INSTITUTE OF COMPUTING TECHNOLOGY

Programming Clouds

Zhiwei Xu 徐志伟
Institute of Computing Technology (ICT)
Chinese Academy of Sciences
zxu@ict.ac.cn

National Basic Research Program of China

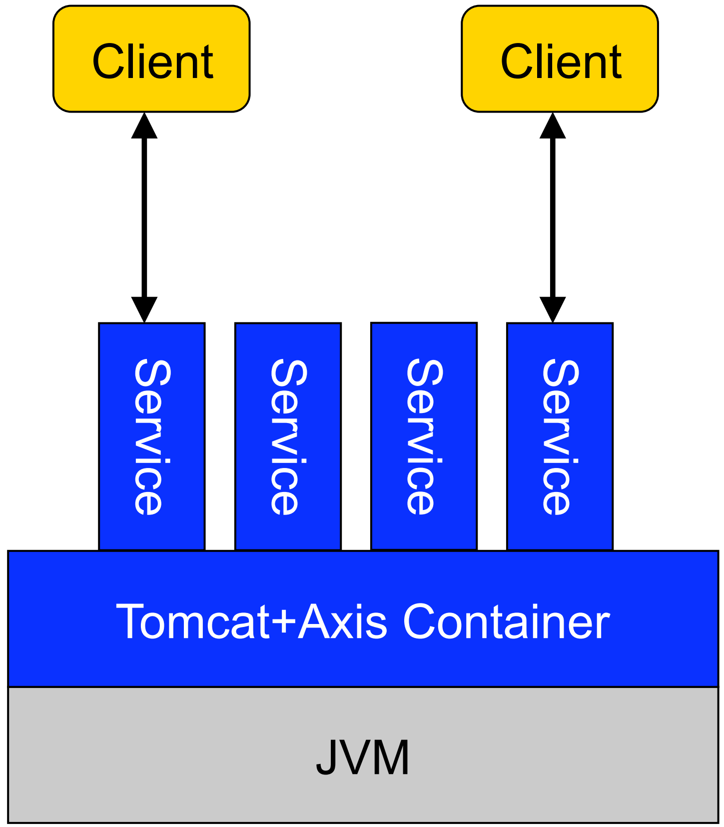
European Commission

XtreemOS
Enabling Linux for the Grid

Contents

- **What is cloud programming?**
- **Landscape of concurrent programming**
- What can we learn from
  - Intra-Process
    - SIMT: CUDA
    - Transactional Memory
  - Inter-Process
    - Map-Reduce, Pig
    - GSML

A process could be huge
Could have many I/O, sys



Cloud Definition: User's Viewpoint

A, ***D***
O, ***P***
C, ***N***, ***S***

- A net computing technology that
  - Provides 7 types of resources
  - For institutional and personal users
  - The resources are
    - In the cloud (Net)
    - Virtualized
    - Owned on demand
    - Used on demand
    - Easy to own and use
- In the cloud: virtualized resources in the Net
- Service: can get the value, not physically owned
- On demand: low cost, flexible
- Virtual ownership: user in control, service quality guarantee
- Ease: fast, low cost

Amazon EC2 Example

A, *D*
O, *P*
C, *N*, *S*

- Cloud
- Service

	Small	Large	Extra Large
Bits	**32**	**64**	**64**
RAM	**1.7 GB**	**7.5 GB**	**15 GB**
Disk	**160 GB**	**850 GB**	**1690 GB**
Compute Units	**1**	**4**	**8**
I/O	**Medium**	**High**	**High**
Firewall	**Yes**	**Yes**	**Yes**

On Demand

- A user wants to render an animation movie of 60 minutes, with 30x60x60=108,000 frames. Need to do it ten times
- Rendering one frame needs 20 seconds
- To buy a PC to do the rendering
  - 10x2,160,000 s = 6000 h = 250 days, ￥7000
- To use EC2
  - With108,000 AMIs, needs half a day, ￥150

Virtual Ownership

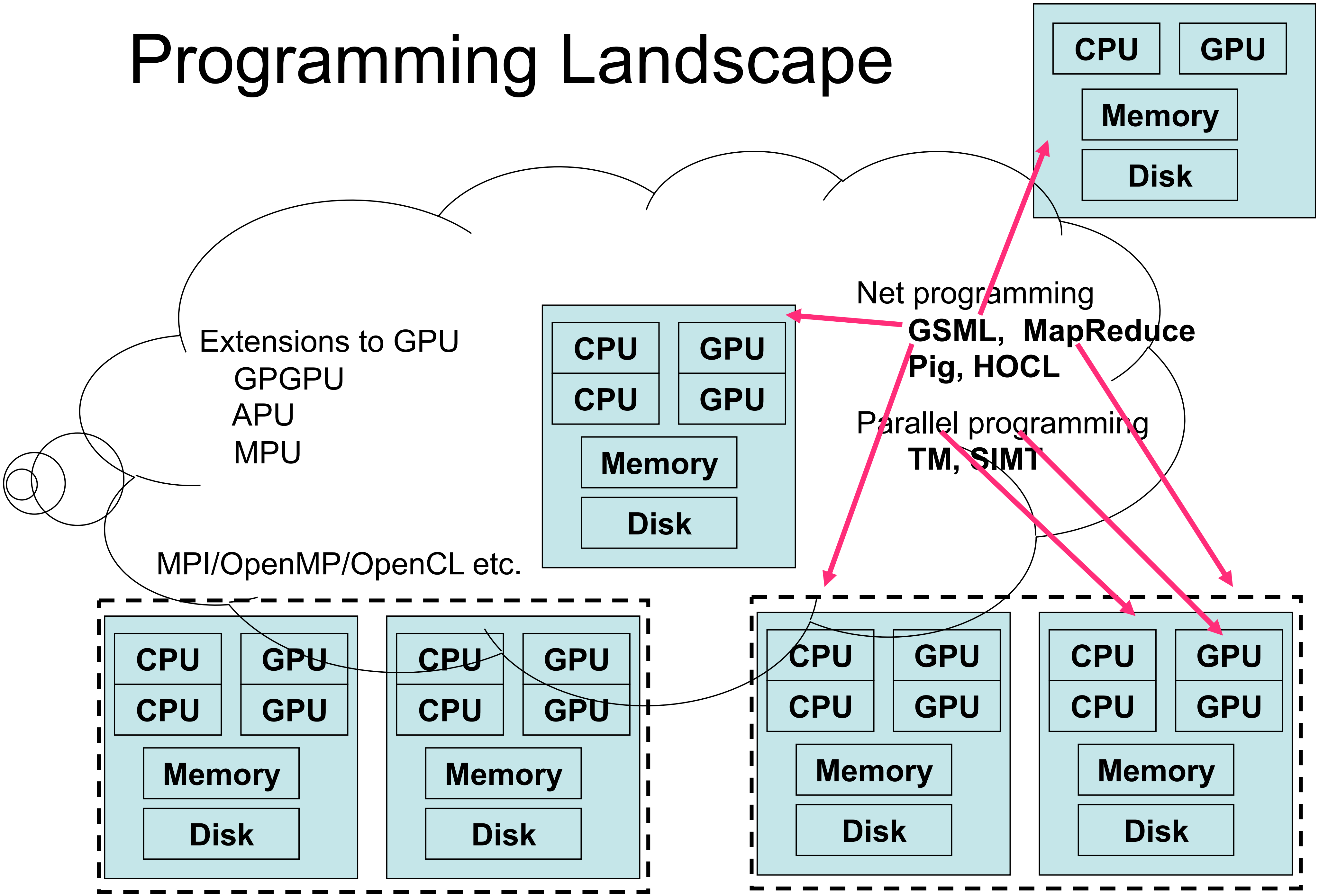
- A user owns AMI (and the underlying EC2/S3), as if he owns a PC or server
  - Linux OS
  - Can develop, deploy, use various software and data
  - Which value can be used as Net services
- Amazon's "guarantee" of service quality
  - Amazon SLA
  - >1000 production users
- Amazon S3 disruptions
  - 2008.2 2 hours, 2008.7 8 hours

