For a given point, a **stencil** is a fixed subset of nearby neighbors.

A **stencil code** updates every point in an $d$-dimensional spatial grid at time $t$ as a function of nearby grid points at times $t-1$, $t-2$, ..., $t-k$, for $T$ time steps.

Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and adaptive mesh refinement, as well as for image processing and geometric modeling.
Example: 2D Heat Diffusion

Let $a[t, x, y]$ be the temperature at time $t$ at point $(x, y)$.

Heat equation

$$\frac{\partial a}{\partial t} = \alpha \left( \frac{\partial^2 a}{\partial x^2} + \frac{\partial^2 a}{\partial y^2} \right)$$

$\alpha$ is the thermal diffusivity.

Update rule

$$a[t, x, y] = a[t-1, x, y] + CX \cdot (a[t-1, x+1, y] - 2 \cdot a[t-1, x, y] + a[t-1, x-1, y]) + CY \cdot (a[t-1, x, y+1] - 2 \cdot a[t-1, x, y] + a[t-1, x, y-1])$$

2D 5-point stencil
**Implementation tricks**

- Reuse storage for even and odd time steps.
- Keep a **halo of ghost cells** around the array with boundary values.

```plaintext
for (t = 1; t <= T; ++t) {
    for (x = 0; x < X; ++x) {
        for (y = 0; y < Y; ++y) {
            // do stencil kernel
            a[t%2, x, y] = a[(t-1)%2, x, y]
            + CX*(a[(t-1)%2, x+1, y] - 2.0*a[(t-1)%2, x, y] + a[(t-1)%2, x-1, y])
            + CY*(a[(t-1)%2, x, y+1] - 2.0*a[(t-1)%2, x, y] + a[(t-1)%2, x, y-1]);
        }
    }
}
```

**Conventional cache optimization:** **loop tiling.**
for (t = 1; t <= T; ++t) {
    cilk_for (x = 0; x < X; ++x) {
        cilk_for (y = 0; y < Y; ++y) { // do stencil kernel
            a[t%2, x, y] = a[(t-1)%2, x, y] + CX*(a[(t-1)%2, x+1, y] - 2.0*a[(t-1)%2, x, y] + a[(t-1)%2, x-1, y]) + CY*(a[(t-1)%2, x, y+1] - 2.0*a[(t-1)%2, x, y] + a[(t-1)%2, x, y-1]);
        }
    }
}

- All the iterations of the spatial loops are independent and can be parallelized straightforwardly.
- Intel Cilk Plus provides a `cilk_for` construct that performs the parallelization automatically.
**Issue:** Looping is memory intensive and uses caches poorly. Assuming data-set size $N$, cache-block size $B$, and cache size $M < N$, the number of cache misses for $T$ time steps is $\Theta(NT/B)$.
Cache-Oblivious Stencil Code

Divide-and-conquer cache-oblivious [FLPR99] techniques, based on trapezoidal decompositions [FS05], are asymptotically efficient, achieving $\Theta(NT/MB)$ cache misses.

```c
void trapezoid(int t0, int t1, int x0, int dx0, int x1, int dx1) {
    lt = t1 - t0;
    if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * lt) {
        int xm = (2 * (x0 + x1) + (2 + dx0 + dx1) * lt) / 4;
        trapezoid(t0, t1, x0, dx0, xm, -1);
        trapezoid(t0, t1, xm, -1, x1, dx1);
    } else if (lt > 1) {
        int halflt = lt / 2;
        trapezoid(t0, t0 + halflt, x0, dx0, x1, dx1);
        trapezoid(t0 + halflt, t1, x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1);
    } else {
        for (int t = t0; t < t1; ++t) {
            for (int x = x0; x < x1; ++x)
                kernel(t, x);
        }
    }
}
```

Although these codes are efficient in practice, they can be significantly more difficult to write than nested loops.
Space-cutting rule: If the "zoid" to be computed is sufficiently wide, recursively compute the two subzoids that result from bisecting the zoid with a sloping line through the zoid’s center point.
Time-cutting rule: If the zoid admits no space cut (in any dimension), recursively compute the two subzoids that result from bisecting the zoid with a line of slope 0 through the zoid’s center point.
**Simulation: 3-Point Stencil**

