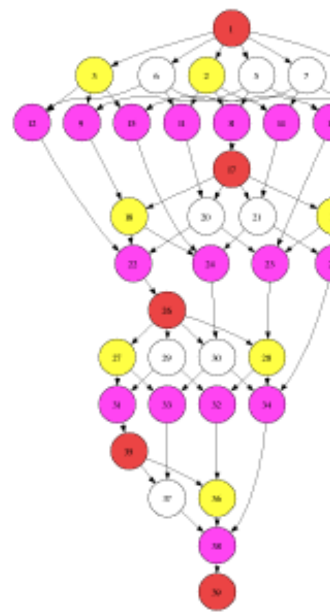
```
gma omp task inout(C[BS][BS])
d matmul( float *A, float *B, float *C) {
/ original sequential code


gma omp target device(cuda) implements(matmul)
        copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
d matmul_cuda ( float *A, float *B, float *C) {
/ optimized kernel for cuda


gma omp target device(cell) implements(matmul)
        copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
d matmul_spe ( float *A, float *B, float *C);

at *A[NB][NB], *B[NB][NB], *C[NB][NB];

 main( void ){
 (int i = 0; i < NB; i++)
for (int j = 0; j < NB; j++)
   for (int k = 0; k < NB; k++)
      matmul (A[i][k], B[k][j], C[i][j]);
```

```
factorization A = LU, overwriting A with the triangular factors */
u_getrf( float *A );

angular system solve B := B * inv(A), with A upper triangular */
u_trsm_right( float *A, float *B );

angular system solve B := inv(A) * B, with A unit lower triangular */
u_trsm_left( float *A, float *B );

trix multiplication C = C - A * B */
u_gemm( float *A, float *B, float *C );

arse_LU() {
int k = 0; k < NB; k++ ) {
agma omp task inout( A[k][k][0:BS-1][0:BS-1] )
getrf( A[k][k] );
 ( int i = k+1; i < NB; i++ )
if ( A[i][k] != NULL )
   #pragma omp task input( A[k][k][0:BS-1][0:BS-1] )\
                    inout( A[i][k][0:BS-1][0:BS-1] )
   lu_trsm_right( A[k][k], A[i][k] );

 ( int j = k+1; j < NB; j++ ) {
if ( A[k][j] != NULL )
   #pragma omp task input( A[k][k][0:BS-1][0:BS-1] )\
                    inout( A[k][j][0:BS-1][0:BS-1])
   lu_trsm_left( A[k][k], A[k][j] );
   for ( int i = k+1; i < NB; i++ )
       if ( A[i][k] != NULL ) {
           if ( A[i][j] == NULL)
              A[i][j] = allocate_clean_block();
           #pragma omp task input( A[i][k][0:BS-1][0:BS-1],\
                                   A[k][j][0:BS-1][0:BS-1] )\
                            inout( A[i][j][0:BS-1][0:BS-1] )
           lu_gemm( A[i][k], A[k][j], A[i][j]);
       }
```

Inline directives:

- saves manual outlining !!!
- Tasks have no name
  - → not multiple implementations





```
Cholesky factorization A = LL^T, overwriting the lower triangle of A with L */
gma omp task inout( A[0:BS-1][0:BS-1] )
l chol_potrf( float *A );

Triangular system solve B = B * inv(A)^T, with A lower triangular */
gma omp target device( cuda ) copy_in ( A[0:BS-1][0:BS-1], B[0:BS-1][0:BS-1] )\
                              copy_out( B[0:BS-1][0:BS-1] )
gma omp task input( A[0:BS-1][0:BS-1] ) inout( B[0:BS-1][0:BS-1] )
l chol_trsm_right( float *A, float *B );

Matrix multiplication C = C - A * B^T */
gma omp target device( cuda ) copy_in ( A[0:BS-1][0:BS-1], B[0:BS-1][0:BS-1],\
                                        C[0:BS-1][0:BS-1] )\
                              copy_out( C[0:BS-1][0:BS-1] )
gma omp task input( A[0:BS-1][0:BS-1], B[0:BS-1][0:BS-1] )\
             inout( C[0:BS-1][0:BS-1] )
l chol_gemm( float *A, float *B, float *C );

Symmetric rank-BS update C = C - A * A^T */
gma omp target device( cuda ) copy_in ( A[0:BS-1][0:BS-1], C[0:BS-1][0:BS-1] )\
                              copy_out( C[0:BS-1][0:BS-1] )
gma omp task input( A[0:BS-1][0:BS-1] ) inout( C[0:BS-1][0:BS-1] )
l chol_syrk( float *A, float *C );

 Cholesky() {
 i, j, k;

 ( k = 0; k < NB; k++ ) {
chol_potrf( A[k][k] );
for ( i = k+1; i < NB; i++ )
   chol_trsm_right( A[k][k], A[i][k] );

for ( i = k+1; i < NB; i++ ) {
   for ( j = k+1; j < i; i++ )
       chol_gemm( A[i][k], A[j][k], A[i][j]);
   chol_syrk( A[i][k], A[i][i]);
```