Ease to Own and Use

- Understand HowTo: <20 minutes
- Register to become a user <15 minutes
- Create AMI’s <5 minutes

- A potential user only needs
  - An email address
  - A credit card

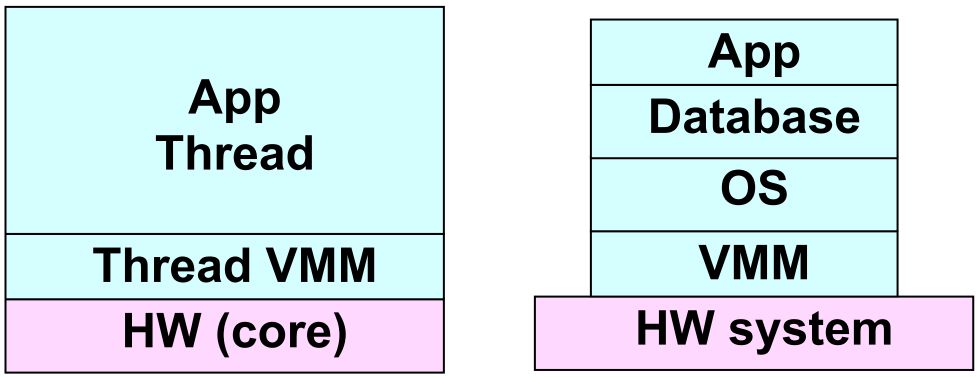
Programming Landscape



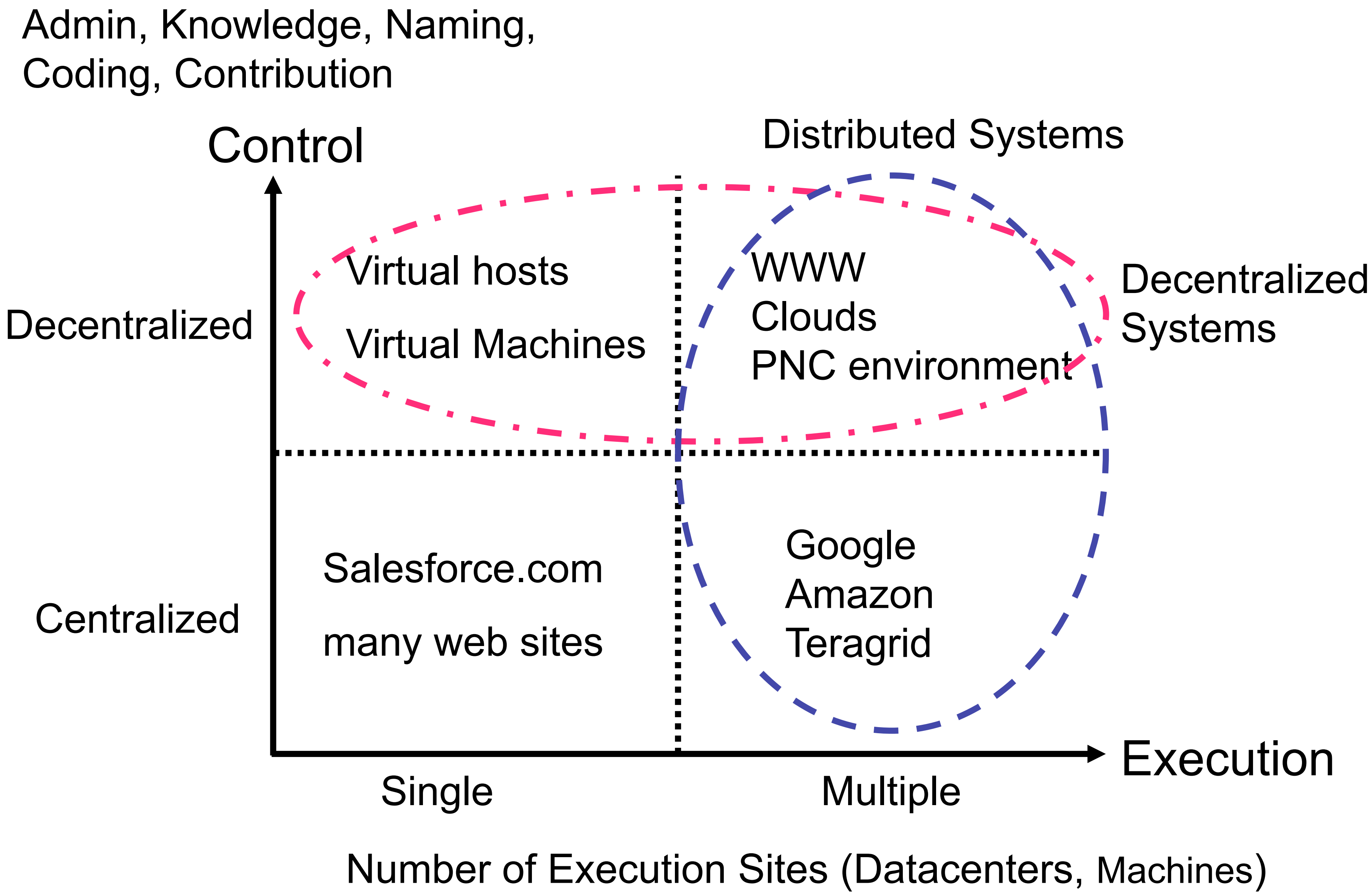
Clash of the Computer and the Network Approaches