- **Rectangular region**
  - $N = 95$
  - $T = 87$

- Fully associative LRU cache
  - $B = 4$ points
  - $M = 32$

- Cache-hit latency = 1 cycle
- Cache-miss latency = 10 cycles

Thanks to Matteo Frigo for this simulation code.
Pochoir Stencil Compiler

- Domain-specific compiler programmed in Haskell that compiles a stencil language embedded in C++, a traditionally difficult language in which to embed a separately compiled domain-specific language.
- Employs a novel cache-oblivious algorithm for arbitrary $d$-dimensional grids which is parallelized using Intel Cilk Plus.
- Makes it straightforward to code arbitrary periodic and nonperiodic boundary conditions, including Neumann and Dirichlet conditions.
- Implements a variety of stencil-specific optimizations.
- The stencil specification can be executed and debugged without the Pochoir compiler.
Benchmarking Platform

- Intel C++ version 12.0.0 compiler with Intel Cilk Plus.
- 12-core Intel Core i7 (Nehalem) machine, where each core has a private 32-KB L1-data-cache, a private 256-KB L2-cache, and a shared 12-MB L3-cache.
5-point stencil on a torus

Pochoir vs. Parallel Loop vs. Serial Loop
3D WAVE EQUATION

25-point stencil on a nonperiodic domain

Pochoir vs. Parallel Loop vs. Serial Loop
19-point stencil on a nonperiodic domain

Pochoir vs. Parallel Loop vs. Serial Loop
<table>
<thead>
<tr>
<th></th>
<th>Berkeley</th>
<th>Pochoir</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Xeon X5550</td>
<td>Xeon X5650</td>
</tr>
<tr>
<td>Clock</td>
<td>2.66GHz</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td>cores/socket, total</td>
<td>4, 8</td>
<td>6, 12</td>
</tr>
<tr>
<td>Hyperthreading</td>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>L1 data cache/core</td>
<td>32KB</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache/core</td>
<td>256KB</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 cache/socket</td>
<td>8MB</td>
<td>12 MB</td>
</tr>
<tr>
<td>Peak computation</td>
<td>85 GFLOPS</td>
<td>120 GFLOPS</td>
</tr>
<tr>
<td>Compiler</td>
<td>icc 10.0.0</td>
<td>icc 12.0.0</td>
</tr>
<tr>
<td>Linux kernel</td>
<td></td>
<td>2.6.32</td>
</tr>
<tr>
<td>Threading model</td>
<td>Pthreads</td>
<td>Cilk Plus</td>
</tr>
<tr>
<td>3D 7-point 8 cores</td>
<td>2.0 GStencil/s</td>
<td>2.49 GStencil/s</td>
</tr>
<tr>
<td></td>
<td>15.8 GFLOPS</td>
<td>19.92 GFLOPS</td>
</tr>
<tr>
<td>3D 27-point 8 cores</td>
<td>0.95 GStencil/s</td>
<td>0.88 GStencil/s</td>
</tr>
<tr>
<td></td>
<td>28.5 GFLOPS</td>
<td>26.4 GFLOPS</td>
</tr>
</tbody>
</table>
OUTLINE

• **Functional Specification**
• **How the Pochoir System Works**
• **Optimizing Strategies**
• **Algorithms**
• **Conclusion**
OUTLINE

• Functional Specification
• How the Pochoir System Works
• Optimizing Strategies
• Algorithms
• Conclusion
**FUNCTIONAL SPECIFICATION**

- Embedded in C++.
- Directly executable and debuggable via any native C++ tool chain.
- Supports arbitrary $d$-dimensional rectangular grids.
- The stencil shape can be arbitrary.
- A point at time $t$ can depend on points at time $t-1, t-2, \ldots, t-k$.
- Both periodic and nonperiodic boundary conditions can be programmed.
# 2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   Pochoir_Shape_2D 2D_five_pt[6]
6       = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7   Pochoir_2D heat(2D_five_pt);
8   Pochoir_Array_2D(double) a(X,Y);
9   a.Register_Boundary(zero_bdry);
10  heat.Register_Array(a);
11  Pochoir_Kernel_2D(kern, t, x, y)
12     a(t,x,y) = a(t-1,x,y)
13         + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
14         + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
15 Pochoir_Kernel_End
16  for (int x = 0; x < X; ++x)
17      for (int y = 0; y < Y; ++y)
18         a(0,x,y) = rand();
19  heat.Run(T, kern);
20  for (int x = 0; x < X; ++x)
21      for (int y = 0; y < Y; ++y)
22         cout << a(T,x,y);
23  return 0;
24 }
```
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2     return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5     Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
6     Pochoir_2D heat(2D_five_pt);
7     Pochoir_Array_2D(double) a(X,Y);
8     a.Register_Boundary(zero_bdry);
9     heat.Register_Array(a);
10    Pochoir_Kernel_2D(kern, t, x, y)
11       a(t,x,y) = a(t-1,x,y)
12           + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
13           + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
14    Pochoir_Kernel_End
15    for (int x = 0; x < X; ++x)
16       for (int y = 0; y < Y; ++y)
17         a(0,x,y) = rand();
18    heat.Run(T, kern);
19    for (int x = 0; x < X; ++x)
20       for (int y = 0; y < Y; ++y)
21          cout << a(T,x,y);
22    return 0;
23 }
```