Annotated function declaration
ALL instances become tasks

```
 a[N];

agma omp task output(a[0:N/2-1])
a(a); //Task A: generate the bottom half of the array

agma omp task output(a[N/2:N-1])
b(a); //Task B: generate the upper half of the array

agma omp task input(a[0:N-1])
c(a); //Task C: use the array a
```

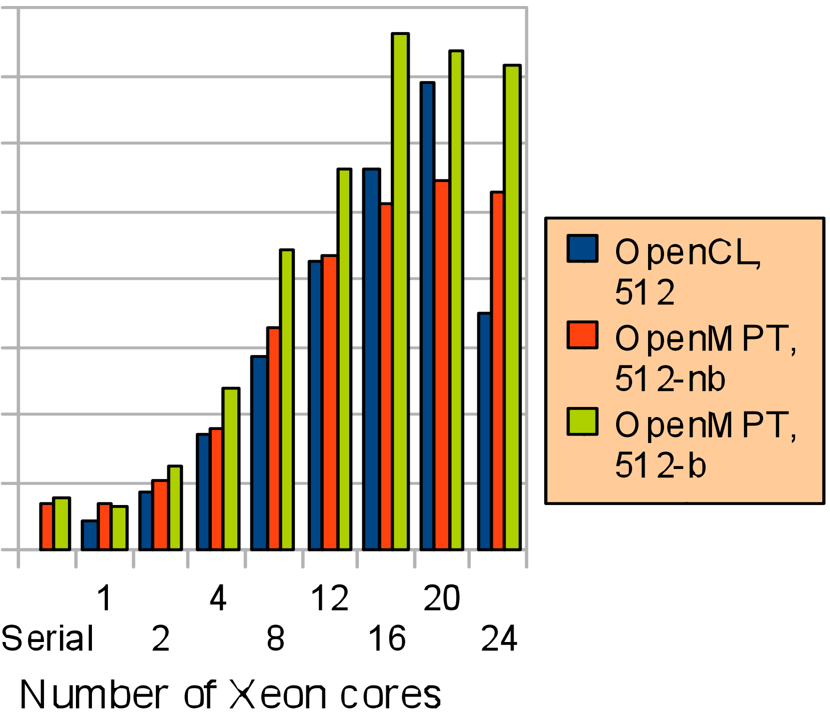
6 Simple example of array sections

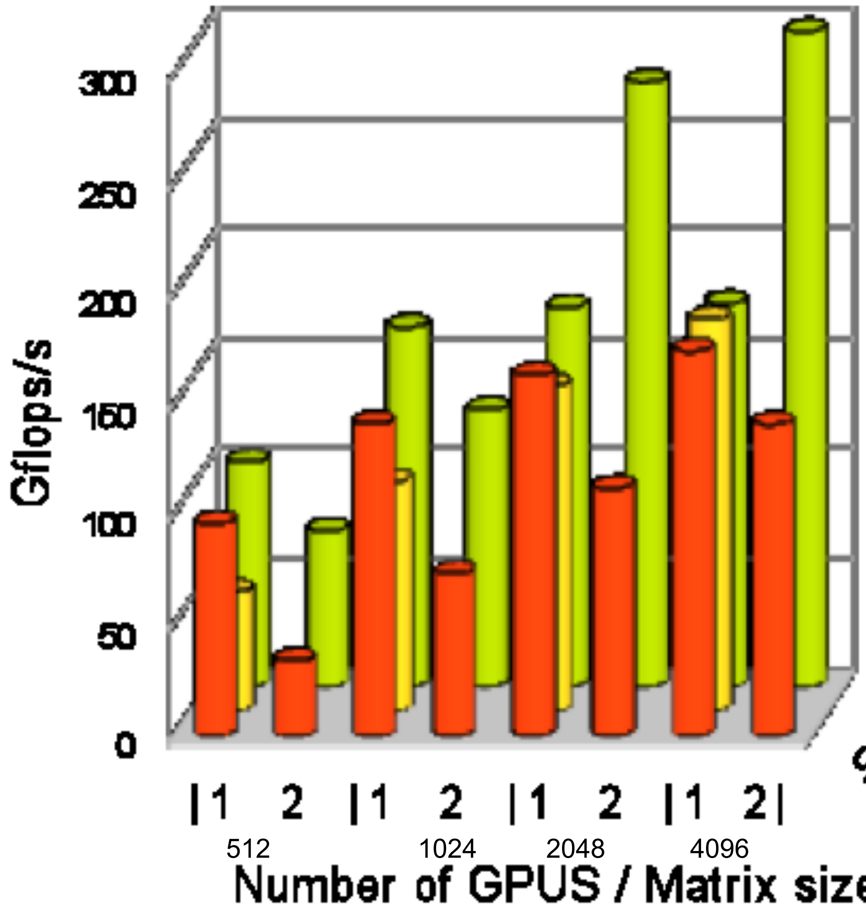
```
d f (int *a, int **b, int c[N][N])

agma omp task input(a)              //refers to the pointer a
fa(a);
agma omp task input([N] a)  \           //refers to the N elements pointed by a
              output([N] a[N/2:])\      //refers to the elements N/2 to N-1