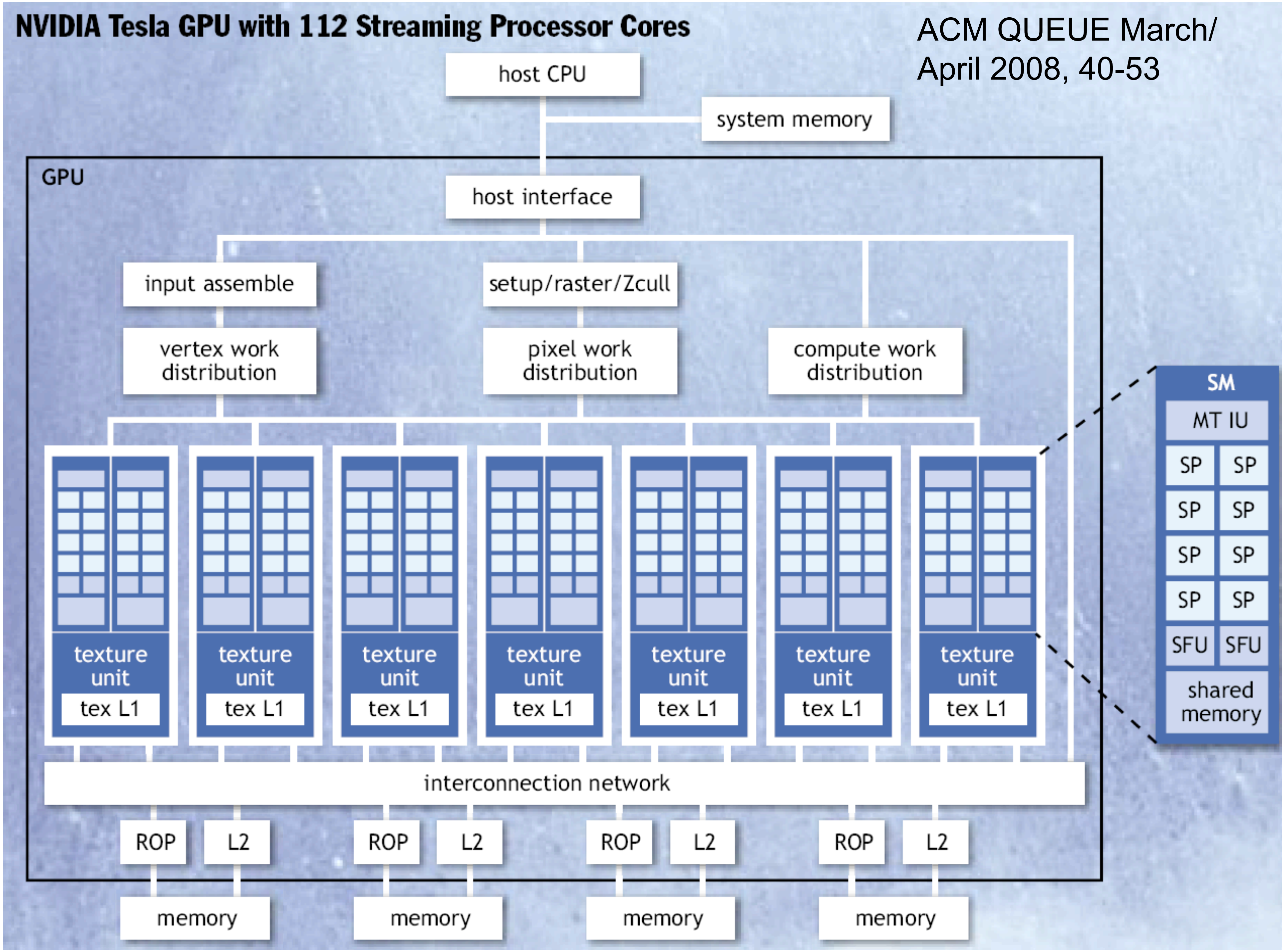
- Fetching 10-byte data from a blog server: 162 ms, 52 context switches at server side
- Sustained < 5% Peak?
- Many levels of programming interfaces
- New coupling

TCP/IP Stack	Web/Web Service Stacks
4 Application	HTML, HTTP; GSML, BPEL, WSRF, WSDL, SOAP, XML
3 Transport	?
2 Inter Network	
1 Network Access	



Distributed and Decentralized Architecture





ACM QUEUE March/ April 2008, 40-53

Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, __texture__
- Function prototypes
  - __global__ void kernelOne(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution
- Kernel – void kernelOne(type args,…)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
  - __syncthreads()…

Example: Matrix Multiplication

- Objective: matrix computing: C = A(wA , hA) x B (wB , wA)
- Method:
  - **tiling** matrix C to square sub-matrix(Csub) :
    improving ratio of compute to off-chip memory access to
    (wA*wB)/(block_size*block_size)
  - **Massive thread level computing parallelism** :
    1.each block : /* must be within one SM */
    computing one square sub-matrix Csub of C;
    2.each thread within block : /* thread executed on one core at a time */
    computing **one element of Csub** ;
    3.block size of Csub = 16 ,respectively **256-thread/block** :
    a. multiple of warp size for no computing resources idle (32 physical thread/per warp)
    b. one steam multiprocessor in G80 can take up 768-thread: 3-block x 256(thread/block), so simultaneously executing 32-thread of a **warp** choosing from these 3 blocks.
- Host side code (the host machine)
- Device side code (the G80 graphic card)