Pochoir_Shape_dimD name[count] = {cells};
- `dim` is the number of spatial dimensions of the stencil.
- `name` is the name of the declared Pochoir shape.
- `count` is the length of `cells`.
- `cells` is a list of the cells in the stencil.

Declare the 2-dimensional Pochoir shape 2D_five_pt as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, e.g., for a(t,x,y), we specify the corresponding cell {0,0,0}, for a(t-1,x+1,y), we specify {-1,1,0}, and so on.
2D Heat Equation

```c
1 Pochioir_Boundary_2D(zero_bdry, arr, t, x, y)
2    return 0;
3 Pochioir_Boundary_End
4 int main(void) {
5    Pochioir_Shape_2D 2D_five_pt[6]
6        = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7    Pochioir_2D heat(2D_five_pt);
8    Pochioir_Array_2D(double) a(X,Y);
9    a.Register_Boundary(zero_bdry);
10   heat.Register_Array(a);
11   Pochioir_Kernel_2D(kern, t, x, y)
12       a(t,x,y) = a(t-1,x,y)
13           + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
14           + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
15   Pochioir_Kernel_End
16   for (int x = 0; x < X; ++x)
17       for (int y = 0; y < Y; ++y)
18           a(0,x,y) = rand();
19   heat.Run(T, kern);
20 }```

Pochoir_Shape_dimD name[count] = {cells};
- dim is the number of spatial dimensions of the stencil.
- name is the name of the declared Pochoir shape.
- count is the length of cells.
- cells is a list of the cells in the stencil.

Declare the 2-dimensional Pochoir shape 2D_five_pt as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, e.g., for a(t,x,y), we specify the corresponding cell {0,0,0}, for a(t-1,x+1,y), we specify {-1,1,0}, and so on.
2D Heat Equation

```c
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat/Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
  a(t,x,y) = a(t-1,x,y)
  + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
  + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
  return 0;
}
```

Pochoir_Shape_2D \texttt{dim} \texttt{name}[\texttt{count}] = \{\texttt{cells}\};
- \texttt{dim} is the number of spatial dimensions of the stencil.
- \texttt{name} is the name of the declared Pochoir shape.
- \texttt{count} is the length of \texttt{cells}.
- \texttt{cells} is a list of the cells in the stencil.

Declare the 2-dimensional \textit{Pochoir shape} \texttt{2D\_five\_pt} as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, e.g., for \texttt{a(t,x,y)}, we specify the corresponding cell \{0,0,0\}, for \texttt{a(t-1,x+1,y)}, we specify \{-1,1,0\}, and so on.
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
return 0;
Pochoir_Boundary_End

int main(void) {
Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
Pochoir_2D heat(2D_five_pt);
Pochoir_Array_2D(double) a(X,Y);
a.Register_Boundary(zero_bdry);
heat.Register_Array(a);
Pochoir_Kernel_2D(kern, t, x, y)
a(t,x,y) = a(t-1,x,y)
 + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
 + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1)) -
Pochoir_Kernel_End
for (int x = 0; x < X; ++x)
for (int y = 0; y < Y; ++y)
a(0,x,y) = rand();
heat.Run(T, kern);
for (int x = 0; x < X; ++x)
for (int y = 0; y < Y; ++y)
cout << a(T,x,y);
return 0;}

Pochoir _dimD name (shape);
• dim is the number of spatial dimensions in the stencil computation.
• name is the name of the Pochoir object being declared.
• shape is the name of a Pochoir shape.

Declare a 2-dimensional Pochoir object heat whose kernel function will conform to the Pochoir shape 2D_five_pt. The Pochoir object will contain all the data and operating methods to perform the stencil computation.
2D Heat Equation