              inout([N][N] b[:][1:10]) // refers to the submatrix [0..N-1][1..10]
fb(a,b);                                //pointed by b

agma omp task input(c) \ //refers to the pointer c
              inout([N][N] c) //refers to the matrix of NxN pointed by c
fc(c);
```

- Models

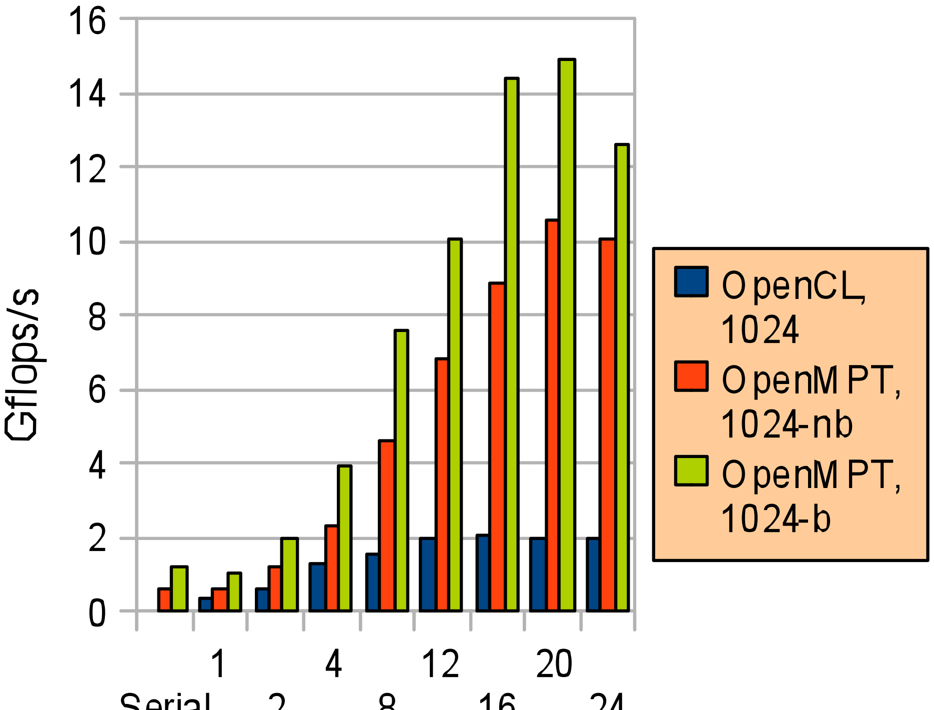
<table>
<tr><th>Model</th><th>Cell /B.E.</th><th>x86_64</th><th>GPUs</th></tr>
<tr><td>opencl</td><td>IBM OpeCL SDK</td><td>AMD/ATI OpenCL SDK</td><td>NVIDIA OpenCL SDK</td></tr>
<tr><td rowspan="2">StarSs</td><td rowspan="2">CellSs</td><td>Nanos++</td><td rowspan="2">Nanos++ (CUDA kernels)</td></tr>
<tr><td>SMPSs</td></tr>
<tr><td>OpenMPT</td><td>-</td><td>Nanos++</td><td>Nanos++ (CUDA kernels)</td></tr>
</table>