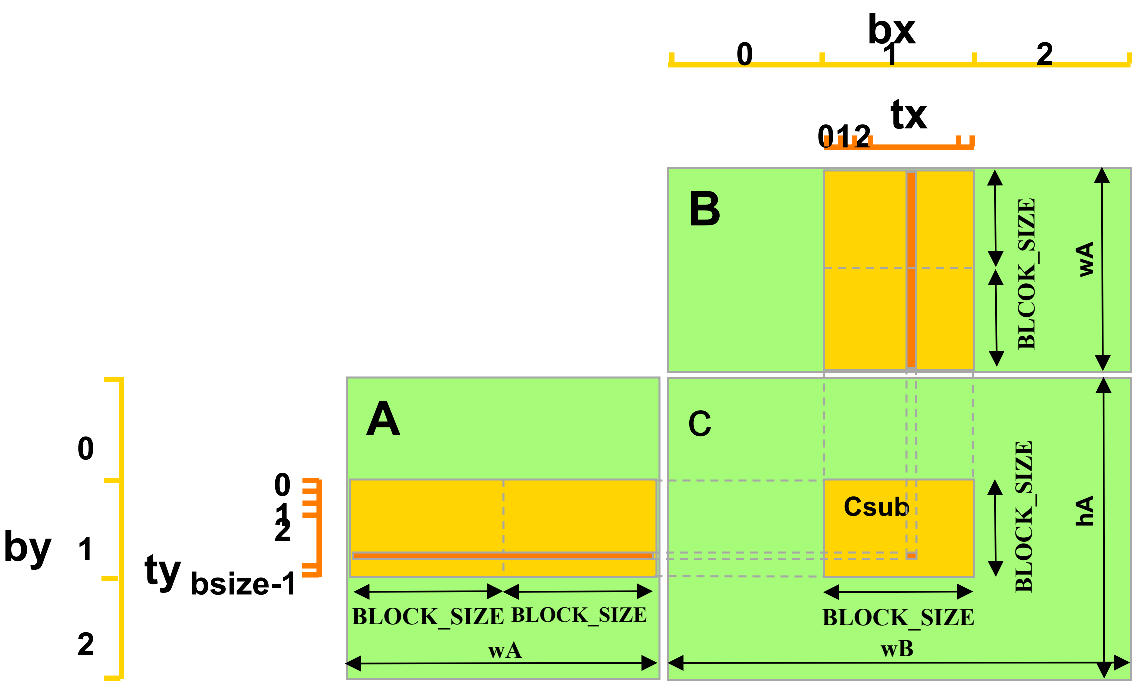


Fig. 1 Matrix Multiplication

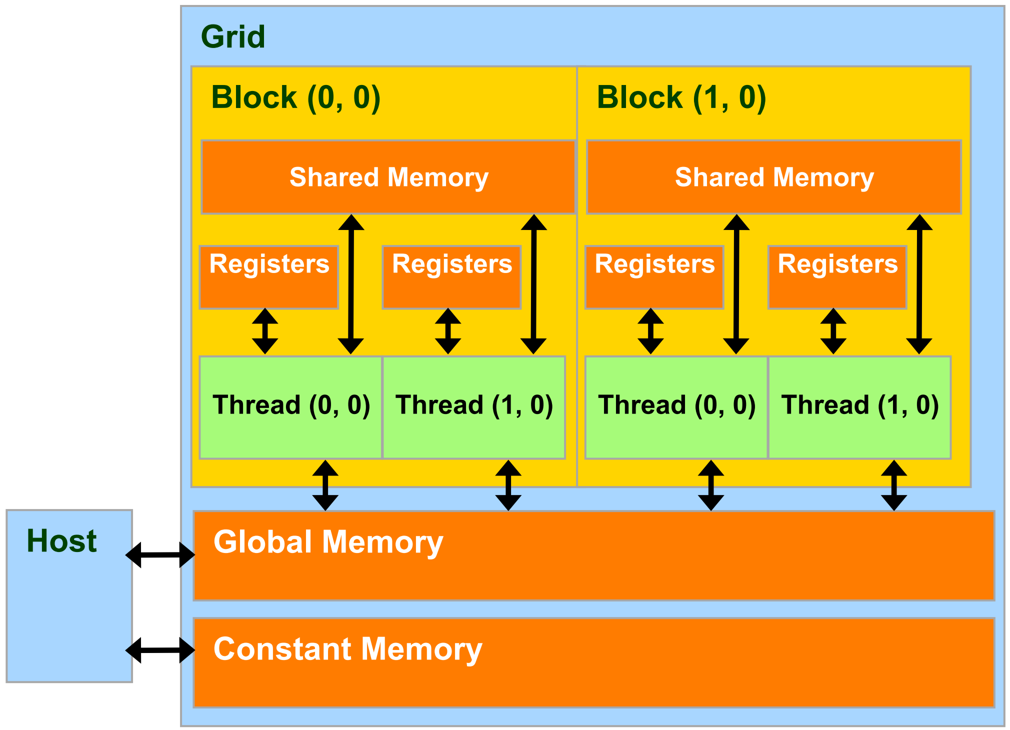


Fig. 2 G80 implementation of CUDA Memories

Step 1: Input Matrix Data Transfer

(Host-side Code)

```
// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);
// Host multiplication function
void Mul(const float* A, const float* B, int hA, int wA, int wB,
float* C)
{
1. // Allocate and Load M, N to device memory
int size = hA * wA * sizeof(float);
cudaMalloc((void**)&Ad, size);
cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
float* Bd;
size = wA * wB * sizeof(float);
cudaMalloc((void**)&Bd, size);
cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
// Allocate C on the device
float* Cd;
size = hA * wB * sizeof(float);
cudaMalloc((void**)&Cd, size);
```

__global__ defines a kernel function called by **host** but executed on **device**

allocates **global** memory on **device** (Fig. 2) to store A

copies A from **host** memory to **global** memory

Step 2: Output Matrix Data Transfer
(Host-side Code)

2. // Kernel invocation code – to be shown later in Step 4;

   …

3. // Read Cd from the device

```
cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(Ad);      cudaFree(Bd);      cudaFree(Cd);
}
```

Note: cudaMemalloc()/cudaMemcpy()/cudaFree()
They are **API** functions of CUDA's **runtime** and used to allocated **linear memory** and transfer data between host and device.

Step 3: Kernel Function

(device-side code)

// Matrix multiplication kernel – per thread code

```
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
// Block index and thread index:
int bx = blockIdx.x; int by = blockIdx.y;
int tx = threadIdx.x; int ty = threadIdx.y;
// Index of the first sub-matrix of A processed by the block
int aBegin = wA * BLOCK_SIZE * by;
// Index of the last sub-matrix of A processed by the block
int aEnd = aBegin + wA - 1;
// Step size used to iterate through the sub-
int aStep = BLOCK_SIZE;
// Index of the first sub-matrix of B processe
int bBegin = BLOCK_SIZE * bx;
// Step size used to iterate through the sub-matrices of B
int bStep = BLOCK_SIZE * wB;
// The element of the block sub-matrix that is computed  by the thread
float Csub = 0;
```

CUDA's keyword:
Block&thread shape :1D/2D/3D
facilitate selecting work and
address shared data
(bx/by tx/ty see Fig.1)

From Fig.1 :
(0,0) is at the upper left corner
X means horizontal;
Y means vertical.