```c
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
  a(t,x,y) = a(t-1,x,y)
  + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
  + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
return 0;
}
```

**Pochoir_Array** \(_{dim}D(type)\)
array\((size_{dim-1}, ..., size_1, size_0)\);

- **type** is the type of the Pochoir array.
- **dim** is the number of dimensions.
- **array** is the name of the declared Pochoir array.
- \(size_{dim-1}, ..., size_1, size_0\), are the number of grid points along each spatial dimension, indexed from 0.

Declare a 2-dimensional Pochoir array a of type double with spatial dimensions X grid points by Y grid points. The Pochoir array contains both underlying storage and requisite operating methods.
2D Heat Equation

1. Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2. return 0;
3. Pochoir_Boundary_End

4. int main(void) {
5.   Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
6.   Pochoir_2D heat(2D_five_pt);
7.   Pochoir_Array_2D(double) a(X,Y);
8.   Pochoir_Register_Boundary(zero_bdry);
9.   heat.Register_Array(a);
10.  Pochoir_Kernel_2D(kern, t, x, y)
11.   a(t,x,y) = a(t-1,x,y)
12.      + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
13.      + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
14.  Pochoir_Kernel_End
15.  for (int x = 0; x < X; ++x)
16.     for (int y = 0; y < Y; ++y)
17.       a(0,x,y) = rand();
18.  heat.Run(T, kern);
19.  for (int x = 0; x < X; ++x)
20.     for (int y = 0; y < Y; ++y)
21.       cout << a(T,x,y);
22. return 0;
23.}

Pochoir_Boundary_dimD(name, array, time, x_{dim-1}, ..., x_1, x_0)
<definition>

Pochoir_Boundary_end
- dim is the number of dimensions.
- name is a boundary function.
- array is a Pochoir array.
- time is the time coordinate.
- x_{dim-1}, ..., x_1, x_0 are the coordinates of each spatial dimension.
- <definition> is C++ code that returns values for array when it is indexed by spatial coordinates that fall outside the declared dimensions.

Declare a **boundary function**
zero_bdry on the 2-dimensional Pochoir array arr indexed by time coordinate t and spatial coordinates x and y, which always returns 0.
2D Heat Equation

1. `Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)`
2. `return 0;`
3. `Pochoir_Boundary_End`
4. `int main(void) {
5.    Pochoir_Shape_2D 2D_five_pt[6]
6.      = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7.    Pochoir_2D heat(2D_five_pt);
8.    Pochoir_Array_2D(double) a(X,Y);
9.    a.Register_Boundary(zero_bdry);
10.   heat.Register_Array(a);
11.   Pochoir_Kernel_2D(kern, t, x, y)
12.      a(t,x,y) = a(t-1,x,y)
13.        + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
14.        + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
15.   Pochoir_Kernel_End
16.   for (int x = 0; x < X; ++x)
17.      for (int y = 0; y < Y; ++y)
18.         a(0,x,y) = rand();
19.   heat.Run(T, kern);
20.   for (int x = 0; x < X; ++x)
21.      for (int y = 0; y < Y; ++y)
22.         cout << a(T,x,y);
23.   return 0;
24. }

array.Register_Boundary(bdry)
• array is a Pochoir array.
• bdry is the name of a boundary function to return a value when array is indexed by spatial coordinates that fall outside array’s declared bounds.

Register the boundary function zero_bdry with the Pochoir array a.
2D Heat Equation

1. Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2. return 0;
3. Pochoir_Boundary_End
4. int main(void) {
5.   Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
6.   Pochoir_2D heat(2D_five_pt);
7.   Pochoir_Array_2D(double) a(X,Y);
8.   a.Register_Boundary(zero_bdry);
9.   heat.Register_Array(a);
10.  Pochoir_Kernel_2D(kern, t, x, y)
11.    a(t,x,y) = a(t-1,x,y)
12.    + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
13.    + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
14.  Pochoir_Kernel_End
15.  for (int x = 0; x < X; ++x)
16.    for (int y = 0; y < Y; ++y)
17.      a(0,x,y) = rand();
18.  heat.Run(T, kern);
19.  for (int x = 0; x < X; ++x)
20.    for (int y = 0; y < Y; ++y)
21.      cout << a(T,x,y);
22. return 0;
23. }

name.Register_Array(array)
- name is a Pochoir object.
- array is a Pochoir array to register with name. Several Pochoir arrays can be registered with the same Pochoir object.

Register the Pochoir array a with the Pochoir object heat.
2D Heat Equation

Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
  a(t,x,y) = a(t-1,x,y)
  + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
  + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
  return 0;
}

Pochoir_kernel_dimD(func, time, x_{dim-1}, ..., x_1, x_0)
<definition>
Pochoir_kernel_end
• dim is the number of dimensions.
• func is the name of the kernel function being declared.
• time is the time coordinate.
• x_{dim-1}, ..., x_1, x_0 are the coordinates of the spatial dimension.
• <definition> is C++ code that defines how each grid point (as represented by Pochoir arrays at a given coordinate) should be updated as a function of neighboring gridpoints earlier in time.

Declare a kernel function kern with time parameter t and spatial parameters x and y.
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   Pochoir_Shape_2D 2D_five_pt[6]
6     = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7   Pochoir_2D heat(2D_five_pt);
8   Pochoir_Array_2D(double) a(X,Y);
9   a.Register_Boundary(zero_bdry);
10  heat.Register_Array(a);
11 Pochoir_Kernel_2D(kern, t, x, y)
12   a(t,x,y) = a(t-1,x,y)
13       + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
14       + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
15 Pochoir_Kernel_End
16 for (int x = 0; x < X; ++x)
17   for (int y = 0; y < Y; ++y)
18     a(0,x,y) = rand();
19   heat.Run(T, kern);
20 for (int x = 0; x < X; ++x)
21   for (int y = 0; y < Y; ++y)
22     cout << a(T,x,y);
23 return 0;
24 }
```