- Benchmarks
  - Matrix Multiply
  - Blackscholes – Computers pricing of European-stye options
  - Perlin Noise – Computes an image filled with noise to improve realistic view of moving graphics
  - Julia set – Computes a set of images of the Julia Set fractal

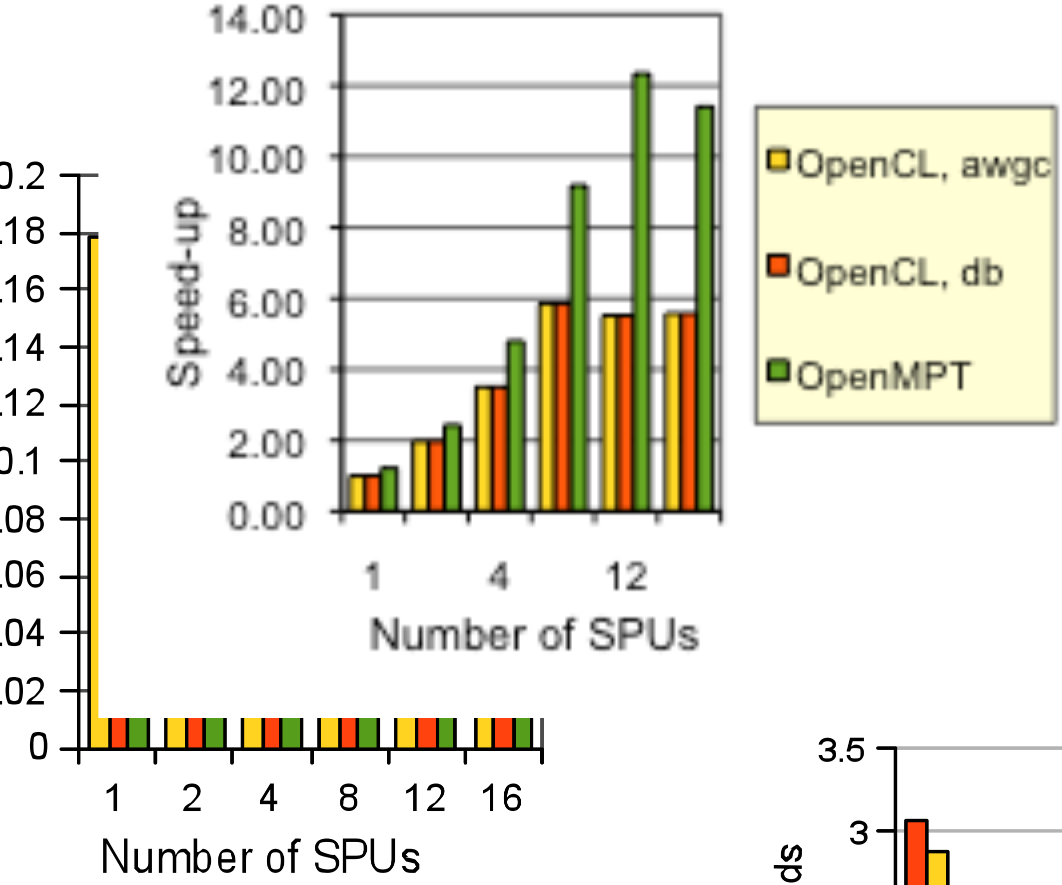




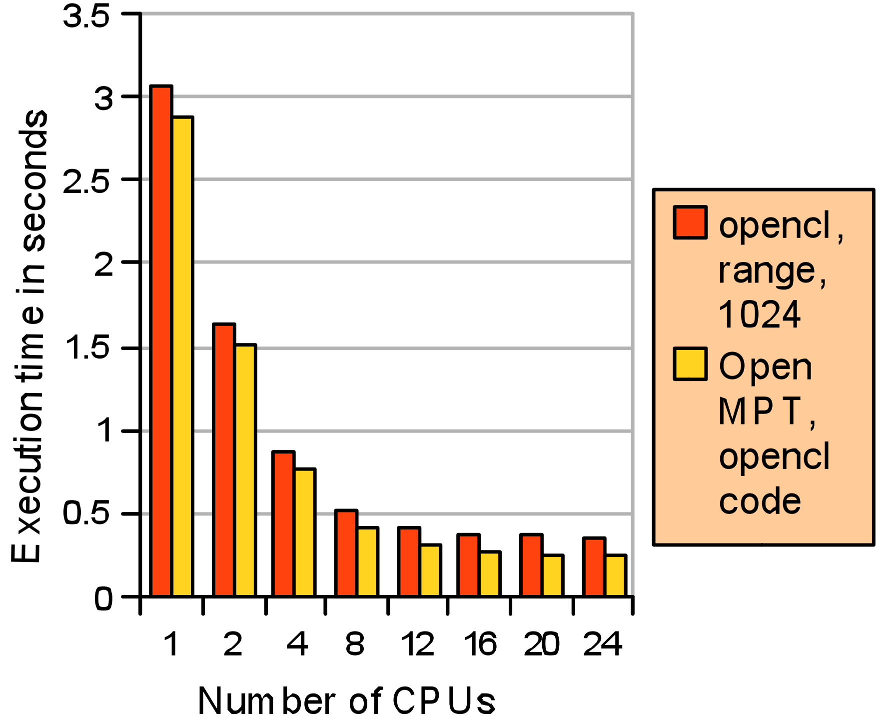
Intel Xeon Server



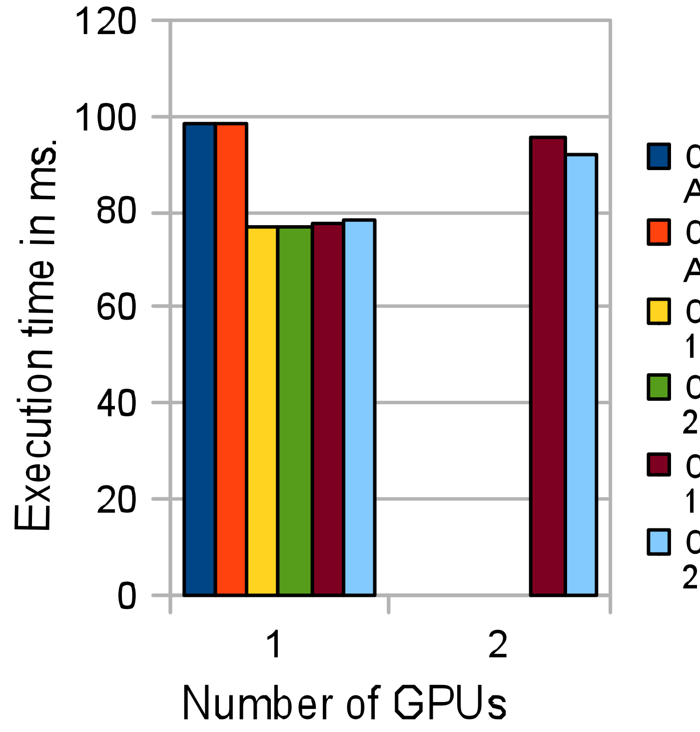
Nvidia GPU GTX 285



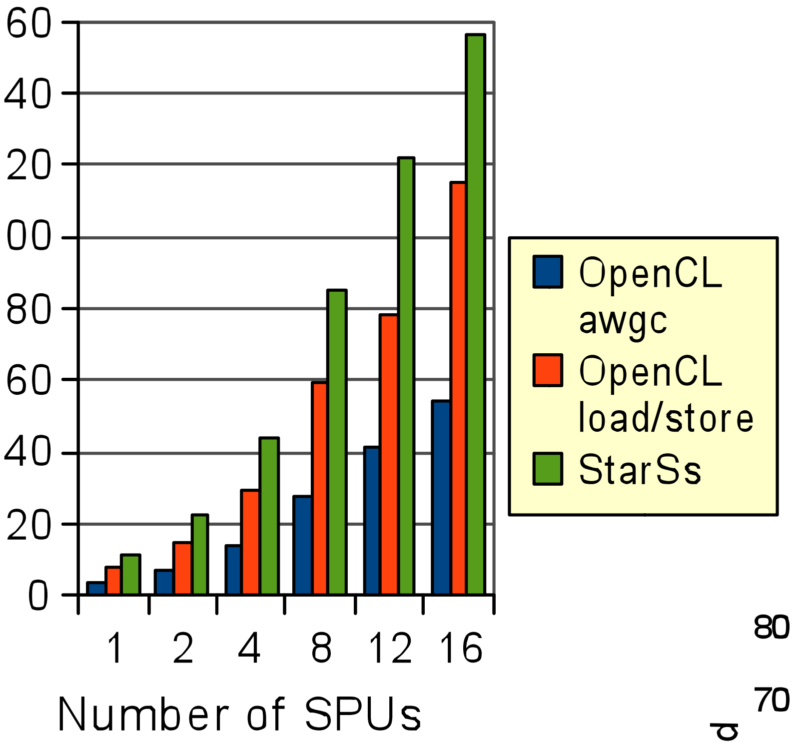
Cell BE



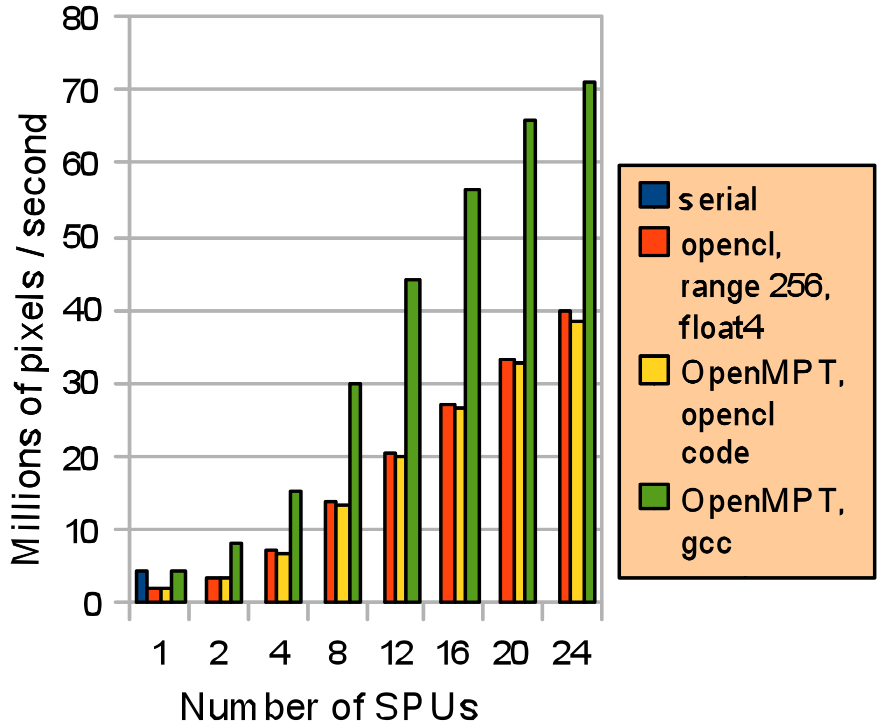
Intel Xeon