Step 3: Kernel Function(cont)

```
for (int a = aBegin, b = bBegin;
    a <= aEnd; a += aStep, b += bStep) {
    // Shared memory for the sub-matrix of A and B
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // each thread loads one element of each matrix
    // Load the matrices from global memory to shared memory;
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    // Synchronize to make sure the matrices are loaded
    __syncthreads();
    // Multiply the two matrices together;
   for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
    __syncthreads();
    }
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
} // Write the block sub-matrix to global memory;
```

Why As/Bs located in "share memory" see Fig.2 :

16K-Byte on chip;

16-bank: suport 16 **simultaneous** accesses when **no** bank confict;

2-cycle access delay compare to 200-cycle delay of **global** memory !

__syncthreads():

CUDA'S intrinsics:

1. like barrier() ;
2. but only synchronizes all threads in a block;
3. guarantee memory consistency (e.g.store serializing) to avoid RAW hazard in shared or global memory

Step 4: Kernel Invocation

(Host-side Code)

// Setup the execution configuration

```
dim3 dimBlock (BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid (wB / dimBlock.x, hA / dimBlock.y);
```

// Launch the device computation threads!

```
Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
```

Built-in Variables(reseved) of CUDA

dim3 gridDim:

Dimensions of the grid in blocks

(**gridDim.z** unused)

dim3 blockDim:

Dimensions of the block in threads

Here:

thread array is 2D: 16x16

block array is 2D: wB/16 x hA/16

since block size is 16.

Host code uses

“<<<dimGrid, dimBlock>>>” as execution configuration to call function Muld.

Why Transactional Memory

- Pitfalls with locks:
  - Priority inversion. A lower priority thread is preempted while holding a lock which is needed by high priority threads.
  - Convoying. When a thread holding a lock is de-scheduled or interrupted, other threads that need the lock are queue up, unable to progress.
  - Deadlock. Threads attempt to acquire locks in different order.
- Atomic primitives such as CompareAndSwap() operate on only one word at a time, resulting in complex algorithms.
- Compositionality. It is difficult to compose multiple calls to multiple objects into atomic sections.

Basic Semantics of Transactional Memory

- Transaction: a sequence of steps executed by a single thread. Allow atomic updates to multiple memory locations.
  - Serializability. Transactions must appear to execute sequentially, in a one-at-a-time order. Do not deadlock or livelock.
  - Atomicity. Transactions are executed speculatively, meaning they only make tentative changes to objects. If a transaction completes without synchronization conflict, then it **commits**. Otherwise it **aborts**. Intermediate states are not observable to other transactions.
- Nested transaction
  - One method can start a transaction and then call another method without worrying about whether or not the nested method call starts a new transaction.
  - A nested transaction can abort without aborting its parent.

TM example I: the enq() method

- The enq() method of a unbounded transactional queue object. All operations within enq() either complete atomically or abort without any side effect.

```
Void enq(T item){
   atomic {
        //construct a new node
        NodeType node = new_node(item);
        //insert the node into the unbounded queue
        node.next = tail;
        tail = node;
   }//atomic
}//enq
```

TM example II: the enq() method with retry mechanism

- The enq() method of a bounded transactional queue. The method enters an atomic block and tests whether the queue is full. If so, it calls retry, which rolls back the enclosing transaction, pauses it, and restarts it later.

```
Void enq(T item){
  atomic {
        if(count == items.length)
          retry;
        items[tail] = item;
        if(++tail == items.length)
          tail = 0;
        count++;
  }//atomic
}//enq
```

TM example III: composing transactions

- The deq_enq() method composes a deq() call that dequeues an item x from a queue q0 and an enq() call that enqueues that item to another queue q1.

```
Void deq_enq(QueueT q0, QueueT q1){
    atomic {
            NodeT item = q0.deq();
            q1.enq(item);
    }//atomic
}//enq
```

TM example IV: conditional synchronization

- The multiple_deq() call succeeds if **either** sub-transaction q0.deq() completes **or** sub-transaction q1.deq() completes.
- The **orElse** statement joins two or more code blocks. The thread first executes the first block. If it calls **retry**, then that sub-transaction is rolled back, and the thread executes the second block. If that block also calls **retry**, then the **orElse** as a whole pauses, and later reruns each of the atomic blocks until one of them completes.

```
Void multiple_deq(QueueT q0, QueueT q1){
    atomic {
            NodeT item = q0.deq();
    } orElse {
            NodeT item = q1.deq();
    }
}//enq
```

Some Challenges of TM

- I/O: writes to disk, display, network, etc
  - I/O operations are hard to roll back
- Performance isolation
  - Most hardware TM can not context switch within a transaction
  - Long transactions can block progress
  - OS system calls put kernel resources inside transactions
- Real time
  - TM makes real time software more challenging