The Pochoir arrays can be initialized in whatever manner the programmer wishes. Time coordinates 0, 1, ..., depth must be initialized, where depth is the shape depth: the zero-based time dimension of the Pochoir shape (usually 1).

Initialize all points of the grid at time 0 to a random value.
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
6   Pochoir_2D heat(2D_five_pt);
7   Pochoir_Array_2D(double) a(X,Y);
8   a.Register_Boundary(zero_bdry);
9   heat.Register_Array(a);
10  Pochoir_Kernel_2D(kern, t, x, y)
11     a(t,x,y) = a(t-1,x,y)
12       + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
13       + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
14  Pochoir_Kernel_End
15  for (int x = 0; x < X; ++x)
16    for (int y = 0; y < Y; ++y)
17       a(0,x,y) = rand();
18  heat.Run(T, kern);
19  for (int x = 0; x < X; ++x)
20    for (int y = 0; y < Y; ++y)
21       cout << a(T,x,y);
22  return 0;
23 }
```

**name.Run(steps, func)**
- **name** is the name of a Pochoir object.
- **steps** is the number of time steps to run the stencil computation.
- **func** is a defined kernel function compatible with the Pochoir shape registered with **name**.

Run a stencil computation on the Pochoir object **heat** for **T** time steps using kernel function **kern**. The Run method can be called multiple times.
2D Heat Equation

1. `Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)`
2. `return 0;`
3. `Pochoir_Boundary_End`
4. `int main(void) {`
5. `Pochoir_Shape_2D 2D_five_pt[6] = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};`
6. `Pochoir_2D heat(2D_five_pt);`
7. `Pochoir_Array_2D(double) a(X,Y);`
8. `a.Register_Boundary(zero_bdry);`
9. `heat.Register_Array(a);`
10. `Pochoir_Kernel_2D(kern, t, x, y)`
11. `a(t,x,y) = a(t-1,x,y)`
    `+ 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))`
    `+ 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));`
12. `Pochoir_Kernel_End`
13. `for (int x = 0; x < X; ++x)`
14. `for (int y = 0; y < Y; ++y)`
15. `a(0,x,y) = rand();`
16. `heat.Run(T, kern);`
17. `for (int x = 0; x < X; ++x)`
18. `for (int y = 0; y < Y; ++y)`
19. `cout << a(T,x,y);`
20. `return 0;`

Elements of the Pochoir array can be read out anytime after the computation by indexing elements with time coordinate \(time+depth-1\), where \(time\) is the number of steps executed and \(depth\) is the shape depth. The \(<<\) operator is overloaded for Pochoir arrays to pretty-print their contents.

Print the elements of the Pochoir array \(a\) to standard out. The statement

\[
\text{cout} \ll a;
\]

would pretty-print the results.
Boundary Conditions

Nonperiodic zero boundary

Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

Periodic (toroidal) boundary

#define mod(r,m) (((r) % (m)) + ((r)<0)?(m):0)
Pochoir_Boundary_2D(periodic, arr, t, x, y)
  return arr.get( t,
                  mod(x, arr.size(1)),
                  mod(y, arr.size(0)) );
Pochoir_Boundary_End

Cylindrical boundary

#define mod(r,m) (((r) % (m)) + ((r)<0)?(m):0)
Pochoir_Boundary_2D(cylinder, arr, t, x, y)
  if (x < 0) || (x >= arr.size(1))
    return 0;
  return arr.get( t, x, mod(y, arr.size(0)) );
Pochoir_Boundary_End
Dirichlet boundary

Pochoir_Boundary_2D(dirichlet, arr, t, x, y)
    return 100+0.2*t;
Pochoir_Boundary_End

Neumann boundary

Pochoir_Boundary_2D(neumann, arr, t, x, y)
    int xx(x), yy(y);
    if (x<0) xx = 0;
    if (x>=arr.size(1)) xx = arr.size(1);
    if (y<0) yy = 0;