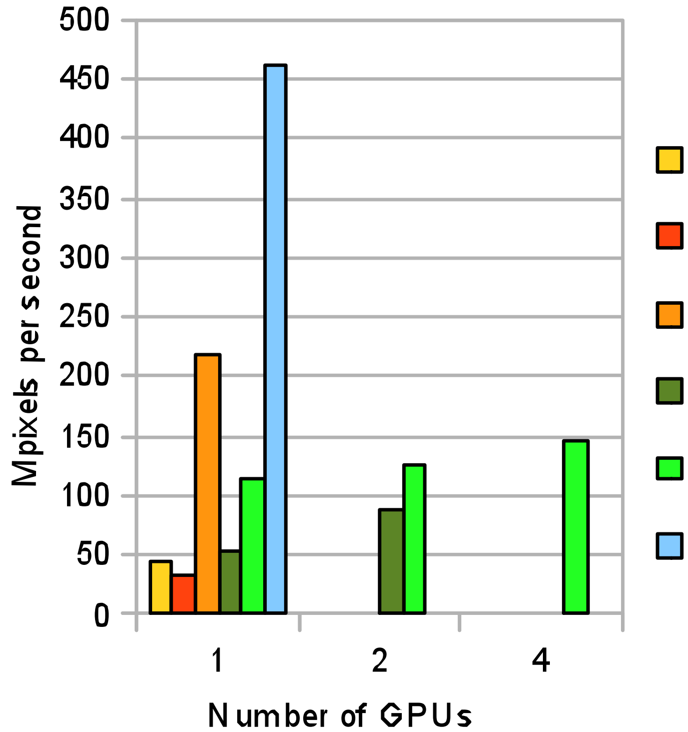
Nvidia GPU GTX 285



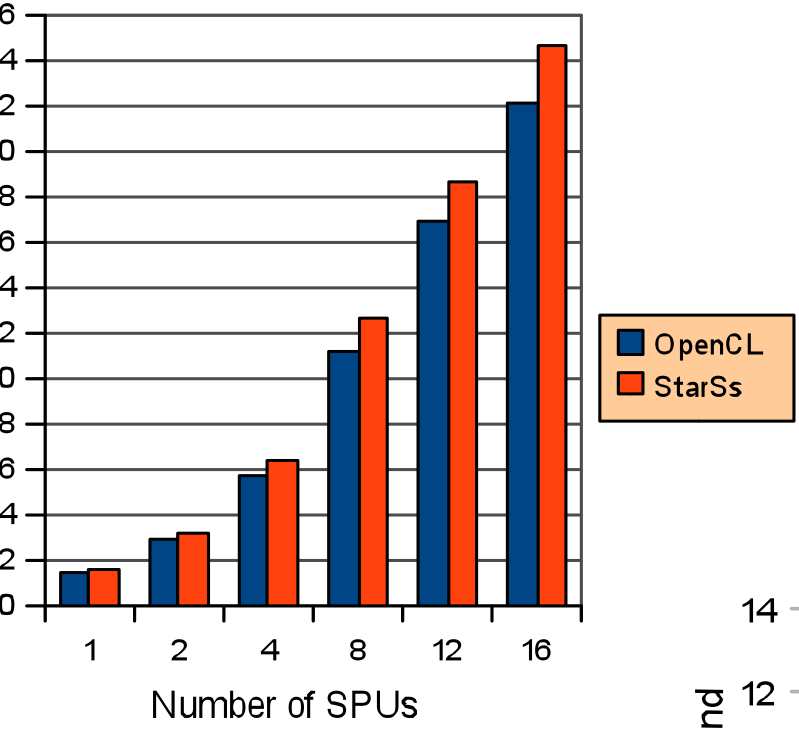
Cell BE



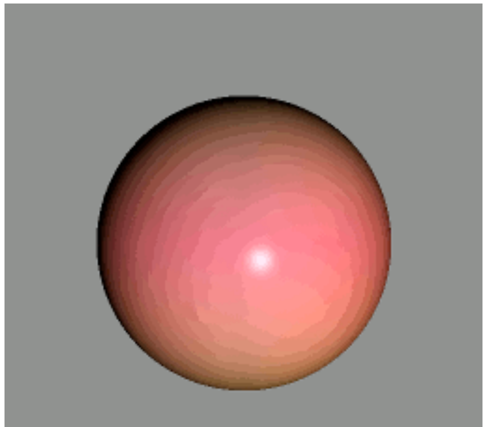
Intel Xeon

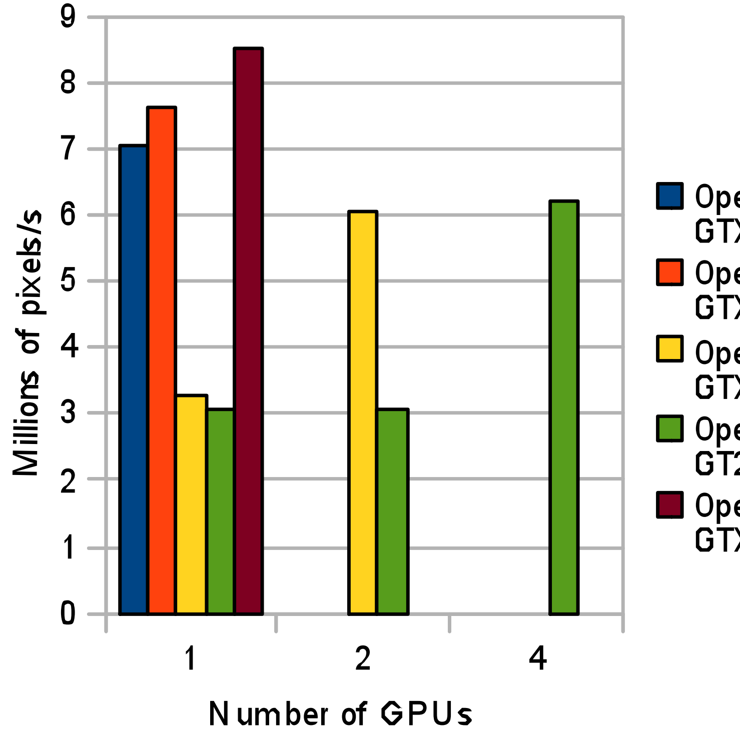


Nvidia GPU GTX 285

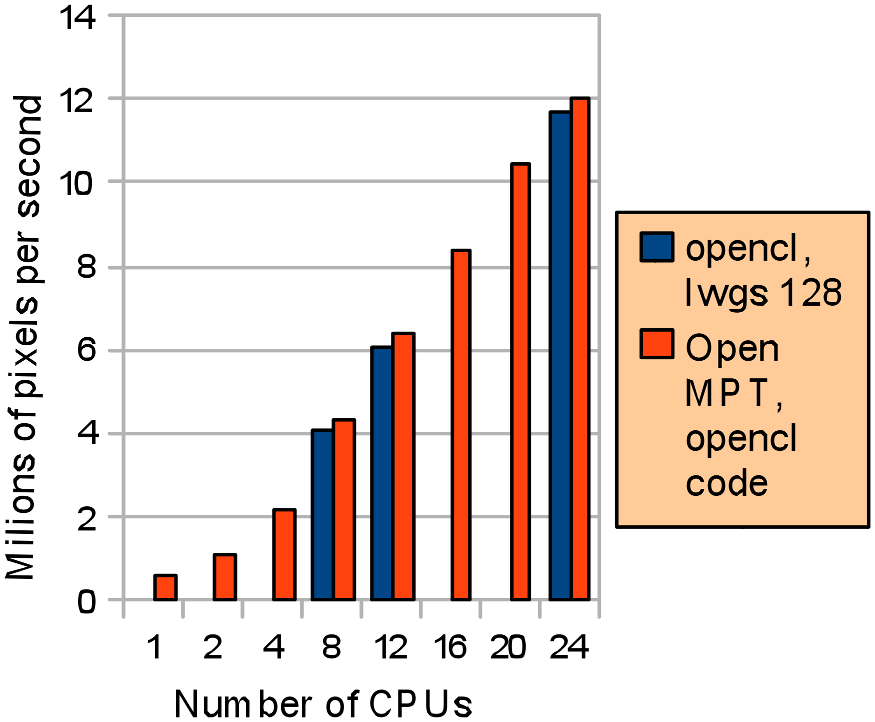


Cell BE





Nvidia GPU
GTX 285



Intel Xeon

- OpenMPT integrates ideas from StarSs and OpenMP
  - Support for task dependences, enabling data-flow like execution and exploitation of local
  - Support for heterogeneity, increasing the portability of applications by means of specializ kernels for each architecture
  - Based on source to source compilation and intelligent runtimes
- Distributed as open source:
  - StarSs releases:
    - CellSs: www.bsc.es/cellsuperscalar
    - SMPSS: www.bsc.es/smpsuperscalar
    - OpenMPT: http://www.bsc.es/plantillaG.php?cat_id=328