Pig Example

Chris Olston et al, Yahoo! Research

Find the top 10 most visited pages in each category

Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

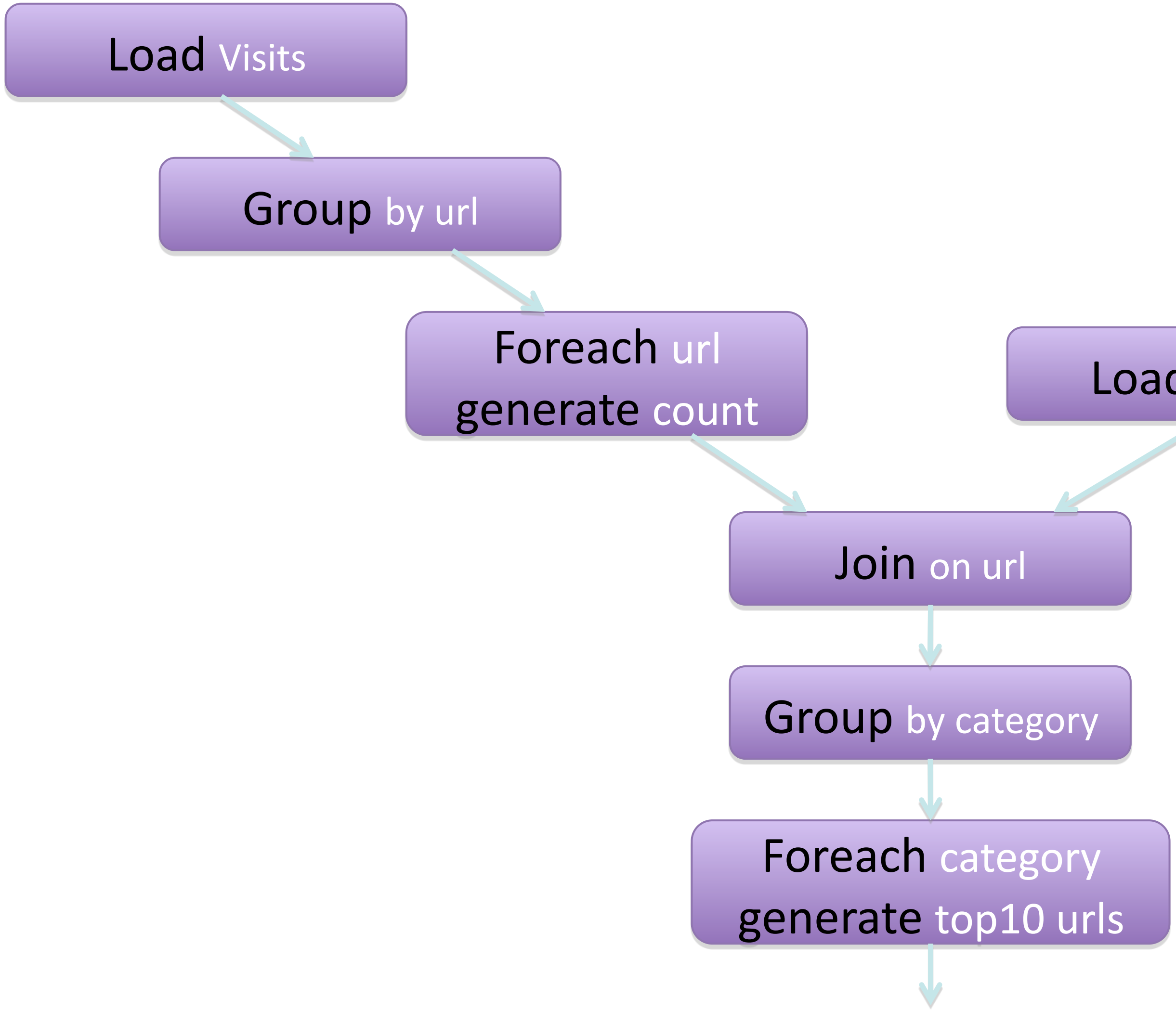
⋮

Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

⋮

Data Flow


Load Visits
Group by url
Foreach url generate count
Load Url Info
Join on url
Group by category
Foreach category generate top10 urls