    if (y>=arr.size(0)) yy = arr.size(0);
    return arr.get(t, xx, yy);
Pochoir_Boundary_End
OUTLINE

• Functional Specification
• How the Pochoir System Works
• Optimizing Strategies
• Algorithms
• Conclusion
Phase 1 goal: Check functional correctness

Phase 2 goal: Maximize performance
If a stencil program compiles and runs with the Pochoir template library during Phase 1, then no errors will occur during Phase 2 when it is compiled with the Pochoir compiler or during the subsequent running of the optimized binary.
**IMPACT OF THE POCHOIR GUARANTEE**

- The Pochoir compiler can parse as much of the programmer’s C++ code as it is able without worrying about parsing it all.
- If the Pochoir compiler can “understand” the code, which it can in the common case, it can perform strong optimizations.
- If the Pochoir compiler cannot “understand” the code, it can treat the code as correct uninterpreted C++ text, confident that all the syntax- and type-checking was performed during Phase 1.
OUTLINE

• Functional Specification
• How the Pochoir System Works
• Optimizing Strategies
• Algorithms
• Conclusion
• Two code clones
• Unifying the handling of periodic and nonperiodic boundary conditions
• Automatic selection of optimizing strategy
  • -split-pointer
  • -split-opt-pointer
• Coarsening of base cases
Two Code Clones

• The *slow clone* handles regions that contain boundaries and checks for out-of-range grid points.
• The *fast clone* handles the larger interior regions which require no range checking.
Two Code Clones

• The slow clone handles regions that contain boundaries and checks for out-of-range grid points.
• The fast clone handles the larger interior regions which require no range checking.

During the recursive algorithm*, the fast clone is used whenever possible.

*The actual trapezoidal-decomposition algorithm decomposes the spatial grid into irregular rectangles.
Two Code Clones

- The slow clone handles regions that contain boundaries and checks for out-of-range grid points.
- The fast clone handles the larger interior regions which require no range checking.

During the recursive algorithm*, the fast clone is used whenever possible. Once the fast clone is used for a region, the fast clone is always used for its subregions.

*The actual trapezoidal-decomposition algorithm decomposes the spatial grid into irregular rectangles.
Two Code Clones

- The *slow clone* handles regions that contain boundaries and checks for out-of-range grid points.
- The *fast clone* handles the larger interior regions which require no range checking.

During the recursive algorithm*, the fast clone is used whenever possible. Once the fast clone is used for a region, the fast clone is always used for its subregions.

---

*The actual trapezoidal-decomposition algorithm decomposes the spatial grid into irregular rectangles.*
Two Code Clones

- The slow clone handles regions that contain boundaries and checks for out-of-range grid points.
- The fast clone handles the larger interior regions which require no range checking.

During the recursive algorithm*, the fast clone is used whenever possible. Once the fast clone is used for a region, the fast clone is always used for its subregions.

*The actual trapezoidal-decomposition algorithm decomposes the spatial grid into irregular rectangles.
Outline

• Functional Specification
• How the Pochoir System Works
• Optimizing Strategies