In Pig Latin

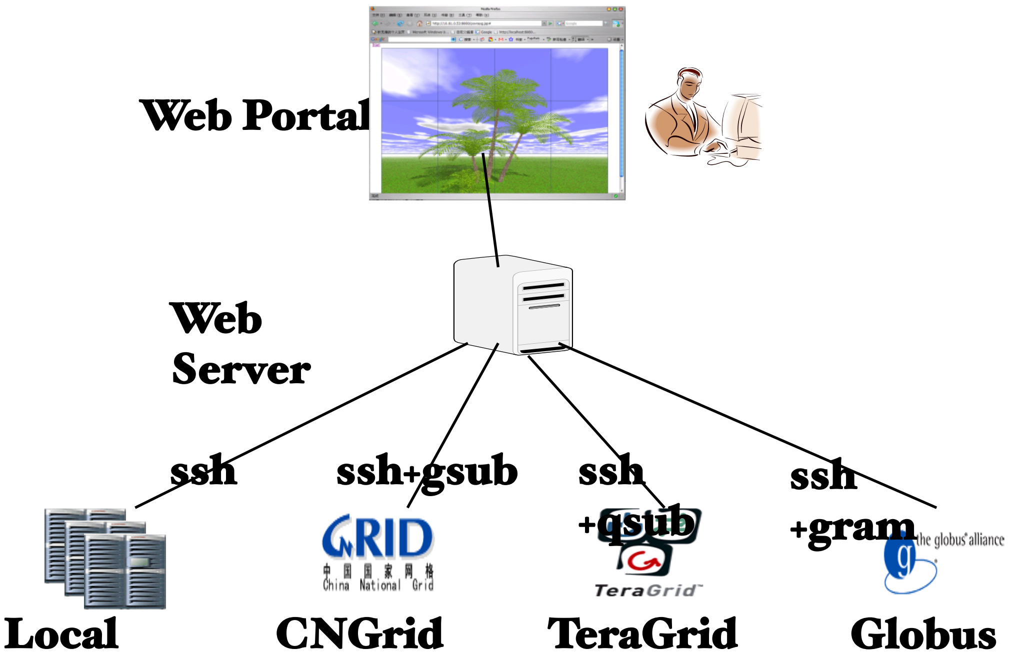
```
visits           = load '/data/visits' as (user, url, time);
gVisits          = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo          = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Traditional: Povray-G



	Lines of Codes	Development Time
UI	74 (html, css, javascript)	155
App Logic	121 (java, JSP)	45
Others	133 (javascript, java)	125
Accessing Resources	379 (Java)	290
Total	712	615

Programming Difficulties

- Many languages
- Multiple modules
- JSP tight coupling
- Web Server is the bottleneck

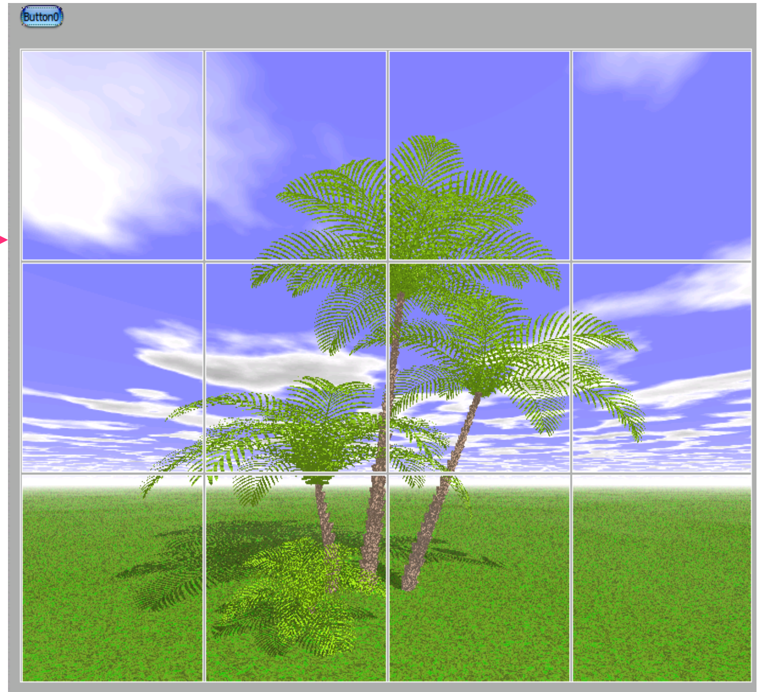
- Overhead in accessing resources: 54% codes, 48% time
- Other overheads (communication, job partition, parallelism): 19% codes, 20% time

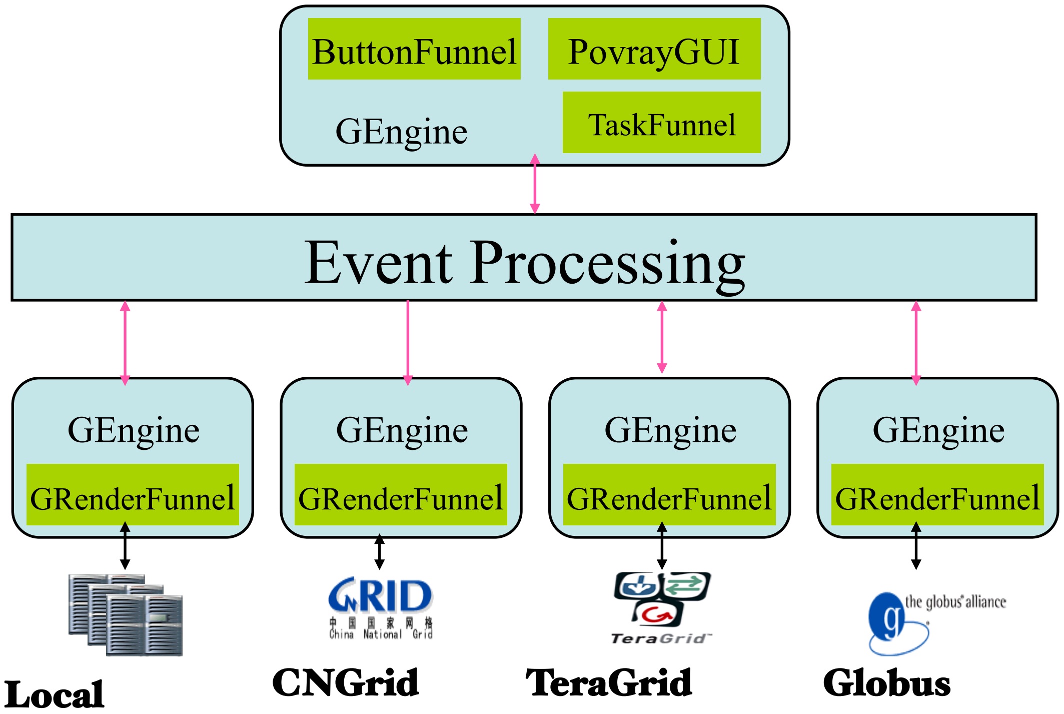
GSML：Povray-G

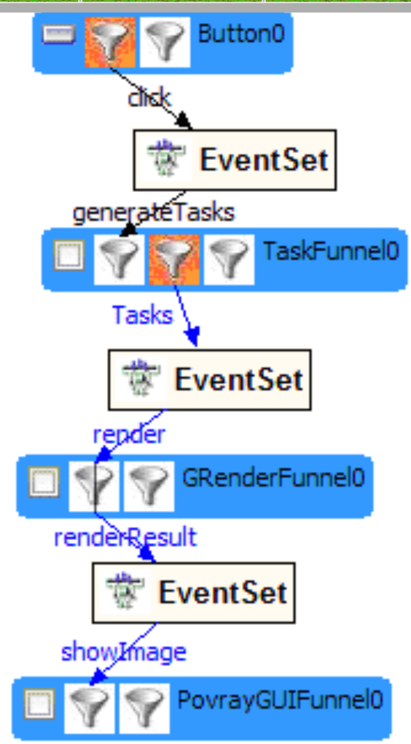
UI

App Logic

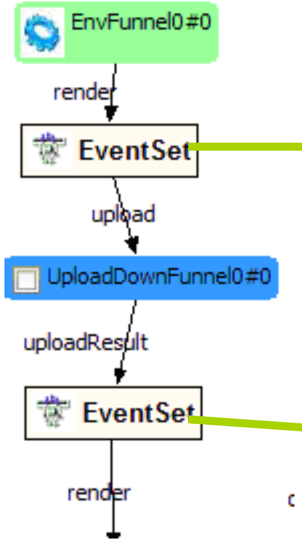
Environment





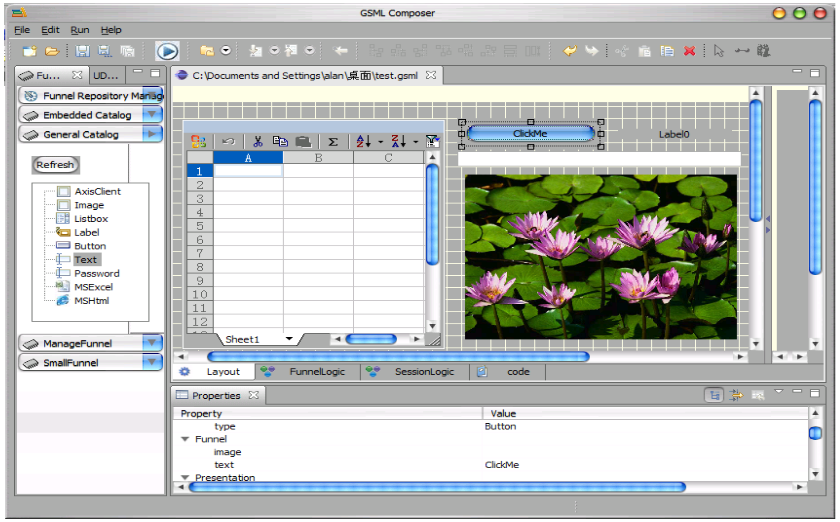


GRenderFunnel in Povray-G

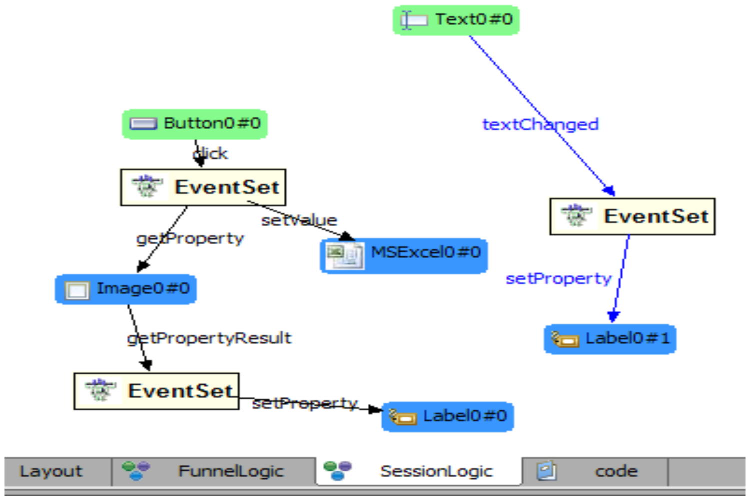


addEventHandler(“render”, new EventHandler(){ //当GRenderFunnel收到render事件触发
public void handle(renderEvent){
String povFile = renderEvent.getParamVal(“povFile”); //从事件中获取渲染的pov文件
EventWait uplodwait = uploadAgent.sendEventWithAck(uploadEvent); //上传文件并返回等待句柄
EventSelect es = new EventSelect(new EventWait[]{uploadwait});//创建eventSelect
EventWait ew = es. Select(); //阻塞语义的调用EventSelect.select()
String result= ew. getEvent().getParamVal(“result”); //从事件中获取上传结果
renderPngEvent.setParamVal(“configFile”,povFile);
renderAgent.sendEvent(renderPngEvent); //renderPngEvent需要参数povFile.
}});

GSML Editor

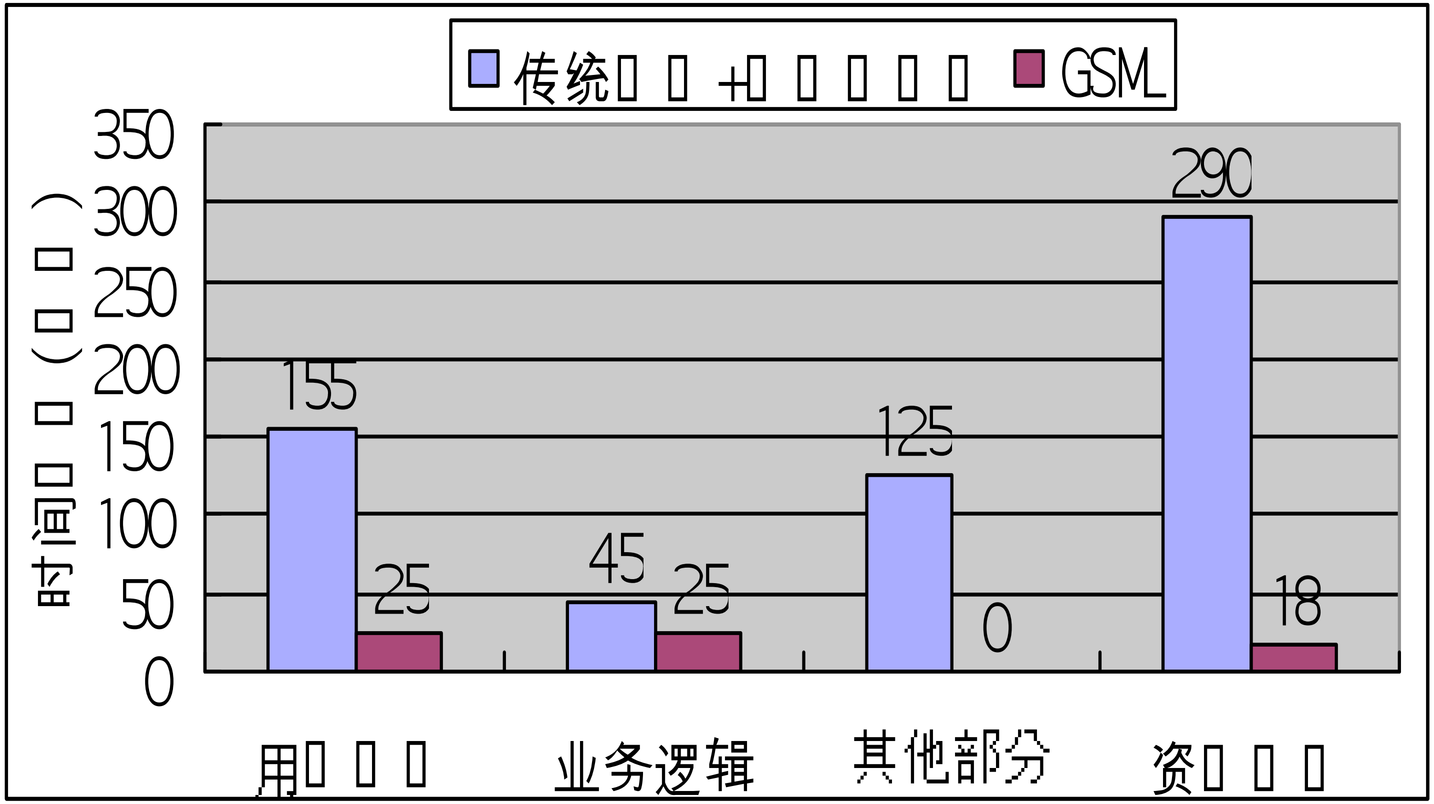


App Logic Composer



GSML vs. Traditional

- **Lines of Codes Reduced 18.5%, Time reduced 88.9%**
  - **UI: 84%**
  - **App Logic: 44%**
  - **Accessing Resources: 94%**
  - **Others: 100%**



Summary

- Many programming models are being researched and used for parallel and net computing, now clouds
- Main issues
  - Efficiency
  - Correctness (e.g., eventual consistency)
  - Usability
- Many open problems
  - Evaluation workloads, metrics
  - What are the suitable models (cf: SIMT)

zxu@ict.ac.cn