• Algorithms
• Conclusion
A *parallel space cut* [FS07] produces two *black* zoids that can be executed in parallel and one *gray* zoid that must execute in series with the black zoids.
A parallel space cut [FS07] produces two black zoids that can be executed in parallel and one gray zoid that must execute in series with the black zoids.
void walk(u, t0,t1, x0,x1,dx0,dx1, y0,y1,dy0,dy1, z0,z1,dz0,dz1) {
    int dt = t1 - t0;
    int dx = max((x1-x0), (x1+dx1*dt)-(x0+dx0*dt));
    int dy = max((y1-y0), (y1+dy1*dt)-(y0+dy0*dt));
    int dz = max((z1-z0), (z1+dz1*dt)-(z0+dz0*dt));

    if (dx>=DX_THRESH && dx>=4*sigma_x*dt) { /* cut x dimension */
        if (x1-x0 == dx) { /* cut an upright zoid */
            /* spawn black zoids */
            cilk_spawn walk(u, t0,t1, x0,x0+dx/2,dx0,–sigma_x, y0,y1,dy0,dy1, z0,z1,dz0,dz1);
            walk(u, t0,t1, x0+dx/2,x1,sigma_x,dx1, y0,y1,dy0,dy1, z0,z1,dz0,dz1);
            cilk_sync;
            /* spawn gray zoid */
            walk(u, t0,t1, x0+dx/2,x0+dx/2,–sigma_x,sigma_x, y0,y1,dy0,dy1, z0,z1,dz0,dz1);
        } else { /* cut an inverted trapezoid */
            ...
        }
    } else if (.../* cut y dimension */) {
        ...
    } else if (.../* cut z dimension */) {
        ...
    } else if (.../* cut t dimension */) {
        ...
    } else { /* call the base case */
        base_case(u, t0,t1, x0,x1,dx0,dx1, y0,y1,dy0,dy1, z0,z1,dz0,dz1);
    }
}
Suppose that we cut each spatial dimension of a $(d+1)$-dimensional zoid to produce 2 black zoids and 1 gray zoid. If there are $k \leq d$ spatial dimensions that can be cut before the next time cut, then $3^k$ subzoids in total are created.
Frigo-Strumpen evaluates all $3^k$ subzoids created by a series of space cuts on $k \geq 1$ of the $d \geq k$ spatial dimensions of a $(d+1)$-dimensional zoid in $2^k$ parallel steps.
A **hyperspace cut** simultaneously cuts as many spatial dimensions as possible. All $3^k$ subzoids created by a hyperspace cut on $k \geq 1$ of the $d \geq k$ spatial dimensions of a $(d+1)$-dimensional zoid can be evaluated in $k+1$ parallel steps.
Theorem. On a $d$-dimensional spatial grid with all spatial dimensions roughly equal to the time dimension $h$, Pochoir’s hyperspace-cutting algorithm achieves $\Theta(h^{d+1-\lg(d+2)}/d)$ parallelism, while Frigo and Strumpen’s original serial space-cutting algorithm achieves $\Theta(h^{d+1-\lg(2^{d+1})}) = O(h)$ parallelism. □

- Both algorithms have the same asymptotic cache complexity.
- Although hyperspace cuts do yield slightly better parallelism in practice, this result is largely theoretical, since $d$ is usually small.
• **Functional Specification**
• **How the Pochoir System Works**
• **Optimizing Strategies**
• **Algorithms**
• **Conclusion**
The Pochoir Project

• Pochoir version 0.5 is available under GPL from supertech.csail.mit.edu/pochoir

• Steven G. Johnson (MIT Math) has joined the Pochoir team.

• We are interested in user feedback on usability, the Pochoir language, performance issues, feature requests, and anything else.

• We would like to collect more examples and benchmarks of stencil computations.
**Future Research**

- Irregular computing domains
  - Macroscopic inhomogeneities
  - Microscopic inhomogeneities
- Boundaries that vary with time
- Automatic halos
- Heuristic autotuning
- Benchmarking
- Theoretical analyses

![Diagram of absorbing layer and Fourier accumulation](image)
Thank You!